Representing and Scheduling Looping Behavior Symbolically

Steve Haynal, Forrest Brewer
University of California, Santa Barbara
haynal@umbra.ece.ucsb.edu, forrest@ece.ucsb.edu

Abstract

This paper presents a very general, exact technique for scheduling looping data-flow graphs. In contrast to the conventional technique using loop iteration variables and integer linear programming, the new technique uses implicit symbolic automata techniques to represent the problem instance. The new technique has several advantages, such as incremental refinement, efficient variable usage and ability to accommodate practical design constraints. A small case study demonstrates the flexibility and viability of this technique.

1. Introduction

Implicit symbolic techniques have progressed tremendously in recent years and have gained mainstream acceptance in formal model checking and verification. In contrast, conventional high-level synthesis techniques have relied on ad-hoc modeling and on heuristics more akin to compiler technology than to hardware synthesis. In this work, our goal is to apply systematic approaches and technologies borrowed from model checking to problems of high-level synthesis. To these techniques, we add the expressive capabilities of non-deterministic finite automata. NFA's have efficient implicit representation and are used throughout the formulation to represent information behavior, protocols and interfaces, and arbitrary design subsystems. Using these tools, we reformulate the loop-dominated data-flow scheduling problem as constrained search among all implicit automata-based executions.

This technique has potential as well as some drawbacks. The potential lies in the ability to accommodate arbitrary automata-based constraints on timing, behavior, capacity, and other chosen properties of the design while maintaining systematic solutions. Surprisingly, many types of scheduling optimizations, such as loop pipelining, are captured implicitly by the technique. The drawbacks are with representation growth. However, by careful choice of encoding and representation we have shown that this technique is practical to at least the scale offered by alternative exact methods and indeed the performance exceeds that of comparable published heuristics[3]. Furthermore, the systematic formulation provides a formal route to abstraction and hierarchical representation of larger designs.

In this paper, we present a brief example followed by a technical description of model construction in sections 1.2-2.2. An overview of model exploration is described in sections 3.1-3.3. Finally, a small case study is presented as results in section 4.

1.1. Motivational Example

Algorithm 1 is an example functional description. For each loop iteration, the subsystem implementing this algorithm reads three input values and writes one result. Furthermore, an earlier addition requires the result of the multiplication, `rv2`, and hence a data dependency between different loop iterations exists. Consequently, `rv2` must be initialized upon entering the loop.

Algorithm 1: Example functional description

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

Figure 1 represents algorithm 1 as a data flow graph, DFG. Each vertex represents an operation. Each directed edge represents a data dependency. The reverse edge from *v2* to *v1* represents a data dependency between different loop iterations. (Read/write operations and associated data dependencies were not included. Interface protocols, ports, and their dependencies *are* included in the case study.) All operations or DFG vertices must be executed once per loop iteration in the class of problems presented here. Some behaviors require operations to execute only once, as in a pre-computation of a coefficient. This is modeled by combining cyclic models from this paper with earlier acyclic models[2].

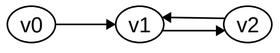

Figure 1: Data flow graph of example

Correctly scheduling this DFG requires assigning each operation to a time-step while observing several criteria. First, all data dependencies must be observed. Second, resource bounds, such as one available adder, must be adhered to. Finally, a scheduling objective, such as minimize iteration latency, typically guides schedule selection.

If one single time-step adder and one single time-step multiplier are available, then the example's only minimum iteration latency schedule is shown in figure 2. 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. This *loop pipelining* is possible because operations from successive iterations may overlap as seen with *v2* and *v0*.

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

Figure 2: Minimum iteration latency schedule

1.2. Automata-Based Solutions

Symbolic scheduling first constructs a composite modeling automaton, **CMA**, that encapsulates all causal schedules for a given DFG. Once **CMA** is constructed, exploration techniques are used to find particular schedules meeting some objective function. For the example, the desired **CMA** is *explicitly* shown in figure 3. (In practice, **CMA** is represented implicitly with ROBDDs.) Each *edge* in this non-deterministic state graph represents a time-step. Operations assigned to a time-step are identified through edge labeling. Loop pipelining is possible because successive iterations are distinguished. If an operation is labeled with no '~', such as *v0*, then *v0~* represents the same operation in the successive iteration and vice versa. Operations labeled with '~' are referred to as *odd* iteration operations and those without '~' as *even*. In figure 3, the minimum iteration latency schedule is highlighted with dashed edges. Two iterations (for both iteration senses) are scheduled in one complete traversal of this cyclic dashed path. Edges denote scheduled activities while states encode in which sense operands currently exist in the design. In general, any cyclic path through **CMA** which executes all operations is a valid steady-state schedule of the loop.

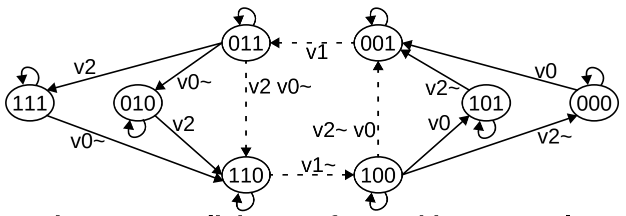

Figure 3: Explicit CMA for working example

2. Constructing Composition Models

The first step in constructing **CMA** is to model each DFG operation with a small *modeling NFA*, **MA**. In the working example, a single time-step operation such as *v0* is modeled by the automaton shown in figure 4. Each labeled *edge* identifies the scheduling of *v0* in one particular iteration sense. Each *state* encodes what iteration sense was last scheduled. For this particular choice of state encoding, '0' represents *v0* last scheduled in the odd iteration sense and vice versa for '1'. In practice, **MA** often need to be more complex than figure 4. **MA** have been generalized to represent complex function units (pipelined, etc.) as well as local and global sequential constraints (IO protocols).

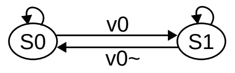

Figure 4: Single cycle modeling NFA

Every operation in the DFG is modeled by its own instance of a **MA**. The Cartesian product of all **MA** form an initial **CMA**. Although figure 3 shows **CMA** after edge pruning, the state of each **MA** is still represented by one bit in each **CMA** state vector ordered *v2,v1,v0*. To illustrate, consider the transition from state 100 to state 001. The bit representing *v2* changes from 1 to 0 which identifies that *v2~* is scheduled on that edge. Also, the bit representing *v0* changes from 0 to 1 which identifies that *v0* is scheduled.

2.1. Dependency and Capacity

Every path in the example traverses a *v0* edge before a *v1* edge since *v1* depends on the result produced by *v0*. In the initial completely connected **CMA**, an edge exists from state 000 to state 010. However, the transition from 0 to 1 in the *v1* bit position indicates that operation *v1* is scheduled yet *v0* is a 0 or '~'. The result operand that *v1* depends upon is not available in the correct iteration sense and *v1* can not be scheduled. This edge and other acausal edges are pruned from **CMA** by dependency refinements.

CMA contains a symbolic transition relation $\Delta \in$ **CMA** where each transition, either global or for local **MA**, may be represented by a present state, next state pair, (*ps*,*ns*). In the example, *v1*'s even *intra*-iteration dependency on *v0* is built by the implication $(0,1)_{v1} \Rightarrow (1,-)_{v0}$. In words, if the **MA** for operation *v1* takes its even *input* transition (0,1), then the **MA** for operation *v0* must have its result *known* (1,-) in the present state and in the even iteration sense. This same intra-iteration dependency must also be built in the *odd* iteration sense. Furthermore, *v1*'s even *inter*-iteration dependency on *v2* is built by the implication $(0,1)_{v1} \Rightarrow (0,\text{-})_{v2}$. This too must be built for both iteration senses. Dependency implications are built for all edges in the DFG and intersected with **CMA**'s transition relation Δ to prune all acausal edges.

Operation *v0* has no dependencies. It is possible that *v0* may schedule in the even iteration sense and then in the odd iteration sense before *v1* has a chance to use the first even iteration result. A capacity implication ensures that a particular result is consumed by all dependents before the next iteration result may be generated. The capacity implication of *v0* on *v1* is built as $(0,1)_{v0} \Rightarrow (-,0)_{v1}$.

2.2. Resource Bounds

Since both *v0* and *v1* require an adder resource and only one adder resource is available, it is illegal to assign these operations to the same time-step. Enforcing this corresponds to removing edges from **CMA** where both *v0* and *v1* are *active* (bits for *v0* and *v1* are changing simultaneously). This is enforced by enumerating all combinations of 0 up to *bound active* transitions for a particular resource class and intersecting this filter with Δ to prune all edges violating resource bounds. Although this constraint appears to be exponential, it requires only $2 \times n \times \mathit{bound}$

nodes, where *n* is the number of operations requiring this resource class, when represented as a ROBDD. This *concurrency* constraint may be generalized to limit interconnect, guide multiplexing and limit required local storage.

3. Composition Model Exploration

The dashed path of figure 3 shows a minimum iteration and control-step latency schedule. Composition model exploration finds a set of all such minimum length repeatable paths using symbolic techniques. We begin with a *loop-cut set* which includes states guaranteed to be on the optimal steady-state loop path. (Initially, we ignore a preamble in favor of a true optimal steady-state solution.) To determine a loop-cut set, first realize that any valid steady-state solution will execute every loop operation at some time-step during one iteration. Consequently, we may chose any operation to serve as the *loop cut*. All edges in **CMA** where the loop-cut operation is scheduled in the even sense form an even transition loop-cut set. All successor states for transitions in this set create the loop-cut starting state set, *LCS*. For example, picking operation *v2* as the example's loop cut results in *LCS* = {110,111}.

3.1. Candidate Loop Schedules

Candidate loop schedules are generated through symbolic exploration of **CMA**. To facilitate this, a termination set, *LCS~*, is determined. Remember that two symmetric iterations of the example are represented by the dashed path in figure 3. In fact, states in the even iteration sense have a symmetric *dual* in the odd iteration. For example, states 110 and 001 are duals. Because of this choice of encoding, a dual state is found by bitwise complement. The dual of *LCS* is *LCS~* = {001,000}.

Since only a steady-state solution for *one* iteration of the loop is desired, exploration proceeds until states in *LCS~* are reached. At each step of this breadth first ROBDD traversal, a *time-step* set is preserved. All time-step sets form an *ensemble time-step* set. For the example, an ensemble time-step set is $\{110,111\}_0$, $\{100\}_1$, $\{000,001,101\}_2$. When states in *LCS~* are reached, a reverse pruning leaves only paths from some states in *LCS* to some states in *LCS~*. In the example, the pruned loop candidate ensemble time-step set is $\{110\}_0$, $\{100\}_1$, $\{000,001\}_2$.

3.2. Loop Candidate Closure

Not all of the loop candidates are repeatable. The example's loop candidate ensemble time-step set contains a path from 110 to 100 to 000. Unfortunately, there is no path from 000 which completes an iteration in two steps. (This can be determined by considering the dual.) Loop candidate closure is a fixed-point pruning of the loop candidate ensemble. Forward and backward pruning proceed until time-step set 0 equals (as duals) time-step *n*. After loop candidate closure is applied to the example, the closed loop ensemble time-step set is $\{110\}_0$, $\{100\}_1$, $\{001\}_2$. If loop candidate closure fails, another time-step set is added to an unpruned ensemble time-step set copy and loop candidate closure is attempted again.

Figure 5 illustrates loop closure abstractly. The sets *LCS* and *LCS~* are equal although in opposite senses. There are two paths, *a* to *a~* and *d* to *d~*, which are directly repeatable after five time-steps. They are repeatable since by symmetry paths also exists from *a~* to *a* and from *d~* to *d*. These represent steady-state schedules with iteration and control-step latency of five. A schedule which favors minimizing iteration latency at the expense of control depth is encapsulated in the path from *b* to *c~* and by symmetry from *c~* back to *b*. This path has average iteration latency of only 4 but requires more control steps.

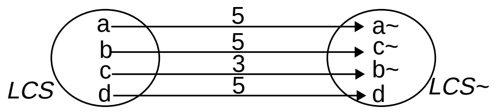

Figure 5: Closed paths from *LCS* to *LCS~*

3.3. Single Loop Schedules

Although **CMA** and the closed loop ensemble time-step set contain a wealth of schedules to choose from, it is sometimes desirable to find a path from some state *s* directly to its dual *s~* as it represents a FSM with number of control steps equal to the minimum iteration latency. To do this, we pick an arbitrary state *s* from time-step 0 of the closed loop ensemble and attempt loop closure with this single state as *LCS*. If loop closure is successful, a closed path from *s* to *s~* exists. In the worst case, this procedure may be costly as it explicitly examines every state from time-step set 0. Fortunately, in all examples we have scheduled, the solution density is high enough that a successful candidate was found between one to twenty attempts.

4. EWF Case Study

The elliptic wave filter, EWF, is a common looping DFG benchmark reported in the literature[2][4][6] with 8 multiplication and 26 addition operations. We use this benchmark as a case study to demonstrate how a designer might interactively use symbolic scheduling. (Readers interested in a more complete set of results should refer to paper [3].)

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. Internally, this IP block contains an optimized 3-stage pipelined floating-point multiplier, a floating-point ALU, a small 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.

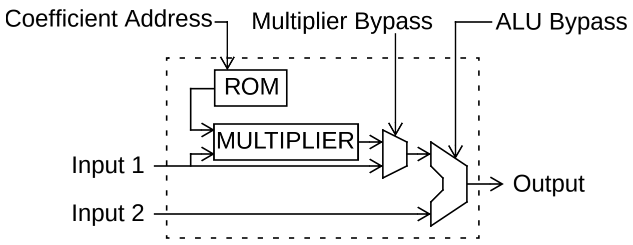

Figure 6: IP block for reuse

The designer codes the EWF algorithm at an abstract level (< 100 lines) and specifies appropriate **MA**s (again < 100 lines). Symbolic scheduling accepts this input and determines all minimum iteration latency schedules. Table 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. 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 1: Constrained IP-block results

IP Blocks	Iteration Latency	CPU Seconds
1	30	3.2
2	18	2.7
3	16	2.5
4	16	2.4

Figure 7 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 is connected to bus structure 1, permits communication to and from the function blocks via the register bank.

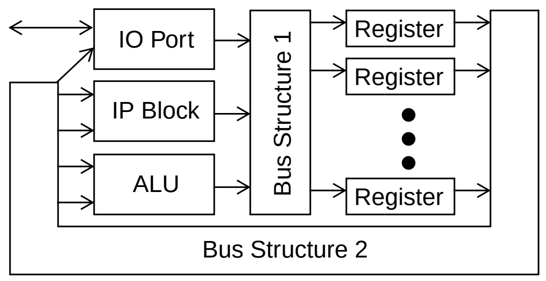

Figure 7: Target high-level architecture

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 symbolic scheduling provide the data shown in table 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 tradeoff decision and opts to reduce interconnect at the expense of iteration latency by choosing the 3/2 bus solution shown in bold. (Even with 3 busses, both function blocks may simultaneously begin execution as a single operand may feed multiple function block inputs.) Once the designer decides on a final constraint configuration, symbolic scheduling 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.

Table 2: Constrained registers & busses results

Bus 1	Bus 2	Registers	IO Ports	Iteration Latency	CPU Seconds
-	-	8	1	Impossible	2.0
-	-	9	1	18	2.9
-	-	10	1	18	3.1
-	1	9	1	30	3.4
-	2	9	1	18	2.9
-	3	9	1	18	3.0
2	2	9	1	29	3.2
3	**2**	**9**	**1**	**20**	**3.1**
4	2	9	1	18	3.1
5	2	9	1	18	3.0

5. Conclusions

A systematic symbolic model is presented for exact scheduling of looping DFGs. First, an automata-based model incorporating numerous practical design constraints is constructed. Next, all shortest repeating paths (high-throughput execution sequences) are found using well developed symbolic model-checking techniques. A case study demonstrated the usefulness and application of symbolic scheduling. Parallel work with symbolic scheduling has addressed control even when present in looping or pipelined behavior. Ongoing and future work focuses on finding best *average* latency schedules and other desired schedules using improved model-exploration techniques.

6. References

[1] L.-F. Chao, A. LaPaugh and E. Sha, "Rotation Scheduling: A Loop Pipelining Algorithm", *IEEE Trans. CAD/ ICAS*, vol. 16, no. 3, pp. 229-239, Mar. 1997.

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

[3] S. Haynal and F. Brewer, "Symbolic Automata-Based Scheduling for Looping DFGs", *IEEE Trans. CAD/ICAS*, submitted Nov. 1999.

[4] I. Radivojevic and F. Brewer, "On Applicability of Symbolic Techniques to Larger Scheduling Problems", *Proc. EDAC-96*, pp. 48-53, 1996.

[5] 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.

[6] 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*, pp. 166-183, Feb. 1996.