Verification of Basic Block Schedules Using RTL Transformations*

Rajesh Radhakrishnan, Elena Teica, and Ranga Vemuri

ECECS Department, ML. 30
814 Rhodes Hall, University of Cincinnati
Cincinnati, OH 45221-0030
{rradhakr,eteica,ranga}@ececs.uc.edu, Fax: 513-556-3025

Abstract. We present an approach to aid in debugging/development of scheduling algorithm implementations. Our technique makes use of a sequence of a correctness-preserving RTL transformation called *Register Transfer Split (RTS)*, to collectively perform the same task as that of a scheduler. Violation of the transformation precondition signals an error and the sequence of RTS transformations applied so far forms a trace which can be used for debugging purposes.

1 Introduction

Researchers have addressed the problem of creating bug-free synthesis systems into separately verifying each synthesis stage. In this paper, we specifically deal with the verification of the scheduling stage.

Lock et al [1] captured the scheduling result in the form of a table and checks for correctness inside the HOL theorem prover. Ashar et al [2] used model checking and symbolic simulation to check the signal correspondences between RTL and behavior. Importantly, all arithmetic operations are uninterpreted and loops are handled via identifying loop invariants. In Eveking et al [3], failure to transform the behavioral description towards the scheduled result signals an error. Naren et al [4] proved the correctness of the force-directed list scheduler (FDLS) algorithm in PVS and embedded the correctness conditions developed during the exercise as program assertions in the implementation.

We do not handle constraint violations in the scheduler and hence only consider *legal* schedules and not optimal ones [5]. Also, our approach is applicable only for schedules for sequences of straight-line code or *basic blocks*.

In Sect. 2 we introduce models for a register transfer and the RTS transformation. Section 3 discusses our methodology. The conclusion is presented in Sect. 4.

2 Models

We use the models of a register transfer and the RTS transformation from [6], where a completeness proof for a set of RTL transformations (including RTS)

* This work was sponsored by DARPA, monitored by US Army Ft. Huachuca under contract number DABT63-96-C-0051 and NSF (grant number 9634462).

T. Margaria and T. Melham (Eds.): CHARME 2001, LNCS 2144, pp. 173–178, 2001.

is presented. *Completeness* of a set of behavior-preserving RTL transformations means: for any two behaviorally-equivalent RTL designs, by applying a finite sequence of these correctness-preserving transformations we can move from one design to the other.

2.1 Register Transfer

A register transfer maps a set of source registers to a set of destination registers. It denotes the activity performed at a certain part in the data path at a *unique control step*. We use the term *expressions* to collectively refer to operators, registers and their interconnections. Let E refer to the set of expressions, OP to the set of operators in E and REG to the set of registers in E.

The data path consists of a set of operators, a set of registers and the interconnect between them. Using the notations above, we can now define the data path (DP) as the following tuple:

$$DP = (E, OP, REG) \tag{1}$$

The activity inside a data path during a register transfer RT is represented by a subset of expressions from E. The interconnect between these expressions are determined by the computations scheduled to be performed in the data path at the control step defined by RT.

Definition 1 *A register transfer RT associated with a data path is a tuple of the form:*

$$RT = (E_{RT},\ REG_{RT}^{out},\ f_{op} :\ OP \rightarrow (E \times E),\ f_{reg} :\ REG_{RT}^{out} \rightarrow E) \tag{2}$$

where $E_{RT} \subseteq E$ and $REG_{RT}^{out} \subseteq REG$ is the set of output registers in RT. f_{op} and f_{reg} define the interconnect between expressions of the data path at the control step corresponding to RT as follows: function f_{op} maps an operator to a pair of expressions (for each of the two source expressions of the operator) and function f_{reg} maps an output register to an expression (its input).

2.2 Well-Formed Register Transfer

In our model, we do not allow register transfers to contain combinational cycles, floating inputs for operators and registers or concurrent operations to be performed on the same hardware resource.

To formally define such requirements we first introduce the definition of an *ancestors* set *Anc* for an expression *e* (operator or register) as the set of all expressions which are connected via a *direct path* to *e*.

Definition 2 *The ancestors set Anc of an expression e of a data path = (E, OP, REG), with respect to a mapping function f_{op} : OP → (E × E), is defined recursively as:*

$$Anc(e) = \begin{cases} \emptyset & e : register \\ Anc(f_{op}(e)\text{`}1) \cup\ Anc(f_{op}(e)\text{`}2)\ \cup\ \{f_{op}(e)\text{`}1, f_{op}(e)\text{`}2\} & e : operator \end{cases} \tag{3}$$

where $f_{op}(e)$‘1 and $f_{op}(e)$‘2 represent the first and second projections of f_{op} respectively.

Definition 3 *A register transfer RT is said to be* **well-formed** *if:*

1. $\forall(e \in E_{RT}) : Anc(e) \subseteq E_{RT}$
   The ancestors set Anc for each operator in E_{RT} must also be present in E_{RT}.
2. $\forall(e \in E_{RT}) : e \notin Anc(e)$
   There are no combinational cycles in an RT.
3. $image(f_{reg}) \subseteq E_{RT}$
   The source of each output register in REG_{RT}^{out} must also be present in E_{RT}.
4. $\forall(e_1, e_2 \in OP) : (e_1 = e_2) \Rightarrow ((f_{op}(e_1)\text{`}1 = f_{op}(e_2)\text{`}1) \wedge (f_{op}(e_1)\text{`}2 = f_{op}(e_2)\text{`}2))$
   Concurrent operations are performed on different hardware resources.

2.3 Register Transfer Split (RTS)

Transformation: **RTS** (RT,*split_set*)
Operation: *Split register transfer RT into two register transfers RT_1 and RT_2 by inserting new temporary registers between them. RT_2 contains the sub-image of RT induced by the operations present in* `split_set` *and RT_1 contains the rest.*
Precondition:

i. Dependencies must be preserved as the operators are assigned to different register transfers.

Body:

Step 1. Copy the sub-image in *RT* induced by the operators in *split_set* and place it in a new register transfer RT_2. The remaining sub-image (induced by any remaining operators) in *RT* goes into another new register transfer RT_1.

Step 2. The output of RT_2 is connected to the inputs of the temporary registers. The outputs of the temporary registers are connected to the inputs of components of RT_1. *temp_set* represents the set of temporary registers.

No change is made to RT if `split_set` has only one operator. The application of the RTS is deemed correct if its precondition, called the *Well-Formedness* precondition, is satisfied.

Well-Formedness Precondition (WFP):

$$WFP(RT, split\_set) \stackrel{\Delta}{=} \forall(e \in split\_set) : Anc(e) \in split\_set \tag{4}$$

$$WFP(RT, split\_set) \Leftrightarrow comp\_behavior(CG) = comp\_behavior(RTS(CG, RT, split\_set)) \tag{5}$$

Equation 4 states that the ancestor operator(s) *Anc(e)* must also be in the *split_set*. This statement corresponds to the first condition for a well-formed register transfer. Equation 5 states that the computational behavior (*comp_behavior*) of the design is preserved if there is no WFP violation.

Thus, the application of RTS is sound if and only if the precondition is satisfied and further, is complete with respect to the scheduling task [6]. Hence the task performed by most scheduling algorithms (excluding those that move code across basic blocks) can be viewed as a sequence of RTS transformations.

3 Methodology

Our methodology is based on the precondition-based correctness of RTS and on the completeness of RTS transformations to perform the scheduling task. Figure 1 illustrates our approach. The primary components of our system are the

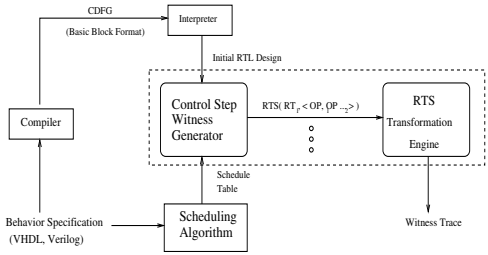

Fig. 1. Verification Methodology

Control Step Witness Generator (CSWG) and the *RTS transformation Engine* (shown in the dotted box in the above figure). The CSWG takes an initial RTL representation of the behavioral design and the *schedule table*. The schedule table is generated by the scheduling algorithm under test and contains a mapping of operations to control steps. For each control step in the schedule table, the CSWG generates an RTS transformation with the operators scheduled at that control step as one of its arguments. All applications of RTS (with its arguments) are written to a log file called the *Witness trace*. If any WFP violation occurs, it too is written to the witness trace along with the offending operators in question. If a violation does occur, the trace file shows the exact steps taken by the scheduler, hence can be used to understand why it scheduled operators that violated the WFP.

An *initial RTL* design of a behavioral specification is obtained by creating a register transfer for every basic block in the behavior. We assign a structural operator/register to each operation/carrier respectively in the behavior. The control flow remains the same. This initial RTL is successively modified by the RTS transformations based on the schedule table. Figure 2 shows an initial RTL being split based on the schedule table using RTS transformations.

3.1 Results

Our system can be used for debugging an existing (or new) implementation of a scheduling algorithm. If the operators are referenced via a special naming scheme in the schedule table, this can be provided as another input to our system (not shown in the Fig. 1).

We used an existing implementation of Force-Directed List Scheduler (FDLS) [5] from DSS [7]. A bug was seeded in at the dependency graph creation routine used by FDLS to get successor and predecessor node information. The following

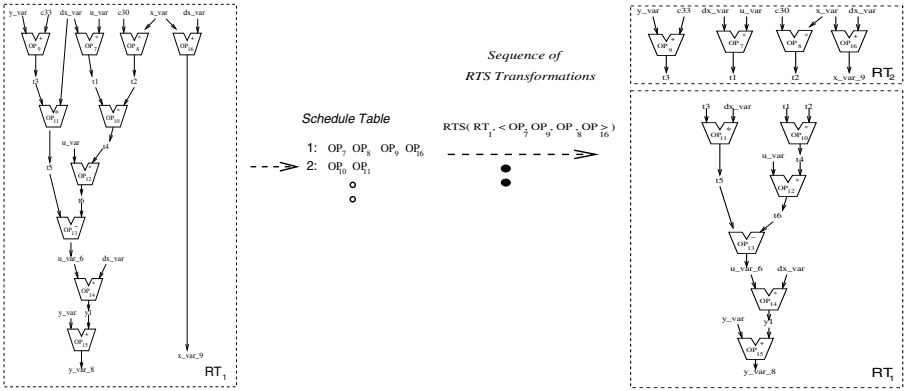

Fig. 2. Sample Flow

examples were run through the scheduler and the resulting schedule table was input to our verification framework. The run times for verifying each design are shown.

The Discrete Cosine Transform (DCT) is the largest example we tried. FFT2D is an 8-point Fast Fourier Transform (FFT) butterfly network composed of two 4-point FFTs (FFT1D). Linear System Solver (LSS) [8] solves a linear system of equations using matrix inversion. The Elliptic Wave Filter (Ellip) and Differential Equation solver (Diffeq) are taken from the High Level Synthesis Workshop Benchmarks'92 [9]. The examples were run on a Sun Enterprise 2. The bug in

Table 1. Information on the synthesis benchmarks used

Design Examples	Total Operations	Schedule Length	CPU Time
1. DCT8x8	1922	72	5.37
2. FFT1D	32	14	0.14
3. FFT2D	104	26	0.39
4. LSS	28	32	0.52
5. Ellip	26	18	0.2
6. Diffeq	10	4	0.08

the scheduler resulted in WFP violations in the Diffeq and Ellip examples. The witness trace file for the Diffeq example is shown below:

Witness Trace File for Diffeq Example:

```
RTS < Blk_3, 6 >
RTS < Blk_4, 7 8 9 11 12 15 16 > RTS WFP Violation: [ 10 13 14 ]
```

From Fig. 2, operation 10 which is the successor of operation 12 was omitted. Also operation 15 was scheduled before operation 14 which in turn required operation 13. There were no WFP violations in the other designs as they were legal schedules, that is, no dependencies were violated.

Though we considered only FDLS, any of the other scheduling algorithms whose output can be captured in the form of a schedule table, could have been used. Our approach cannot be directly applied to algorithms like percolation scheduling [10] which perform scheduling across basic block boundaries. However a schedule table can be extracted after the code motions are performed.

4 Conclusion

We presented an approach to verifying legal, basic block schedules produced by a scheduler in a high-level synthesis system. Based on the schedule table, a sequence of *RTS* transformations is used to perform the same task as the scheduler. The *Well-Formedness* precondition of RTS checks the correctness of the input to RTS. If a precondition is violated then the sequence of transformations applied so far forms a trace and can be used for debugging purposes.

References

[1] T. Lock, M. Mendler, and M. Mutz. Combined Formal Post- and Presynthesis Verification in High Level Synthesis. In *Formal Methods in Computer-Aided Design*, pages 222–236, 1998.

[2] P. Ashar, A. Raghunathan, A. Gupta, and S. Bhattacharya. Verification of scheduling in the presence of loops using uninterpreted symbolic simulation. In *International Conference on Computer Design (ICCD)*, 1999.

[3] H. Eveking, H. Hinrichsen, and G. Ritter. Automatic Verification of Scheduling Results in High Level Synthesis. In *Design, Automation and Test in Europe (DATE)*, 1999.

[4] N. Narasimhan, E. Teica, R. Radhakrishnan, S. Govindarajan, and R. Vemuri. Theorem Proving Guided Development of Formal Assertions in a Resource-Constrained Scheduler for High Level Synthesis. In Andreas Kuehlmann, editor, *International Conference on Computer Design (ICCD)*, Austin, TX, October 1998. IEEE Computer Society.

[5] R. Walker and S. Chaudhuri. Introduction to the scheduling problem. *IEEE Design and Test*, 12(2):60–69, 1995.

[6] E. Teica and R. Vemuri. A Mechanical Proof of Completeness for a Set of Register-level transformations. Technical Report 257/05/01/ECECS, University of Cincinnati, 2001.

[7] J. Roy, N. Kumar, R. Dutta, and R. Vemuri. DSS: A Distributed High Level Synthesis System. In *IEEE Design and Test of Computers*, volume 9, pages 18–32, 1992.

[8] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishers, 1996.

[9] N. Dutt. High Level Synthesis Workshop Benchmarks'92.

[10] G. De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, 1994.