Software Synthesis using Timed Decision Tables

Sumit Gupta
Rajesh Gupta
http://www.ics.uci.edu/~iesag

Technical Report #99-01
Department of Information and Computer Science
University of California, Irvine, CA 92697-3425

January 1999

Abstract

Timed Decision Tables (TDTs) have been used earlier for modeling behavioral descriptions, applying presynthesis optimizations for efficient circuit synthesis and HDL restructuring. We describe here work that optimizes TDT models for generation of software in a high-level programming language. The optimization for software synthesis is targeted at reducing the numbers of conditionals and actions in the generated code. The TDT-based optimizations and software synthesis is implemented in C++. Experimental results on a set of examples show significant reduction in the number of conditionals checked.

Table of Contents

List of Figures

1 Introduction

Software generation and optimization has been a focus of recent work on embedded systems due to the increasing role of software in most applications [1, 7, 8]. The emphasis of embedded software has been primarily on improved code generation techniques for diverse architectures [2, 3, 4], on optimization of memory resources [10, 5] and on address optimization [11, 12]. However, as system design process incorporates more abstract models (such as StateCharts, UML etc) there exists a need to generate software from these models in a high level language (HLL) to handle increasingly complex software. The high-level language description can then be input to multilevel optimization techniques as is the case in conventional compiler frameworks.

Software synthesis refers to the process of generation of high-level language code from abstract (behavioral) models. Prior work has been done on code generation from dataflow or synchronous dataflow models [9, 14, 15]. Our work builds upon the Timed Descision Table (TDT) model which has been used for hardware description language (HDL) based optimizations and HDL code restructuring [13]. This model captures behavioral system descriptions which can then be used for hardware and software synthesis. Behavioral optimizations are attractive for their potential to apply optimization techniques on a broader scope (e.g. beyond basic blocks, loops) while keeping the computation costs low.

2 Timed Decision Tables

The TDT model has been explained at length in [13]. TDT models in general can be behavioral or structural. In a structural TDT, actions represent RTL operations whereas behavioral TDTs may use multi-cycle operations. A TDT description consists of three tables: (1) a *control flow table*, which captures the control-flow for a behavioral model, (or specifies a controller for a structural TDT); (2) a *dependency table* that captures the dependency among operations for a behavioral level model, (or data dependency among components in a structural model); (3) a *delay table*, which specifies the

Tdt_0 :

a_0	a_1	a_2	a_3	tdt_2				
					c_0	Y	Y	N
					c_1	Y	N	-
-	ŝ	m	m	m	a_0	1	0	0
s	-	m	m	m	a_1	1	0	0
m	m	-	ŝ	m	a_2	0	1	0
m	m	s	-	m	a_3	0	1	0
m	m	m	m	-	tdt_2	0	0	1

Figure 1: *A TDT description*

operation delay. In the sequel, we consider only behavioral TDTs.

An example of a TDT is given in Figure 1. The *control flow* table consists of four quadrants: (1) the condition stub is the set of conditions; (2) the condition entries indicate possible conjunctions of conditions as rules; (3) the action stub is the list of actions (4) the action entries indicate the actions that are active for a certain rule. So a *rule* is a column in the right quadrants of the table, where the condition entry quadrant corresponds to the decision part of the rule and the action entry quadrant to the action part of the rule.

The *dependency matrix* represents the dependencies among actions. Dependencies can be one of, *serial predecessor (s)*, *serial successor (ŝ)*, *concurrent (c)* and *mutually exclusive (m)*. The *delay table* is used to model the execution delay associated with a datapath operation, to distinguish between the timing semantics of signals and variables (as given in VHDL) and to specify communication protocols such as to represent bus delays etc. The delay table completes the TDT description by incorporating information sufficient to generate timing accurate HDL code from TDTs.

2.1 Hierarchy in TDTs

Hierarchy is represented in TDTs by allowing an action to be another TDT. The TDT shown in Figure 1 is obtained by flattening the hierarchy of TDTs shown in Figure 2. The two actions, Tdt_1 and Tdt_2 called in Tdt_0 are TDT models, of which Tdt_1 is also shown in the figure. Hierarchy is necessary since it allows modularity in the specification and more importantly, prevents the explosion in rules for complex systems. However, since a structure good for

Tdt_0 :

tdt_1	tdt_2			
		c_0	Y	N
-	m	tdt_1	1	0
m	-	tdt_2	0	1

Tdt_1 :

a_0	a_1	a_2	a_3			
				c_1	Y	N
-	$\hat{s}$	m	m	a_0	1	0
s	-	m	m	a_1	1	0
m	m	-	$\hat{s}$	a_2	0	1
m	m	s	-	a_3	0	1

Figure 2: *Hierarchy of TDTs*

conceptualization may not always be good for synthesis, the hierarchy of the TDTs can be re-structured based on resource sharing and frequency of calls to shared code [13].

A set of behavior preserving transformations have been defined for the TDT model [13]. Various column transformations and row transformations in both the condition and action quadrants have been developed. Column operations are column merging and elimination and action entry modification. Row operations in the condition quadrant are row elimination, insertion, negation, encoding and swapping. Similarly, row operations in the action quadrant are row merging, elimination and swapping. The TDT model also facilitates easy identification of duplicate actions and subsequent action sharing.

3 Software Synthesis from TDTs

Synthesis of software from TDTs requires a selection of a schedule of operations and subsequent HLL code generation according to a chosen style [14, 15]. Scheduling a TDT eliminates concurrent "c" entries from the dependency table. Operation scheduling is an important aspect of the software synthesis process. However, for a given scheduling strategy, choice of coding style has a significant impact on the quality of the eventual code. Our approach to scheduling is based on relative scheduling that generates software

Tdt_1:

c_1	Y	Y	Y
c_2	N	Y	Y
c_3		N	Y
a_1	1	1	1
a_2		1	1
a_3			1

Tdt_2:

c_1	Y	Y	Y
c_2		Y	Y
c_3			Y
a_1	1		
a_2		1	
a_3			1

Figure 3: *Two types of TDTs*

Generated from Tdt_1

```
if c1 then
  if c̄2 then
    a1;
  else  if c̄3 then
          a1; a2;
        else
          a1; a2; a3;
```

Generated from Tdt_2

```
if c1 then
   a1;
   if c2 then
      a2;
      if c3 then a3;
```

Figure 4: *Software corresponding to the TDTs*

as a set of threads that begin with non-deterministic (anchor) actions [16]. Here, we focus on the coding style. TDTs can be used directly for generation of HLL code by a mechanistic translation of its operational semantics [13]: each rule corresponds to a condition clause determined by the condition column and an action part that is sequenced according to the dependency table and invoked with the delay determined by the delay table. That would be applicable in case the rules are invoked disjointly. Consider the example in Figure 3 (we show here only the control flow tables in the TDTs). Software synthesis from the two tables is shown in Figure 4.

The HLL code from both Tdt_1 and Tdt_2 has three conditional checks corresponding to the three rules in the two TDTs. However, the nesting of condition checks and reduced number of action activations from Tdt_2 leads to shorter assembly code from most compilers. Some code optimization (for instance, code motion across conditionals) can optimize the code from Tdt_1, for instance, by moving action a_1 to just after the first IF statement.

In general, conditions in HLL code adversely affect the quality of compiled code generated, particularly, for modern deeply pipelined processors. Code

c_1	Y	Y	N
c_2	N	Y	X
a_1		1	
a_2	1		
a_3	1		1

quality can, therefore, be improved by reducing conditionals. Also, in general, it is more efficient to use nested conditionals since these reduce the condition checking work in the generated code.

For the purpose of software synthesis, the TDT optimization goals are: (a) to minimize the number of conditionals; (b) within each conditional, minimize the number of Boolean tests, i.e. minimize the number of condition entries in the TDT; (c) and finally to minimize action duplication or maximize action sharing by minimizing number of action entries in the TDT. In the absolute minimum case, after dead code elimination, each condition is checked only once and each action is invoked only once. However, this is not always possible since it depends on the semantics of the target HLL.

Consider the example TDT shown in Figure 5 along with corresponding *Verilog* (HDL) and *C* (HLL) descriptions. The Verilog code uses a control jump statement, *disable*, to avoid repeating action a_3, whereas the C code is completely structured and repeats the action a_3 in two control flow paths. The coding style is affected by the type of the TDT used (Figure 4) and by the choice of HLL (Figure 5). We consider the TDT types in the following section.

3.1 Algebraic Model for Control Flow in TDTs

For each condition variable 'c' in a TDT, we define a positive condition literal, l_c, corresponding to a 'Y' value in a condition entry and a negative condition literal, $l_{\bar{c}}$, corresponding to a 'N' value. The '$\cdot$' operator among action and condition literals represents a conjunction operation, which is both commutative and associative.

A TDT is a set of rules consisting of a condition part, which determines when the rule is selected, and an action part, which lists the actions to be

```
begin: blockA                  if (C1)
  if (C1)                        {
    begin                          if (C2)
      if (C2)                        a1;
        begin                      else
         a1;                         {
         disable blockA;              a2;  a3;
        end                          }
      a2;                          }
    end                        else
  a3;                            a3;
end;
```

Figure 5: *Generated code in (a) Verilog (b) C*

executed for this rule. The condition part of a rule, $\mathcal{K}_i$, is represented as:

$$\prod_{ce(i)\neq' X';i=1}^{i=num_c} \kappa_i, \qquad \kappa_i = \begin{cases} l_{c_i} & : \quad ce(i) = `Y' \\ l_{\bar{c}_i} & : \quad ce(i) = `N' \end{cases}$$

where num_c is the number of conditions in the TDT and $ce(i)$ is the condition entry value at the ith condition row for this rule. The action part of a rule, α_i, is given as $\prod_{ae(i)\neq' 0';i=1}^{i=num_a} l_{a_i}$, where num_a is the number of actions and $ae(i)$ is the action entry value at the ith action row.

A rule is a tuple, denoted by $r_i = (\mathcal{K}_i : \alpha_i)$, which can be expressed as a product or *cube* of corresponding action and condition literals. For a given TDT, T, we define an algebraic expression, E_T, that consists of a disjunction of cubes corresponding to the rules in T. Therefore, a TDT can be represented as,

$$E_T = \sum_{i=1}^{ncolumn} r_i = \sum_{i=1}^{ncolumn} (\mathcal{K}_i : \alpha_i)$$

where $ncolumn$ is the number of columns in E_T. For simplicity, we use 'c' and 'a' instead of l_c and l_a in the following. A TDT expression is comprised of the sum of products of the literals. A cube is a conjunction of some or all of the literals l_c and l_a. A *minterm* is a cube in which every literal in the TDT expression appears. In a TDT, a cube is represented by a rule or

column comprising of the condition part and the action part in the control-flow table. A column, col_1, in a TDT is said to dominate another column, col_2, if for every row in col_1, col_2 has the same entry in that row as col_1. As a coding style, the conditionals in the dominated columns are nested inside conditionals in dominant columns.

The TDT expressions for the two TDTs in the example shown in Figure 4 are given as:
$Tdt_1 = C_1\bar{C}_2a_1 + C_1C_2\bar{C}_3a_1a_2 + C_1C_2C_3a_1a_2a_3$
$Tdt_2 = C_1a_1 + C_1C_2a_2 + C_1C_2C_3a_3$
Expression Tdt_1 uses 8 condition literals and 6 action literals, whereas expression Tdt_2 uses 6 condition literals and 3 action literals. Fewer literals coupled with nesting of condition checks leads to a shorter code sequence from Tdt_2. In general, TDTs are of two types:

- **Disjoint Rule TDT** (DR-TDT) : In a DR-TDT, the condition clauses are disjoint. Each condition clause activates only one set of actions. There exists no assignments of condition variables where two rules are activated simulataneously.

- **Minimum Condition TDT** (MC-TDT) : MC-TDT uses the minimum number of conditions to activate a rule. This corresponds to minimum number of condition literals in each rule. For instance, Tdt_2 corresponds to the minimum 2-level sum of products representation. Each action is enabled by a minimum number of conditions.

To improve quality of generated code, TDT optimizations must attempt to generate TDTs which use fewer action and condition literals and maximally order the columns according to dominance relation. The number of action literals is minimized throgh a transformation called *Action Sharing* that attempts to minimize the number of entries in the action table. This transformation is discussed in [17]. Condition literals are minimized by making each product term in the two-level algebraic expression prime. A prime cover is generated using two-level logic minimizer followed by column ordering based on dominance relation. The overall flow for software synthesis is shown in Figure 6. Code restructuring has been discussed in [18]. Action sharing cor-

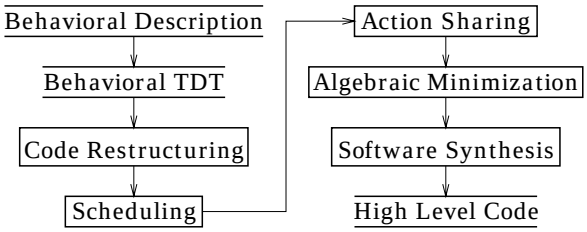

Figure 6: *Software Synthesis Methodology*

responds to the identification of identical actions in the TDT and subsequent merging of the corresponding action rows. Merging is valid only when the serialization relations among actions are maintained.

4 Implementation & Results

We have implemented the TDT based modeling system in C++ using the hardware description language (hdl) VHDL [19] as the front-end system specification language and the output language as well. The present system performs all the TDT optimizations, provides a user interface for specification of assertions for *don't care* analysis, performs TDT flattening and kernel extraction and software synthesis.

Benchmark	Conditions checked	
	DR-TDT	optimized TDT
prawn cpu	117	52
arm counter	175	47
traffic light	46	29
kalman filter	89	26

Table 1: *Num. of conditionals in generated code*

In Table 1, we compare the software generated using the disjoint TDTs versus optimized TDTs. The comparison is based on the number of conditionals checked. The benchmarks are the prawn cpu from [19], the arm counter, the traffic light controller and the Kalman filter [20]. The optimized TDTs generate 40-70 % fewer conditional checks than the DR-TDTs for all the benchmarks considered.

Work is ongoing on characterization of the effect of these optimizations on the final assembly code. Future reports will include results to assess the impact of TDT generated code on final code size.

5 Acknowledgements

The authors would like to acknowledge support from Escalade and Synopsys Inc. under the California MICRO program (97-062 and 98-055) and from NSF (award number CCR-980686898).

References

[1] P.Marwedel, G.Goossens *Code Generation for Embedded Processors*, Kluwer Academic Publishers, 1995

[2] R.Leupers, F.David, *A Uniform Optimization Technique for Offset Assignment Problems*, Intl Symp on System Synthesis,1998

[3] G.Araujo, A.Sudarsanam, S.Malik, *Instruction Set Design and Optimizations for Address Computations in DSP Architectures*, Intl Symp on System Synthesis,1996

[4] A.Sudarsanam, S.Liao, S.Devadas, *Analysis & Evaluation of Address Arithmetic Capabilities in Custom DSP Architectures*, DAC,1997

[5] S.Liao, S.Devadas, et al *Storage Assignment to Decrease Code Size*, ACM Trans. on Prog. Lang. and Systems,1996

[6] T.Okuma, et al, *Instruction Encoding Techniques for Area Miniimization of Instruction ROM*, Intl Symp on System Synthesis,1998

[7] S.Ritz, S.Pankert, H.Meyr, *High Level Software Synthesis for Signal Proc. Systems*, Intl Conf on Acoustics,Speech & Signal Proc,1990

[8] P.Murthy, S. Bhattacharyya, E.Lee, *APGAN and RPMC:Complementary Heuristics for Translating DSP Block Diagrams into Efficient Software Implementations*, Design Automation of Embedded Systems,1997

[9] D.Powell, E.Lee, W.Newman, *Direct Synthesis of Optimized DSP Assembly Code from Signal Flow Block Diagrams*, Int. Conf. on Acoustics, Speech & Signal Processing,1990

[10] P. Panda, H. Nakamura, N.D. Dutt, A. Nicolau, *Augmenting Loop Tiling with Data Alignment for Improved Cache Performance*, IEEE Transactions on Computers, to appear 1999

[11] M.Miranda, F.Catthoor et al, *High-level Address Optimisation and Synthesis Techniques for Data-Transfer Intensive Applications*, IEEE Trans. on VLSI Systems,1998

[12] C.Liem, P.Paulin, A.Jerraya, *Address Calculation for Retargetable Compilation and Exploration of Instruction-Set Architectures*, Design Automation Conference, 1996

[13] J. Li, *Timed Decision Tables and its Applications in Pre-synthesis and Partial Synthesis of Digital Circuits*, Phd Thesis, University of Illinois at Urbana-Champaign, 1998

[14] W.Sung, J.Kim, S. Ha *Memory Efficient Software Synthesis from Dataflow Graphs*, Intl Symp on System Synthesis,1998

[15] M.Chiodo, P.Giusto et al, *Synthesis of Software Programs from CFSM Specifications*, Design Automation Conference,1995

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

[17] J. Li, R.K. Gupta, *Limited Exception Modeling and Its Use in Presynthesis Optimizations*, Design Automation Conference,1997

[18] J. Li, R.K. Gupta, *HDL code restructing using Timed Descision Tables*, Intl Workshop on Hardware/Software Co-Design,1998

[19] Z. Navabi, *VHDL: Analysis and Modeling of Digital Systems*, McGraw-Hill 1993

[20] N. Dutt et al, *High Level Synthesis Repository*, ftp://ftp.ics.uci.edu/pub/hlsynth

A Implementation Details

The TDT system presented here has been developed in C++ with VHDL as the front-end language. All the previous work on optimization of TDTs including all the table operations and flattening and merging of a TDT hierarchy have been implemented. In addition, a assertion based sub-system for specifying *don't care* information has also been implemented. A TDT intermediate format has been developed in which the TDT model can be output. The system can also parse the TDT intermediate format and re-create the TDT model. A "pretty print" function has also been developed to print out the TDT model in a way which makes it more amenable to reading and understanding the model. As mentioned in the main body of the paper, the TDT model can be used to generate code in VHDL. Action Sharing has been implemented such that actions can be shared within a flattened TDT. The TDT expressions can also be extracted and then, from them the kernels can be extracted. A TDT model simulator is currently being developed.

A.1 Assertion Subsytem

The assertion subsystem allows assertions to be given on the conditions in the TDT model. For this, all the conditions in the TDTs are first assigned a unique label in the format C{TDTNum}_CondNum. For example, the second condition in TDT number ten, would have the label, C10_2. Assertions can then be specified by a command with the format,

assert {Boolean Combination of Conditions} = zero or one

For example, an assertion may look like,

assert not(C10_1) AND not(C10_2) = 0

The assertions are then used to create don't care columns in the TDT model which are used to optimize the TDTs.

A.2 Kernel Extraction

Kernel extraction starts with determining the TDT expression followed by extracting the kernels for each condition literal. The algorithm for kernel

extraction is as given in [18]. An example is given below:

```
Tdt 8 =
        c8_0 c8_1 a8_0 a8_1  +
        c8_0 c8_1' a8_2 a8_3 a8_4
Extracting Kernels of the Tdt Expressions
Extracting Kernels for Tdt num 8
Printing Kernels
Kernel K0 =
        c8_0  c8_1  a8_0  a8_1
        + c8_0  c8_1'  a8_2  a8_3  a8_4
Kernel K1 =
        c8_1  a8_0  a8_1
        + c8_1'  a8_2  a8_3  a8_4
extractedLiterals =
        c8_0
```

A.3 Obtaining the Distribution

A distribution of the *TDT Based Optimization System* (Topts) is available on the internet at *http://www.ics.uci.edu/~iesag/Topts*.

The current release of the Topts distribution is Version 0.1. It has been developed on a Sun Solaris platform using g++ as the compiler. However, except for the *bison* and *flex* files, the rest of the distribution can be compiled under the Microsoft Visual C++ environment. The bison and flex files can be compiled by installing their window's port available at *http://www.cygnus.com*. The installation can be installed by untaring and issuing a "make" command in the *src* directory. The distribution requires "g++", "bison" and "flex". The executable, *Topts*, resides in the directory *bin* after compilation.

The software is invoked by issuing the command *Topts*. It has an interactive command-line help. The following commands are supported:

- Help - Prints the help message on given command
  Syntax: 'help [command—all]'

- Quit - Quits from the FrontEnd system
  Syntax: 'quit'
- EnterAssert - Enters the external Assertions sub system
  Syntax: 'enter_assert or assert [optional_script_fileName]'
- ListAssert - Lists the external Assertions
  Syntax: 'list_assert'
- ResetAssert - Clears the entered external Assertions
  Syntax: 'reset_assert'
- Flattens the Tdt hierarchy by one level by default. Flattens completely, if the "all" directive is given
  Syntax: 'flatten [all]'
- MergeAll - Carry out all the merging transfromations
  Syntax: 'merge_all'
- Optimize_Tdt - Perform column/row reductions
  Syntax: 'op_tdt '
- Merge Actions - Extracts Action Sharing and Merges the actions in the Tdts
  Syntax: 'merge_acts'
- Finds the Same Actions - finds all the same actions in the Tdts
  Syntax: 'find_same_acts, assign_ids'
- Prints the Tdt Expressions of all the Tdts or the specified Tdt Num
  Syntax: 'print_tdtExpr [optional_tdt_num]'
- Extracts the Kernel from the Tdt Expressions of all the Tdts or the specified Tdt Num
  Syntax: 'extract_kernel, ext_kern [optional_tdt_num]'
- Print_Tdt - Prints the TDT intermediate format
  Syntax: 'printtdt'

- Pretty Print Tdt - Prints the TDT intermediate format in a readable format - although not parsable
  Syntax: 'prettyprinttdt'

- ReadVhdl - Reads Vhdl from a file
  Syntax: 'readvhdl, read, r [filename]'

- PrintVhdl - Prints Vhdl of the current Design
  Syntax: 'printvhdl'

- PrintConditions - Prints the conditions in all the Tdts of the current Design
  Syntax: 'printconds, printconditions'

- ReadSimInp - Reads Simulator Input from a file and run simulation
  Syntax: 'readsiminp, runsim [filename]'
  Note: This is not fully implemented in this release

- McToDrTdt - Converts the Minimum Condition Tdt to Disjoint Rule Tdt
  Syntax: 'mctodr'