Symbolic Evaluation of Performance Models for Tradeoff Visualization

Jeffrey Walrath and Ranga Vemuri
University of Cincinnati
Department of ECE & CS
Cincinnati, Ohio 45221-0030
ranga.vemuri@UC.EDU

Abstract

Often during the design process, it is necessary to analyze the effects of tradeoff among various performance attributes of the design. Visual representations in the form of plots, graphs, and tables are typically used. These visualizations can be generated using different applications such as MatLab®, Mathematica®, Excel®, and so forth. In order to use these tools, performance equations for the design must be rendered in an equational form with only a few variables, usually 2-3. However, performance models usually consist of a large number of attributes and evaluation procedures, typically written using the full power of a programming or hardware description language. We introduce a symbolic evaluation procedure to simplify performance models. Given partial data, such evaluation yields a residual performance model which is simpler than the original model. Symbolic evaluation can be effectively used to obtain residual performance equations in terms of the variables whose tradeoff visualization is of interest to the user.

1 Introduction

Performance models of a design are often written in a programming language, or more likely, a hardware description language (HDL) such as VHDL. A performance model consists of attributes attached to entities in the design, and attribute evaluation rules to compute values of some attributes in terms of other attributes [1]. Attribute evaluation rules are written using the constructs provided by the programming language or HDL. In this paper we will show our performance models in a simple notation that can be embedded into any high-level HDL or programming language. First, we will consider an example to motivate the rest of the paper:

Figure 1 shows a schematic representation of a design net-list which consists of several gates and nets. Figure 2 shows a set of attributes and their evaluation rules to compute the gate area and longest path time of the design where attributes are identified using the notation Object-Name/Attribute Name. Attributes can be classified into two categories: primitive and computed. Primitive attributes are those whose values must be supplied for the evaluation of the other attributes. Computed attributes are those for which explicit evaluation rules are written. These attribute evaluation rules could easily be written in a HDL such as VHDL. Conversely, these evaluation rules can be extracted from performance models written in a HDL or a performance modeling language such as PDL [2].

In our example, the active area of the design is specified as the sum of the individual gate areas and the longest path delay is specified by propagating the input net times through the gates and nets in the design to the output nets. Suppose we wish visualize how active area varies when we vary areas of g1 and g2. This would be a 3-D plot. The active area equation cannot be plotted as it is, since it has the two other unknown variables: g3 area and g4 area. In order to visualize the desired tradeoff between g1 area and g2 area to avoid violating an overall active area goal, we must instantiate the values of g3 area and g4 area and partially evaluate the active area equation with these values. Figure 3 shows the result of such partial evaluation which
is in the form of a residual equation. Figure 3 also shows a plot of the residual equation obtained using a visualization package called Gnuplot.

As another example, suppose we wish to visualize the variation in the longest path delay, denoted by n8'time, with respect to the g4'delay. To do so, the equation for n8'time must be partially evaluated with all its influencing parameters except g4'delay, instantiated to specific values. This necessitates evaluation of several other equations as well. Figure 4 shows the residual equation for n8'time following such partial evaluation and Figure 4 shows a plot of this equation.

**Symbolic evaluation** is the process of assigning values to some or all of the primitive attributes and evaluating the attribute evaluation rules to obtain a simplified performance model. In addition, symbolic evaluation also involves determining the proper order for evaluating the attribute evaluation rules. For example, the rule for calculating n8'time cannot be evaluated until the rule for n8'time is evaluated. To ensure a proper order can be determined, a check must be done for cyclic dependencies among the equations. A consistent evaluation order can not be determined in the presence of cyclic dependencies among the attributes.

In this paper we introduce and use a notation for writing performance models. This notation is suggestive enough to realize that it could be implemented in a HDL or a programming language. This notation is quite expressive and allows for declaration of a variety of evaluation rules. We also describe procedures for partial symbolic evaluation of each expression type. Full evaluation is simply a special case when all the primitive attribute data is available.

## 2 Performance Models

Conceptually, a performance model is created by augmenting a design net-list with a collection of attributes and evaluation rules [3]. Attributes are values which represent various non-functional design parameters such as heat dissipation, power consumption, input voltage, and clock rate.

Although a performance model can be created by augmenting the design by hand, an alternative approach is shown in Figure 5. A generic model is a series of declarations of module, carrier, and port objects. These objects are the types of the various components that may appear in a design. Within each object are declarations for attributes and evaluation rules which define how to calculate values for the attributes. This generic model is compiled with a specific design to produce a performance model of the design.

Compiling a model involves taking each component in the design, finding the corresponding declaration in the generic model, and assigning all the attributes and evaluation rules. The result is a performance model for the specific design. It is with this performance model that symbolic evaluation can be performed and used for tradeoff visualization.

Attributes are classified as either **primitive** or **computed**. Primitive attributes are those which are given a value by the user and are not calculated. Conversely, computed attributes are defined by an expression and can involve other attributes in the design. Every computed attribute will have an evaluation rule (ie. an expression) defining how to compute a value for the attribute.

In addition, each attribute has a type associated with it and possible base types include integer, real, boolean, character, string, or enumerated type. Additionally, attributes can be created as aggregate data types such as records, lists, and arrays. A record is defined as a collection of data fields with each field having its own specified type. Elements of the record are uniquely referenced by their field name. A list attribute is an unbounded collection of values of the same type. For example, an attribute could be a list of integers or a list of boolean values. Each element in the list is referenced by its unique integer location in the list with the first element being location one. An array attribute is just a bounded list.

Since attributes are attached to design components (modules, nets, ports) of the net-list, each component is assumed to have a unique name. In case of hierarchical net-lists these names can be automatically formed. Thus, the combination of an element name and attribute name refer to a unique instance of an attribute within the performance model.

There are several different types of expressions that can be used for creating attribute evaluation rules. An evaluation rule assigns a value to an attribute. The value to be
assigned is specified by an expression on the right-hand side of the rule. Any expression may involve attributes of other components in the design.

The various expression types include mathematical expressions, if-then-else expressions, case-switch expressions, two different for-each expressions, begin-end blocks, and function calls. Functions are written in a pascal-like manner and are side effect free. Function declarations can include such things as nested variable declarations, nested loops, and return statements. For information regarding the particular syntax please refer to the article by Venuri et al[2].

Example Performance Model

This example shows a performance model to determine power consumption for the combinational circuit shown in Figure 6. One method for estimating power consumption of a combinational circuit in CMOS technology is based upon switching probabilities [4]. Each primary input is assigned an input switching probability. This is the probability that the input will change from its previous value on the next application of an input vector. For a circuit with random input, there is 50% chance that a particular input will change. If the circuit has a reconvergent fanout, then the switching probabilities for all other gates can also be calculated. Knowing the frequency at which the primary inputs change, the switching frequency for each gate in the circuit can be calculated. With the switching frequency, the standard CMOS power equation 

$$
\text{power} = 0.5 \times \text{voltage}^2 \times \text{frequency} \times \text{capacitance}
$$

can be used to estimate the power consumed by a gate. The total power consumed by the circuit is the sum of the powers consumed by each gate.

Figure 7 shows the performance model for estimating the power consumption for the net-list shown in Figure 6. The sum_real and min_real functions were eliminated for space. Sum_real summed a list of reals and min_real found the minimum value in a list of reals.

3 Symbolic Evaluation of Models

There are several operations that must occur during the symbolic evaluation of a performance model. The following sections briefly describe the various steps for symbolic evaluation.

Cyclicity Checking: Before the expressions in the performance model can be evaluated, a check for cyclic dependencies must be done. Since an expression can refer to any attribute in the performance model, it is possible to write a series of expressions which incorporate a cycle of dependencies among the expressions. Figure 8 is a simple example of a cyclic dependency among three evaluation rules. When a cyclic dependency exists, it is not possible to determine the correct order for evaluating the expressions. Dependency analysis is performed on the set of expressions to ensure that no cyclic dependencies exist.

Evaluation Order: Once a performance model has been checked and there are no cyclic dependencies, the next step is to evaluate the expressions in the model. Since expressions can rely on other attributes within the model,
there is an associated order in which the various evaluation rules must be evaluated. For example, if there were two attributes x and y with evaluation rules x = y + z and y = 4/2 + 1, the expression for x can not be evaluated until the expression for y has been. This concept is called an evaluation order. Before any evaluation can begin, the evaluation order for all expressions within the performance model must be determined.

The process for determining the evaluation order of the attributes begins by making a dependency tree for each attribute. Each node in a dependency tree represents an attribute in the performance model. A directed edge between two nodes represents a dependency. In the previous example, the attribute x depends on the attributes y and z. Thus, there would be two directed edges from x to y and z in the dependency tree. Leaves of the dependency tree are primitive attributes because a primitive attribute has no dependence on any other attributes in the model. A performance model may contain more than one dependency tree, and the collection of trees together make a dependency graph for a performance model.

An evaluation order for all the attributes is determined based on the dependency graph. Leaf nodes in each dependency tree are given the lowest order of one. Then correspondingly, each node is assigned an evaluation based on the evaluation order of its children. A node’s evaluation order is calculated by adding one to the largest evaluation order of all its children. Figure 9 shows a dependency tree and the respective evaluation order for all the nodes. All trees in the dependency graph are sorted in this manner to determine the evaluation order for every attribute. Because the performance model has been checked for cyclic dependencies, there will be no cycles. (In fact, cyclicity checking and evaluation order determination can be done together.)

Symbolic Evaluation: Having determined an evaluation order, symbolic evaluation of the performance model can be done with partial, full or no primitive attribute data. Some primitive attributes can be left unspecified. Partial symbolic evaluation is the process where expressions are symbolically evaluated even when there are attributes which are unknown. In this case, expressions are reduced but not completely eliminated because they rely on some attribute that cannot be evaluated. For example, if \( x = y + z + 3 \) and \( z = 3 \), then the expression could be partially evaluated to \( x = y + 6 \).

An attribute is considered known if its value reduces to a specific value and doesn’t remain as a residual expression; otherwise an attribute is considered unknown [5, 6]. For example, an attribute which is an integer is known if it has an integer value otherwise it unknown. Because there are several different types of expressions possible in a performance model, symbolic evaluation for each expression is slightly different.

The algorithm for evaluation is denoted by a function called \( \text{SymEval}() \). Input to the function is the expression to evaluate, and it returns a value of either known or unknown based upon the result of evaluation. \( \text{SymEval}() \) is a recursive function. During the evaluation process, the \( \text{SymEval}() \) function replaces anything in an expression that is known. The following sections define methods for evaluating the various expressions and statements.

Mathematical Expressions: Every mathematical expression is parsed into an expression tree in the correct order of precedence. Each node in the expression tree is an operator with one or two children expressions depending on whether the operator is unary or binary. First, all the children expressions are evaluated. When the operator is unary and \( V_1 \) is known, the math expression can be replaced by the resulting unary operation. When the operator is binary and both \( V_1 \) and \( V_2 \) are known, the binary operation is performed and the expression is replaced by the result.

If-Then-Else Expressions: As shown below, each part of the if-then-else expression is evaluated first. When \( V_2 = \text{known} \), the true-expr is just a single real value. When \( V_2 = \text{unknown} \), true-expr is a residual expression that has been reduced as much as possible. The same holds for \( V_3 \) and false-expr. If \( V_1 = \text{known} \) and the conditional-expr is true, the if-then-else expression is replaced by the evaluated results of true-expr. Correspondingly, if \( V_1 = \text{known} \) and the conditional-expr is false, the if-then-else is replaced with the evaluated result of false-expr. When \( V_1 = \text{unknown} \), nothing is replaced.

Case Expressions: All expressions within the case expression are evaluated. When \( V_1 = \text{unknown} \), the case expression can not be replaced. The only thing to occur is the reduction of each expression composing the case expression. However, when \( V_1 = \text{known} \), several possibilities arise. If a match-expr evaluates to known and is equal to the switch-expr value, the corresponding residual arm-expr can replace the case expression if and only if all match-expr expressions up to that arm have evaluated to known. For example, if \( V_6 = \text{known} \) and is equal to the switch, the case expression can only be replaced if \( V_2 \) and \( V_4 \) are also known. When this is not true, then the case expression can not be replaced. The default expression can replace the case expression only when all match-expr expressions are known and none match the value of the switch-expr.

Figure 9: Example of evaluation order for attributes
Foreach Expression: Each one of the three foreach expressions is evaluated differently. The first type is the foreach expression that iterates over ranges. If \( V_1, V_2, \) or \( V_3 \) is equal to \textit{unknown}, no replacement of the foreach expression is possible. However, when all three are \textit{known}, the foreach expression can be unrolled. The loop variable \( \text{var} \) is set equal to the \textit{left-expr} and the \textit{body-expr} is evaluated. During the evaluation of the \textit{body-expr}, any reference to the loop variable is replaced with the current value. Once the body expression is evaluated, the residual expression is stored in the next location in the list that is being generated by the foreach expression. This continues until the loop terminates. Once the list has been created, the foreach expression is replaced by the list.

\[
\text{attr} = \text{foreach} \ \text{var} \ \text{in} \ (V_1 = \text{SymEval(left-expr)}) \ \text{to} \ (V_2 = \text{SymEval(right-expr)}) \ \text{by} \ (V_3 = \text{SymEval(body-expr)}))
\]

The other type of foreach expression iterates over a set/list of data. When the set/list itself has a \textit{known} status (the status of the elements can be \textit{unknown}), the foreach expression can be unrolled. The procedure is to set the loop variable equal to one of the items in the set/list. Then the body expression is evaluated and any reference to the loop variable is replaced by the current value. The residual expression is stored in the list being created by the foreach expression. The loop variable is set equal to the next item in the set/list and the process repeats. This continues until all items in the set/list have been iterated. The foreach expression is then replaced by the resulting list.

\[
\text{attr} = \text{foreach} \ \text{var} \ \text{in} \ \{\text{obj}, \text{obj}_2, ...\} \ (V_1 = \text{SymEval(body-expr)}))
\]

Begin-End Expression: The process for evaluating the begin-end expression begins by setting a temporary fail flag to false. Each variable declaration statement is evaluated along with the initial value if there one. If any of the variable declaration statements evaluated to \textit{unknown}, the fail flag is set to true. Then each programming statement in the begin-end expression is evaluated. If during the evaluation of a statement, the result is \textit{unknown}, the fail flag is set to true. When a return statement is reached, several conditions are checked. First, the return expression is evaluated. If that value is \textit{known} and the fail flag is still false, then the entire begin-end expression can be replaced by the residual return expression. However, if the fail flag is true, that means a previous statement did not completely evaluate so the begin-end expression can not be replaced.

Function Call: Function calls are the most complicated expression to evaluate. First, a copy of the function body (which is a begin-end expression) is made. Then variable declarations are added to the top of the copied function body. For each argument in the function argument list, a declaration is made for that variable and the initial value is set to the value being passed to the function. Once the function call is replaced, the begin-end expression is evaluated. The following example illustrates this process:

\[
\begin{align*}
\text{attr} &= \text{min_val}(\text{obj}_1' \text{val}, \text{obj}_2' \text{val}) \\
\text{function min_val}(a, b) \ &\begin{align*}
\text{a} &= \text{obj}_1' \text{val} \\
\text{b} &= \text{obj}_2' \text{val} \\
\text{end} \end{align*}
\end{align*}
\]

\[
\begin{align*}
\text{sc1/gate\_type} &= \text{and\_gate} \\
\text{sc1/probs} &= [\text{in1'\_signal\_prob, in2'\_signal\_prob}] \\
\text{sc1/signal\_prob} &= \begin{align*}
\text{begin} \ &\begin{align*}
\text{temp} &:= 1.0 \\
\text{temp} &= \text{in1'\_signal\_prob} \ast \text{in2'\_signal\_prob} \\
\text{return} \ &\text{temp}
\end{align*}
\end{align*}
\end{align*}
\]

Figure 10: Initial symbolic evaluation

4 Performance Visualization Using Symbolic Evaluation

Using the performance model shown in figure 7, the following examples illustrate how partial evaluation can be used to simplify the performance model and generate equations for visualization.

One of the primitive attributes for this performance model is \textit{gate\_type}. Each component in the net-list is either an and, or, or net gate. Each \textit{gate\_type} attribute is set according to the net-list shown in Figure 6. First, symbolic evaluation is performed with only these and no other primitive attribute values set. This results in a residual model. In this residual model, all function calls are replaced with the appropriate begin-end expressions. In addition, all set foreach expressions are unrolled. Since the attribute \textit{gate\_type} is known, several statements inside the function \text{prob\_in} are also reduced. Figure 10 shows the residual evaluation rules for just component \text{sc1}; the other two components have similar reductions and results.

Starting with this residual model, a series of partial evaluations were performed, each one with a different set of primitive attributes defined. In all the evaluations, the input probabilities were set to 0.5. For the first evaluation, all the capacitances were defined with voltage and system_freq left undefined. Figure 11 shows the resulting set of equations produced by partial evaluation with this setting for the primitive attributes. The plot in Figure 11 shows a visualization, obtained using Gnuplot, of the residual model. Such visualization, of course, couldn’t be obtained with the original model. And, full evaluation will yield one specific point in this plot. Without using partial evaluation, the only other way to obtain such a plot is possibly using many rounds of full evaluation for many sets of full primitive attribute data. Such an approach will not only be expensive in time, but also could completely miss “interesting” trade-offs points in the plot where sudden crests and troughs may occur.

To illustrate the flexibility of the partial evaluation based approach we will show to two other partial evaluations of this example.

Partial evaluation was done with values defined for sc2'-capacitance, sc3'capacitance, and mc\_voltage. Figure 12 shows the resulting equations generated by that evaluation. The plot in Figure 12 shows a visualization of these equations.
The partial evaluation approach for performance visualization developed in this paper is flexible and nearly as efficient as full evaluation. Tradeoff visualization can often be achieved with only one round of partial evaluation. The partial evaluation approach discussed in this paper has been successfully embedded in the PDL (Performance Description Language) and VHDL based approach to performance modeling and analysis described in [2].

For the power example shown in the previous section on a design consisting of approximately 25,000 components, compiling a performance model took 1.5 minutes with evaluation of the model having one unknown attribute taking roughly 16 minutes. A successive evaluation of the residual model with a value for the unknown attribute took slightly less than 4 minutes. These times were measured on a sparc-4 with 128MB of memory.

REFERENCES


