Polynomial Methods for Component Matching and Verification

James Smith
Stanford University
Computer Systems Laboratory
Stanford, CA 94305

Giovanni De Micheli
Stanford University
Computer Systems Laboratory
Stanford, CA 94305

1. Abstract

Component reuse requires designers to determine whether or not an existing component implements desired functionality. If a common structure is used to represent components that are described at multiple levels of abstraction, comparisons between circuit specifications and a library of potential implementations can be performed quickly. A mechanism is presented for compactly specifying circuit functionality as polynomials at the word level. Polynomials can be used to represent circuits that are described at the bit level or arithmetically. Furthermore, in representing components as polynomials, differences in precision between potential implementations can be detected and quantified.

2. Introduction

The increased complexity of integrated circuits has forced designers to reuse existing circuitry when constructing new systems. The proliferation of reusable blocks has promised opportunities to complete new designs more quickly and with fewer errors. However, searching the space of existing implementations is time consuming and fraught with pitfalls, as the suitability of existing blocks is determined by manual methods or verbal descriptions. This search promises to become more complex as the number and need for reusable designs increases. The structures and methods presented in this paper enable automation of this search.

Component matching is the problem of allocating complex blocks given a system specification. This problem reduces to determining if an element in a library of existing designs performs the same function as part of the specification. For example, in designing the baseline JPEG encode block of Figure 1, subblocks are required to perform a discrete cosine transform (DCT), quantization, DC (zero frequency) encoding, and AC (non-zero frequency) encoding. This system can be synthesized by matching the arithmetic specification of each of these functions to a bit level description of the implementation.

Component matching is closely related to verifying that a specification and implementation match exactly, but presents a few important differences. Several blocks may be able to satisfy the functionality of a specification, but contain very different implementations, allowing for tradeoffs in execution time, area, power consumption, precision and other qualities. For example, in performing DCT operations, an implementation may compute the cos(x) to a very high degree of accuracy, yielding smooth object edges, but incurring a high cost in area and execution time, while another implementation may employ the same technique at a lower level of precision to preserve area and increase performance. Both implementations match the specification's functionality, yet yield different numerical results.

The examples discussed above can be specified very efficiently with polynomial models. For example, cos(x) can be approximated by:

$$1 - x^2/2! + x^4/4! - x^6/6! + \ldots + x^n/n!$$

This paper presents methods for developing analogous word level polynomial models for existing implementations given a bit level description of the implementation. These methods are ideally suited for circuits that implement arithmetic functions and, in this paper, are applied to combinational circuits.

In constructing polynomial models, we derive a means for determining whether a specification and implementation perform the same function and a means for quantifying their differences. Quantifying the differences between these two models allows functionality to be approximated within specified bounds to achieve higher performance or lower area and power costs. Furthermore, quantifiable differences can be compensated for by adding logic around an existing block.

3. Related Work

Component matching has historically been restricted to allocating combinational logic gates. Many structures,

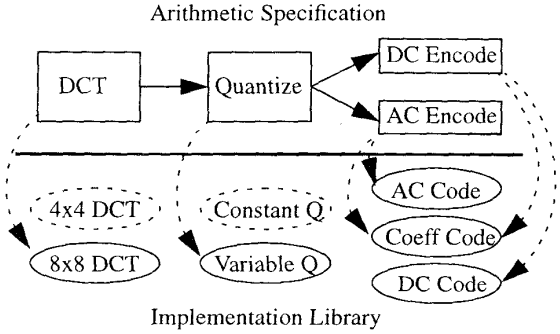

Fig. 1 Mapping JPEG encode onto existing designs

ICCAD98, San Jose, CA, USA

such as ROBDDs ([Br86]), are ideal for mapping HDL specifications onto a library of gates. The canonicity and ease of composition that ROBDDs provide make them ideal for matching small combinational circuits. However, for more complex functions, the potentially exponential size of BDD structures makes comparison of BDDs time consuming and memory intensive. When comparisons are sought between functions that are not described at the bit level, BDD structures are not sufficient to represent circuit functionality. Furthermore, BDDs can yield information on whether or not a specification and implementation match exactly, but offer no path for quantifying the degree to which the two are the same. That is, two functions that have similar, but not equal, BDD structures may implement drastically different functions while two very different BDDs may implement the same function with different degrees of precision.

Binary Moment Diagrams (BMDs) ([BrCh95]) have been developed to ease the memory and time required to manipulate complex structures by generating word level representations. BMDs have been used to verify the functionality of linear circuits [ChBr96] and could be adapted to perform component matching for those circuits. However, BMDs are unsuitable for use in non linear functions because of the resulting exponential complexity. Hybrid Decision Diagrams ([ClFu95]) and Multi Terminal BDDs ([ClFu93]) suffer from similar restrictions. PHDDs, developed in [ChBr97] are well suited to handling the non linearities associated with floating point arithmetic, but can still be exponentially complex for non-linear functions.

Minato introduced a method for modeling and manipulating circuits that implement polynomial functions using Zero-suppressed BDDs ([Mi96]). This structure provides an efficient representation for those circuits for which a polynomial description is specified, but becomes exponentially large if discontinuities exist in the function. The methods we will present here develop a mechanism for deriving the polynomial representation given a Boolean circuit description. In addition we will present a mechanism for manipulating and modeling circuits that contain discontinuites and for detecting these discontinuites.

Efficient component matching requires data structures that are canonical, constructible in polynomial time, and allow for simple composition. This paper will demonstrate methods for determining polynomial representations for circuits that are described at the bit level. Furthermore, we will prove that a unique minimum order polynomial representation exists for all circuitry without feedback. In representing hardware as polynomials, blocks can be efficiently compared with one another to determine if they implement the same functionality. In addition, polynomials are easily composable, allowing efficient determination of the functionality of hierarchical or partitioned blocks.

4. Bit Level Polynomial Representations

Generating a word level polynomial representation for a Boolean function may appear to be an inconsistent problem because Boolean functions are inherently discontinuous. However, a Boolean function, $y = F(x)$, where x and y are bit vectors, can be treated as a collection of coordinates (x, y) which can be fit to a minimum-order polynomial. If the order of this polynomial is known to be n, then n+1 coordinates can be extracted from the function and a set of n+1 equations and variables (the coefficients of the polynomial) can be constructed and solved. Thus, the problem of generating a word level polynomial representation for a Boolean function reduces to determining the order of the polynomial.

4.1 Existence and Uniqueness

The following theorem is the basis for determining the polynomial representation of circuits described at the bit level. This theorem, derived from the binomial distribution from traditional calculus, is proven for integers and adapted to prove the existence and uniqueness of polynomial representations of Boolean functions.

Theorem 4.1 Given a polynomial function F(x) of order n, where $x \in Z$, the function $\hat{F}(x) = F(x+1) - F(x)$ is of order exactly n-1.

Proof Let $F(x) = \sum_{i=0}^{n} c_i \cdot x^i$

$$\hat{F}(x) = \sum_{i=0}^{n} c_i \cdot (x+1)^i - c_i \cdot x^i$$

$$c_i (x+1)^i - c_i x^i = c_i \cdot \left(\sum_{j=0}^{i} \binom{i}{j} \cdot x^j - x^i \right) = c_i \left(\sum_{j=0}^{i-1} \binom{i}{j} x^j \right)$$

thus, each term of order i in F(x) contributes a polynomial of order exactly i-1 to $\hat{F}(x)$. Thus, x^n contributes a polynomial of order n-1 and is the only term to do so. Therefore $\hat{F}(x)$ is of order n-1 ❑.

To illustrate Theorem 4.1, note that if $F(x) = x^3$, then $F(x+1) - F(x) = x^3 + 3x^2 + 3x + 1 - x^3 = 3x^2 + 3x + 1$. From Theorem 3.1, a useful corollary can be derived.

Corollary 4.1.1 For all $x, m \in Z$, the set of row vectors is linearly independent:

$[(x)^m, (x)^{m-1}, \ldots, 1]$

$[(x+1)^m, (x+1)^{m-1}, \ldots, 1]$

...

$[(x+m)^m, (x+m)^{m-1}, \ldots, 1]$

Proof From Theorem 4.1, by recursively subtracting row i-1 from row i, we reduce the order of each entry in row i by 1 each iteration. If this is performed i times for each row i, then each entry in row i with an original entry of order less than i will be 0 (i.e. order is reduced to zero after i iterations, then a constant is subtracted from itself) and each row with an original entry of order greater or equal to i will be nonzero. Thus, the following set of row vectors results:

$[(x)^m, (x)^{m-1}, \ldots, 1]$

$[(x+1)^m - (x)^m, (x+1)^{m-1} - (x)^{m-1}, \ldots, 0]$

...

$$[(x+m)^m - \left(\binom{m}{1} \cdot (x+m-1)^m \right) + \ldots + (-1)^m, 0, \ldots, 0]$$

which are linearly independent. Therefore the original set of

vectors is linearly independent ❑.

To illustrate Corollary 4.1.1, notice that, for x = 0 and m = 3:

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 8 & 4 & 2 & 1 \\ 27 & 9 & 3 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 7 & 3 & 1 & 0 \\ 19 & 5 & 1 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 6 & 2 & 0 & 0 \\ 12 & 2 & 0 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 6 & 2 & 0 & 0 \\ 6 & 0 & 0 & 0 \end{bmatrix}$$

Thus, the initial set of vectors is linearly independent.

The following theorems establish the existence of polynomial representations for combinational univariate functions and the uniqueness of the minimum order polynomial representation.

Theorem 4.2 (*Existence*) Let $\vec{x} \in B^n$, $\vec{y} \in B^k$, and $x, y \in Z$ be the integers corresponding to $\vec{x}, \vec{y}$. Given a Boolean function $F: \vec{x} \rightarrow \vec{y}$, F can be represented by a polynomial $y = a_m x^m + a_{m-1} x^{m-1} + ... + a_0$ where $m < 2^n$.

Proof If $\vec{x} \in B^n$, then there are 2^n possible values that x can take on {0, 1, ..., 2^n-1} and 2^n corresponding values that y can take on {F(0), F(1), ..., F(2^n-1)}. The solution to the set of linear equations ($m = 2^n$-1):

$$F(m) = c_m m^m + c_{m-1} m^{m-1} + ... + c_0$$
$$F(m-1) = c_m (m-1) + c_{m-1} (m-1)^{m-1} + ... + c_0$$
...
$$F(0) = c_m 0^m + c_{m-1} 0^{m-1} + ... + c_0$$

exists if no row $[i^m, i^{m-1}, ..., 1]$ is a linear combination of the others. We know this is true from Corollary 4.1.1. Note that the dimension of y does not affect the polynomial representation of F ❑.

Theorem 4.3 (*Uniqueness*) The minimum order polynomial representation of a Boolean function $F: \vec{x} \rightarrow \vec{y}$, where $\vec{x} \in B^n$, $\vec{y} \in B^k$, is unique.

Proof Assume there exist two minimum order polynomial representations for F, where $x, y \in Z$ are the integers corresponding to $\vec{x}, \vec{y}$:

$$y = a_m x^m + a_{m-1} x^{m-1} + ... + a_0$$
$$y = b_m x^m + b_{m-1} x^{m-1} + ... + b_0$$

=> there are two possible solutions to the set of linear equations:

$$F(m) = c_m m^m + c_{m-1} m^{m-1} + ... + c_0$$
$$F(m-1) = c_m (m-1)^m + c_{m-1} (m-1)^{m-1} + ... + c_0$$
...
$$F(0) = c_m 0^m + c_{m-1} 0^{m-1} + ... + c_0$$

=> there exists a row $[F(i), i^m, i^{m-1}, ..., 1]$ that is a linear combination of the others. But the subrow $[i^m, i^{m-1}, ..., 1]$ is linearly independent of all other such subrows (from Corollary 4.1.1) which means that the full row is linearly independent =><=. Therefore, the minimum order polynomial is unique ❑.

An example of the application of Theorems 4.2 and 4.3 is the following set of Boolean equations (input width n = 2 and output width k = 5) that model an existing circuit:

$$F_0(x) = x_0$$
$$F_1(x) = x_1 \cdot x_0$$
$$F_2(x) = 0$$
$$F_3(x) = x_1$$
$$F_4(x) = x_1 \cdot x_0$$

$y = x^3$ is the unique, minimum order polynomial (m = 3) that represents this circuit, and would match a specification that requires the computation of the third power of x.

4.2 Polynomial Computation

In the previous section, we have proven that any combinational circuit can be uniquely represented by a minimum order polynomial. Once the order of this polynomial is determined, then the coefficients of the polynomial can be calculated by examining a finite number the circuit outputs. Thus, the problem of determining a canonical polynomial representation for a circuit can be reduced to finding the order of the polynomial that represents that circuit.

To begin deriving a method for determining the order of a Boolean function, remember from Theorem 4.1, that the order of a polynomial F(x) will be reduced by exactly one by computing $\hat{F}(x) = F(x+1) - F(x)$. Furthermore, from Theorem 4.2, a polynomial representation exists for a Boolean function F(x), where $x \in B^n$. Therefore, the order of F(x) can be determined exactly by recursively performing $\hat{F}(x) = F(x+1) - F(x)$ until $\hat{F}(x)$ is identically zero for all values of x. In the algorithm discussed here, two's complement arithmetic is employed to compute this difference. The number of iterations required to set $\hat{F}(x) = 0$ is the order of the unique, minimum-order polynomial that represents the circuit.

In computing the order of a Boolean function, we assume that each bit of the function is represented as an Ordered Binary Decision Diagram. While this does present an exponentially sized data structure for some functions, we will show a heuristic in Section 7 that reduces this data structure to linear complexity with respect to the number of input bits. In the succeeding sections, we derived the steps required to reduce the order of F(x) by one.

4.2.1 Determining F(x+1)

The first step in computing $\hat{F}(x) = F(x+1) - F(x)$ is to determine F(x+1). This can be performed in polynomial time by replacing each bit x_n of x with $(x_n \oplus x_{n-1} \cdot x_{n-2} \cdot ... \cdot x_0)$ in the OBDD of F(x).

4.2.2 Determining -F(x)

The next step in computing $\hat{F}(x)$ is determining -F(x). Using two's complement arithmetic, this could be performed by inverting each bit $F_i(x)$ of F(x) and adding one (-F(x) = F'(x) + 1). Inversion of $F_i(x)$ is simple as it only requires inverting each leaf of the corresponding OBDD. However, if we make the assumptions that F(x) is an n bit function, x is an n bit word, and the BDD of $F_i(x)$ has at least n nodes, computing F'(x) + 1 is at least of complexity $O(n^4)$.

To reduce the complexity of the negation, the increment of the bitwise inversion of F(x) does not have to be calculated. This results in the following computation, a slightly altered version of $\hat{F}(x)$:

$$\tilde{F}(x) = F(x+1) + F'(x) = F(x+1) - F(x) - 1$$

Note that on successive computations of $\tilde{F}(x)$, denoted $\tilde{F}^i$, the subtraction of one does not accumulate:

$$\tilde{F}^2(x) = \tilde{F}(x+1) - \tilde{F}(x) - 1$$
$$= (F(x+2) - F(x+1) - 1) - (F(x+1) - F(x) - 1) - 1$$
$$= (F(x+2) - F(x+1)) - (F(x+1) - F(x)) - 1$$

Thus, instead of $\tilde{F}(x) = F(x+1) - F(x)$ being used to successively reduce the order of F(x) by one, $\tilde{F}(x) = F(x+1) - F(x) - 1$ is a less complex way to reduce the order of F(x) by one.

4.2.3 Performing F(x+1) - F(x)

Once F(x+1) and F'(x) have been determined, the two functions must be summed to produce $\tilde{F}(x)$. If performed in ripple carry fashion, this is an exponentially expensive operation with respect to word length due to the propagation of the carry (for the ith bit, the carry computation requires 3^i logic operations). To eliminate the computation of the carry, a carry save addition can be performed (bitwise):

$$\tilde{F}_{sum}(x) = F(x+1) \oplus F'(x)$$

$$\tilde{F}_{carry}(x) = F(x+1) \cdot F'(x)$$

However, this yields an $\tilde{F}(x)$ that is uniquely specified by $\tilde{F}_{sum}(x)$ and $\tilde{F}_{carry}(x)$:

$$\tilde{F}(x) = \tilde{F}_{sum}(x) + (\tilde{F}_{carry}(x) << 1)$$

Note that there are now two terms that must be negated when computing $\tilde{F}^2(x)$: $\tilde{F}_{sum}(x)$ and $\tilde{F}_{carry}(x)<<1$. Negating both terms requires a bitwise inversion and an increment of each term. As in Section 4.2.2, in order to avoid these increments, 2 must be subtracted from the summation of $\tilde{F}(x+1)$ and $\tilde{F}'(x)$. This results in the following computation, a slightly altered version of $\tilde{F}(x)$:

$$\tilde{F}(x) = F_{sum}(x+1) + (F_{carry}(x+1) << 1) + F'_{sum}(x) + (F'_{carry}(x) << 1)$$
$$= F_{sum}(x+1) + (F_{carry}(x+1) << 1) - (F_{sum}(x) + (F_{carry}(x) << 1)) - 2$$

Since F(x+1) and F'(x) are specified as the summation of a sum and carry term, their summation must be performed in two steps, as if 2 carry save adder stages (Figure 2) were combined.

In computing $\bar{F}(x) = F(x+1) - F(x) - 2$, the order of F(x) is successively being reduced by one using a computation that is of polynomial complexity with respect to the length of the input word.

4.2.4 Checking if F(x+1) - F(x) = 0

When $\bar{F}(x) = -2$, we know F(x+1) - F(x) = 0 and the order computation is complete. In order to efficiently determine if $\bar{F}(x)$ is -2, a two stage carry save increment is performed at the end of each recursive step, allowing $\bar{F}(x)+1=-1$ to be a sufficient condition for completing order computation. Each bit of the resulting sum (S_{test}) is checked for tautology and each bit of the resulting carry (C_{test}) is checked whether it is tautologically zero. We refer to this test as the *tautology check*, and it is sufficient to guarantee $\bar{F}(x) = -2$ as explained in Theorem 4.4. As a result the ripple carry computation does not need to be performed.

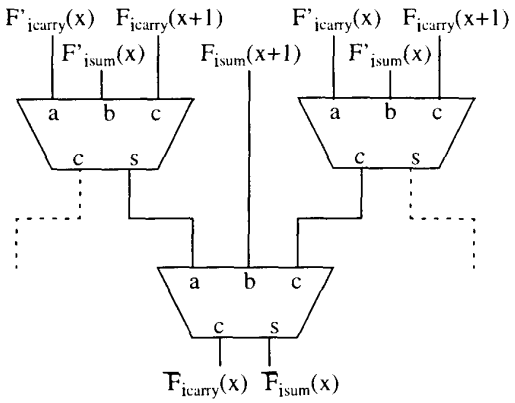

Fig. 2 Two stage carry save adder for computation of $\bar{F}(x)$

Theorem 4.4 Given three Boolean vectors s, c, $f \in B^n$, where $f = s + (c<<1)$, then $f = -1$ iff $s_i \oplus c_{i-1} = 1$ and $s_i \cdot c_{i-1} = 0$ for all i.

Proof Forward implication (by induction):

Base case: since $s + (c<<1) = -1 \Rightarrow s_0 = 1$

$$c_{-1} = 0 \Rightarrow s_0 \oplus c_{-1} = 1 \text{ and } s_0 \cdot c_{-1} = 0$$

Assume: $s_j \oplus c_{j-1} = 1$ and $s_j \cdot c_{j-1} = 0$ for all $j \le i$

Inductive step: $f_{i+1} = 1$ and $s_j \cdot c_{j-1} = 0$ for all $j \le i$

$$\Rightarrow s_{i+1} \oplus c_i = 1.$$

Reverse implication:

$f_i = 1$ since $s_i \oplus c_{i-1} = 1$ and $s_i \cdot c_{i-1} = 0$ for all i.

$\Rightarrow f = -1 \Rightarrow s + (c<<1) = -1$ ❑.

At this point, we have developed an algorithm, of polynomial complexity, for iteratively reducing the order of a Boolean function by one and determining if the result of each iteration produces a polynomial of order zero (Figure 3).

4.2.5 Bounding Function

A continuous function F:x->y, where $x, y \in Z$, has a corresponding Boolean function $F: \vec{x} \rightarrow \vec{y}$, where $\vec{x} \in B^n$ and $\vec{y} \in B^k$, defined only over the domain $[0, 2^n-1]$. This is important to consider when performing order computations because $F(\vec{x}+1) - F(\vec{x})$ actually corresponds to $F(0) - F(2^n-1)$ if $x = 2^n-1$. In performing order computations, this may result in f(x) appearing to be discontinuous over the domain $[-\infty,\infty]$ even if it is continuous over the range of possible values for x. Thus, in executing order computations it is necessary to determine a bounding function that specifies

which values do not need to be considered when performing tautology checks.

Definition 3.1 Given a function F(x), where x is an n bit word, the bounding function B(x) on the mth order iteration is:

$$B(x) = \sum_{i=2^n-m}^{2^n-1} \left(\prod_{j=0}^{n-1} (x_j = (i>>j) \bmod 2) \right)$$

In words, this is the sum of the Boolean vectors whose corresponding integer values are greater than 2^n-m. For example, after one iteration in determining the order of a function with respect to an n bit variable x, the bounding function is $B = x_{n-1} \cdot x_{n-2} \cdot \ldots \cdot x_0$. After two iterations, the bounding function would be $B = x_{n-1} \cdot x_{n-2} \cdot \ldots \cdot x_0 + x_{n-1} \cdot x_{n-2} \cdot \ldots \cdot x_0'$.

If a function is out of range when incremented, i.e. $x+1 = 2^n$, then the resulting $\bar{F}(x)$ is immaterial, since the input pattern can not be applied. Thus, $\bar{F}(x)+1 = -1$ requires that if S_{test} is not a tautology, the bounding function must be true. Similarly, if C_{test} is not tautologically zero, the bounding function must be true if $\bar{F}(x)+1 = -1$. The tautology check is therefore performed on each bit of

$$(S_{test} + B)(C_{test}' + B).$$

For example, if, after two order computations, bit k of S_{test} is $(x_{n-1} \cdot x_{n-2} \cdot \ldots \cdot x_0)'$, then $S_{test} + B = (x_{n-1} \cdot x_{n-2} \cdot \ldots \cdot x_0)' + x_{n-1} \cdot x_{n-2} \cdot \ldots \cdot x_0 + x_{n-1} \cdot x_{n-2} \cdot \ldots \cdot x_0' = 1$ and the bit satisfies the tautology check. Thus, within the interval x = [0, 2^n-1], S_{test} is a tautology. The complete algorithm for determining the order of a Boolean function is shown in Figure 3. An example is completed in Appendix A.1.

Once the order of the function has been determined to be m, F(x) is evaluated at x = 0, x = 1, ... , x = m. Solving the following set of linear equations for $c_0, c_1, \ldots, c_m$ yields the polynomial representation of the Boolean function:

$$F(m) = c_m m^m + c_{m-1} m^{m-1} + \ldots + c_0$$
$$F(m-1) = c_m (m-1)^m + c_{m-1} (m-1)^{m-1} + \ldots + c_0$$
$$\ldots$$
$$F(0) = c_m 0^m + c_{m-1} 0^{m-1} + \ldots + c_0$$

4.3 Extension to Multivariate Functions

The techniques described above consider only univariate functions. However, multivariate polynomials exhibit the same features that allow order computation to be performed by recursively reducing a functions order. That is, $\bar{F}(x, y) = F(x+1, y) - F(x, y) - 2$ reduces the order of F(x, y) with respect to x by one on each iteration if y is held constant. Thus, the order of F(x, y) can be determined with respect to x and with respect y. However, the unique, minimum order polynomial computation requires solving a set of m · n simultaneous linear equations, where m is the order with respect to x and n is the order with respect to y.

5. Implementation of Discontinuous Functions

While the methods described above work well for circuits that implement continuous functions, such as

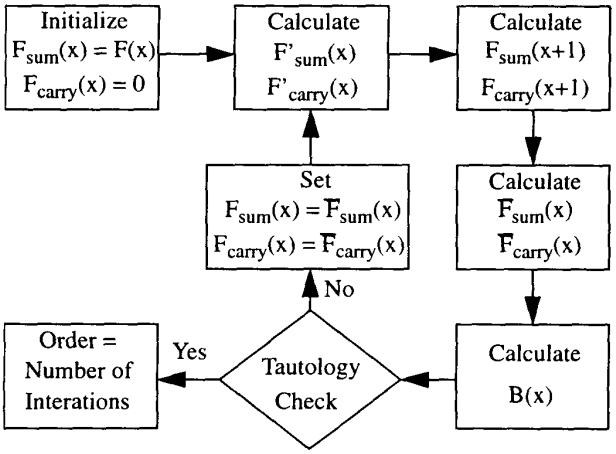

Fig. 3 Algorithm for computing the order of a Boolean function

arithmetic units, some circuits employ branches that result in discontinuous functions. For example, the JPEG Coefficient Encoder selects an output based on the range of the quantized input values

if (q==0) coefficient = 0;
else if (|q| < 2^1) coefficient = 1;
else if (|q| < 2^2) coefficient = 2;
...
else coefficient = 15;

The encoder is continuous within each branch, but discontinuous at $q = 2^i$. Using the order computation methods described above, the discontinuities will cause the minimum order polynomial for q to be of order 2^n, if q is an n bit word. To prevent an exponential number of order computation iterations from being performed on such functions, we define a *discontinuity threshold.* Once the number of iterations has reached this threshold, the function is assumed to be discontinuous. This threshold is determined heuristically.

Branch discontinuities can be detected efficiently, allowing order computation to be performed on each branch of the circuit. Given a function $F: x \rightarrow y$, where $x \in B^n$ and $y \in B^k$, branch discontinuites can be detected by performing an order computation on F(x) for the case $x_n = 0$ and the case $x_n = 1$. If the orders for each computation are different, and below the discontinuity threshold, a discontinuity has been detected and exists between x = 011...1 and x = 100...0. If the order of F(x) for $x_n = 0$ or $x_n = 1$ is still above the threshold, then a discontinuity exists within the corresponding domain. Within that domain, an order computation is then performed on F(x) for the case $x_{n-1} = 0$ and the case $x_{n-1} = 1$. This continues until the discontinuity is detected.

Similar to performing a binary search, detection of a single branch discontinuity is of linear complexity with respect to the number of input bits, not considering the complexity of the order computation. An example is

completed in Appendix A.2.

6. Error Quantification

Polynomial representations provide a means for quantifying the difference between a specification S(x) and an implementation F(x). This can be achieved by computing the polynomial $\epsilon(x) = S(x) - F(x)$ and using traditional numerical methods to find the maximum value of $\epsilon(x)$. In quantifying the maximum error of an implementation, systems traits such as performance and area can be optimized by selecting faster or smaller designs that implement less precise arithmetic.

A means of approximating a specification for non polynomial functions can be derived from the results of Taylor series approximation. If a function $F_{approx}(x) = 1 + (dF(0)/dx)x/1! + (d^2F(0)/dx^2)x^2/2! + ... + (d^nF(0)/dx^n)x^n/n!$, then the difference between $F_{approx}(x)$ and F(x) is $\epsilon(x) = (d^{n+1}F(c)/dx^{n+1})x^{n+1}/(n+1)!$ where $0 < c < x$. Thus, if the error in a Taylor series approximation to a function can be bounded, then the difference between an implementation that matches that approximation and the specification can be bounded. For example, an implementation that is determined to be of order 4 and yields the polynomial $1 - x^2/2! + x^4/4!$ matches the cosine function used in DCT with an error $e < .0083$ over the interval [0, 1].

The ease with which polynomials can be composed can allow seemingly inappropriate implementations to be combined to fulfill a specification. For example, the Boolean function that implements $F(x) = x^2$ may appear to be a completely inappropriate match for the specification cos(x). However, if an adder exists in the implementation library, F(x) can be allocated and composed with the adder to approximate the cos(x): $1 - F(x)/2! + F(F(x)/4!$.

7. Complexity Issues

The techniques described above are of polynomial complexity with respect to input and output word length. Solving the set of linear equations for polynomial coefficients is of cubic complexity with respect to the order of the polynomial, and we assume this order is small (less than 16). However, the underlying OBDD data structure can be of exponential complexity for common functions. Thus, reducing the complexity of polynomial computation requires reducing the complexity of the order computation which, in turn, requires reduction of the complexity of the OBDD.

Assume a function F(x) has an OBDD with 2^n intermediate nodes, where x is an n bit word. If x is partitioned into two words $(x_{n-1}x_{n-2}...x_{n/2}00...0)$ and $(00...0x_{n/2-1}x_{n/2-2}...x_0)$, the OBDDs that describe each partition will require no more than two sets of $2^{n/2}$ intermediate nodes. Similarly, partitioning x into m words will result in a worst case total node count of $T = m \cdot 2^{n/m}$. Minimizing T with respect to m yields:

$$dT/dm = 2^{n/m} - (n/m)\,2^{n/m} \cdot \log_{10}2$$
$$\Rightarrow 2^{n/m} \cdot (1 - n/m \cdot \log_{10}2)$$
$$\Rightarrow m = n \cdot \log_{10}2$$

(1) DCT $\quad DCT = \sum_{i=0}^{7}\sum_{j=0}^{7} x(i,j)$

(2) Quantize $\quad Q = DCT/128 - DC_{previous}$

(3) Coefficient Code $\quad C = \log_2 Q$

(4) DC Code

C	BaseCode	Length
0	010	3
1	011	4
2	100	5
3	00	5
4	101	7
5	110	8
6	1110	10
7	11110	12
8	111110	14
9	1111110	16
10	11111110	18
11	111111110	20

$DC = (BaseCode<<C) + (Q \bmod 2^C)$

Fig. 4 Arithmetic description of the blocks for the DC path of JPEG encode (inputs: x(i, j), output: DC).

Partitioning x into words of length $(1/\log_{10}2) \approx 4$ will minimize OBDD complexity and allow order computation of functions that are of order less than 2^4. This will result in overall OBDD complexity of $(n/4) \cdot 2^4 = 4n$.

This severe reduction in complexity comes at the cost of accuracy. As described above, partitioning is performed by setting each bit of x that is not part of the current partition arbitrarily to zero. Thus, for a function with an 8 bit input, order computation is completed for two partitions: $(x_7x_6x_5x_40000)$ and $(0000x_3x_2x_1x_0)$. This is equivalent to computing the order of the function over the domain x = {0, 1, 2 ... , 15,16, 32, 48, ... , 240}. Without partitioning, order computation is exact and performed considering every possible value of x.

8. Application

Generating polynomial descriptions allows a specification and implementation to be compared simply by comparing the coefficients of the polynomials. Consider the DC path for the JPEG encode system described in Figure 4. The Verilog description for each block of this system was synthesized and the resulting netlists were used as library elements for which polynomial models were computed.

The DCT block requires that an order computation be performed for each input x(i, j). The order of the this block with respect to each input is determined to be one and the resulting polynomial is:

$$DCT = x(0,0) + ... + x(7, 7).$$

The order of the quantize block is similarly determined to be one with respect to DCT[15:7] and $DC_{previous}$ and the resulting polynomial is:

$$Q = DCT[15:7] - DC_{previous}.$$

Order computation for the coefficient and DC encoding blocks yield an order greater than the discontinuity threshold of 4. As a result, the upper bits of the inputs to each block are successively set to 0 and 1, as described in Section 5, and the following intervals and polynomials are determined:

Interval	C Polynomial	DC Polynomial
Q = 0	C=0	DC = 2 + Q
0<Q<2	C=1	DC = 6 + Q
1<Q<4	C=2	DC = 16 + Q
3<Q<8	C=3	DC = Q
7<Q<16	C=4	DC = 80+ Q
15<Q<32	C=5	DC =192+ Q
31<Q<64	C=6	DC = 896+ Q
63<Q<128	C=7	DC = 3840+ Q
127<Q<256	C=8	DC = 15872+ Q
255<Q<512	C=9	DC = 64512+ Q
511<Q<1024	C=10	DC = 26e4+ Q
1023<Q<2048	C=11	DC = 1e6+ Q

The resulting polynomials for generating the output DC match the corresponding polynomials in the arithmetic specification.

9. Experimental Results

To quantify the performance of the polynomial methods presented in this paper, a combinational n bit multiplier was constructed out of combinational 4 bit multipliers. Multiplier logic was synthesized from Verilog to construct the Boolean equations that implement the Synopsys DesignWare multiplier. These equations were then ported to the Cal-2.0 BDD package which was used to perform BDD operations.

The time required to determine the order of this circuit, shown in Figure 5, was obtained running on a 200MHz R4400 Indy Workstation with 64MB of memory. Note that by using the complexity reduction methods from Section 6, order computation was performed on successive 4 bit chunks of each input word. This yielded a maximum BDD size of 61 nodes which fit completely in the 16KB cache.

As expected, execution time varied with the square of the size of the input word. This is due to the function F(x, y) being of order one with respect to each input and having two inputs. Note that a similar computation for F(x) = x + K would have been of linear complexity with respect to the size of x and a more complex function such as $F(x) = x^2 \cdot y^2$ would have varied with the fourth power of the size of the input word.

10. Conclusion

In performing high level synthesis and reusing existing designs, automating allocation requires a means for quickly determining whether an existing block performs the function outlined in the specification. Current methods for completing this task become prohibitively memory intensive or time consuming for circuits that implement non-linear functions. We have developed an algorithm for performing component matching that overcomes these limitations for linear and non-linear functions and is well suited to mapping arithmetic blocks.

Word Sizes	Logic Ops	Exec. Time
4	2003207	0.41s
8	8012236	1.34
16	32050480	4.76
32	128197824	19.31
64	512783104	79.30

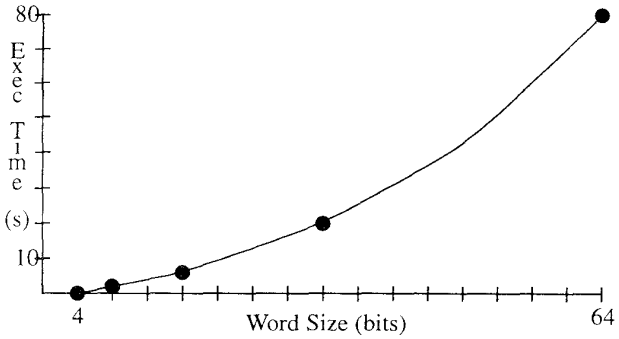

Fig. 5 Execution time required to determine F(x, y) = xy is of linear complexity with respect to x and y.

Circuit specifications can be efficiently matched to existing implementations by generating the unique minimum order polynomial functions for the specification and the implementation and comparing those polynomials. These functions can be generated with polynomial complexity with respect to the number of input bits to each function. Discontinuities in the specification or implementation can be detected, allowing polynomial representations to be computed for intervals between discontinuities. Furthermore, using polynomial representations, differences between a specification and implementation can be quantified, allowing tradeoffs between precision and speed. In addition, the ease with which polynomials can be composed can allow such differences to be compensated for by combining multiple existing blocks or constructing logic around a single block.

While polynomial methods are of polynomial complexity with respect to the number of input bits, they are of exponential complexity with respect to the number of discontinuities. This makes these methods well suited for matching blocks that have compact arithmetic representations, such as those found in DSP, computer graphics, and ALUs, but less efficient for blocks that contain many discontinuities such as controllers.

In this paper, we have considered only combinational circuits. To expand the realm of applicability of these methods, an algorithm for constructing polynomial representations of circuits that employ feedback and sequential elements will be developed.

Acknowledgments

This work is funded under ARPA grant DATB63-95-C0049.

References

[Br86] R. Bryant "Graph Based Algorithms for Boolean

Function Manipulation", IEEE Transactions on Computers, C-35(8), 1986.

[BrCh95] R. Bryant and Y.A. Chen, "Verification of Arithmetic Circuits with Binary Moment Diagrams"", Proceedings of the 32nd ACM/IEEEDesign Automation Conference, p. 535 - 541, 1995

[ChBr96] Y.A. Chen and R. Bryant, "ACV: An Arithmetic Circuit Verifier", Proceedings of the ACM/IEEE International Conference on Computer Aided Design, p. 361-365, 1996.

[ClFu95] E.M. Clarke, M. Fujita, and X. Zhao, "Hybrid Decision Diagrams", Proceedings of the ACM/IEEE International Conference on Computer Aided Design, p. 159 - 163, 1995

[ClFu93] E.M. Clarke, K. McMillan, X. Zhao, M. Fujita, and J. Yang, "Spectral transformations for Large Boolean Functions with Applications to Technology Mapping", Proceedings of the 30th ACM/IEEE Design Automation Conference, IEEE Computer Society Press, 1993

[ChBr96] Y.A. Chen and R. Bryant, "*PHDD: An Efficient Graph Representation for Floating Point Circuit Verification", Proceedings of the ACM/IEEE International Conference on Computer Aided Design, p. 2-7, 1997.

[Mi96] S. Minato, "Implicit manipulation of Polynomials Using Zero-Suppressed BDDs", 1996

Appendix A Examples

In the following examples, we demonstrate the details of order computation and discontinuity detection and show how these techniques are applied to match a simple rasterizer specification to an existing implementation.

A.1 Order Computation

Consider the function $F(x) = x^2$ where x is a 2 bit word. Initializing the sum s to F(x) and the carry c to zero yields the following input vectors:

$s_0 = x_0$ $\quad c_0 = 0$
$s_1 = 0$ $\quad c_1 = 0$
$s_2 = x_0' \cdot x_1$ $\quad c_2 = 0$
$s_3 = x_0 \cdot x_1$ $\quad c_3 = 0$
$s_4 = 0$ $\quad c_4 = 0$

The following steps are followed to determine the order of these input vectors:

(1) F(x+1):

$s_0 = x_0'$ $\quad c_0 = 0$
$s_1 = 0$ $\quad c_1 = 0$
$s_2 = x_0 \cdot (x_1 \oplus x_0)$ $\quad c_2 = 0$
$s_3 = x_0' \cdot (x_1 \oplus x_0)$ $\quad c_3 = 0$
$s_4 = 0$ $\quad c_4 = 0$

(2) F'(x):

$s_0 = x_0'$ $\quad c_0 = 1$
$s_1 = 1$ $\quad c_1 = 1$
$s_2 = x_0 + x_1'$ $\quad c_2 = 1$
$s_3 = x_0' + x_1'$ $\quad c_3 = 1$
$s_4 = 1$ $\quad c_4 = 1$

(3) $\vec{F}(x)$:

$s_0 = 1$ $\quad c_0 = 0$
$s_1 = x_0'$ $\quad c_1 = 1$
$s_2 = x_1 \oplus x_0$ $\quad c_2 = x_1' \oplus x_0$
$s_3 = x_0 + x_1$ $\quad c_3 = x_1'$
$s_4 = x_0' \cdot x_1$ $\quad c_4 = 1$

(4) Tautology Check

$s_0 = 0$ $\quad c_0 = 0$ $\quad$ fails

(5) $\vec{F}^2(x)$

$s_0 = 0$ $\quad c_0 = 1$
$s_1 = 1$ $\quad c_1 = 0$
$s_2 = 1$ $\quad c_2 = 0$
$s_3 = x_0' + x_1'$ $\quad c_3 = x_0$
$s_4 = x_0' \oplus x_1$ $\quad c_4 = x_0' \cdot x_1$

(6) Tautology Check

$s_0 = 1$ $\quad c_0 = 0$
$s_1 = 0$ $\quad c_1 = 0$ $\quad$ fails

(7) $\vec{F}^3(x)$

$s_0 = 0$ $\quad c_0 = 1$
$s_1 = 0$ $\quad c_1 = 0$
$s_2 = 1$ $\quad c_2 = 1$
$s_3 = x_1$ $\quad c_3 = x_1'$
$s_4 = x_0 \cdot x_1'$ $\quad c_4 = x_0 + x_1$

(8) Tautology Check

$s_0 = 1$ $\quad c_0 = 0$
$s_1 = 1$ $\quad c_1 = 0$
$s_2 = 1$ $\quad c_2 = 0$
$s_3 = 1$ $\quad c_3 = 0$
$s_4 = 1$ $\quad c_4 = 0$

Three iterations reduce F(x) to 0 for all x. Thus, F(x) is of order 2.

A.2 Branch Discontinuity Detection

To illustrate how to detect discontinuities, consider the function F(x) where x is a four bit word:

if (x>11) then $F(x) = x^3$;
else $F(x) = x^2$

If we proceed blindly, computing the order of F(x) will generate an order of 2^4 because of the discontinuity at x=11. However, if we start with an initial discontinuity threshold of 4, then after four order iterations, the uppermost bit of x will be set to zero, then one, and the order computations will be performed for each case. The order computation for x_3=0 will result in an order of 2. The order computation for x_3=1 will again reach the fourth iteration without converging. The second most significant bit is set to zero, then one, and the order computation is performed again. Then order computation for x_3x_2=11 will result in an order of 3 and the computation for x_3x_2=10 will result in an order of 2. Since both computations converged, but converged to different values, there is a discontinuity on the interval boundary. Thus, in the interval [0, 11] an order of 2 will be determined and in the interval [12, 15] and order of 3 will be determined.