Symbolic Algebra and Timing Driven Data-flow Synthesis

Armita Peymandoust **Giovanni De Micheli**

Computer Systems Laboratory, Stanford University
Stanford, CA 94305

{armita, nanni}@stanford.edu

Abstract

The growing market of multi-media applications has required the development of complex ASICs with significant data-path portions. Unfortunately, most high-level synthesis tools and methods cannot automatically synthesize data paths such that complex arithmetic library blocks are intelligently used. Symbolic computer algebra has been previously used to automate mapping data flow into a minimal set of complex arithmetic components. In this paper, we present extensions to the previous methods in order to find the minimal critical path delay (CPD) mapping. A new algorithm is proposed that incorporates symbolic manipulations such as tree-height-reduction, factorization, expansion, and Horner transformation. Such manipulations are used as guidelines in initial library element selection. Furthermore, we demonstrate how substitution can be used for multi-expression component sharing and critical path delay optimization.

1. Introduction

Automating the design of data paths from high-level specifications is necessary to meet the tight time to market requirements of multi-media applications. The optimal choice of the arithmetic units implementing complex data flows strongly affects the cost, performance and power consumption of the silicon implementations. Unfortunately, current commercial tools rely on synthesis directives (pragmas) from designers in order to map data-flow into complex arithmetic library elements.

However, current high-level synthesis tools are effective in capturing HDL models of hardware and mapping them into control/data-flow graphs (CDFGs), performing scheduling, resource sharing, retiming, and control synthesis [1]. The approach presented in this paper fits seamlessly into the current high-level synthesis flow. We propose to analyze the data-flow segments of the CDFG models in light of the arithmetic units available as library blocks, and to construct data paths that exploit at best the given library. We assume that design is done using libraries that contain beyond the basic elements such as adders and multipliers, more complex cells such as multiplier-accumulator (MAC), sine, cosine, etc. An example of such a library is Synopsys Designware [2] library.

Two factors are key in automating the optimal mapping of data flow blocks. First, a functionality description formalism for data flow and library components. Second, a method supporting the decomposition of the data flow into a set of library elements. Polynomial representation has been proven as an effective technique for representing both high-level specification and bit-level description of an implementation (library component) [3, 4]. Symbolic computer algebra has been utilized to decompose the polynomial representation of a data path into library elements in the symbolic synthesis tool SymSyn [5]. However, the algorithm implemented previously in SymSyn was limited to find the minimal-component mapping of a given data flow.

Due to the importance of high performance design, we have developed a new algorithm in the SymSyn framework to automatically map the data flow to arithmetic library elements such that the optimum critical path delay is achieved. For such purpose, we leverage results from Gröbner basis applications and symbolic polynomial manipulation techniques. In this paper, we propose a timing-driven decomposing algorithm that uses various polynomial manipulation techniques guidelines to achieve optimal component mapping and resource sharing.

The paper is organized as follows: Section 2 describes related work and gives a brief overview of data-flow synthesis using symbolic computer algebra. Section 3 provides an insight to the polynomial manipulations used for delay optimization with simple examples. In Section 4, we present how we can leverage results from symbolic algebra and methods previously used in logic minimization to construct an algorithm that finds a minimal-critical path delay decomposition of a polynomial representing a (portion of) data flow. Finally, Section 5 presents some experimental results.

2. Related Work

Our tool, SymSyn [5] maps a (portion of) data flow to complex arithmetic library elements such that a given criteria is satisfied (minimal number of components or minimal critical path delay). In order to achieve this goal, we assume that the data flow is represented as a polynomial. This assumption is based on the fact that polynomial approximations such as Taylor, Pade, continued fractions, Hermite-Pade, and others are generally used to implement complex arithmetic functions in hardware [8]. This is especially the case for the growing market of multi-media applications. We also require the library elements to have a polynomial representation associated with them. In previous work [3,4], a mechanism was provided to derive a

minimum order word-level polynomial representation for a given complex Boolean circuit description. This method provides polynomial representations for legacy library elements without trivial polynomial representations.

2.1 Symbolic algebra and data-path synthesis

Traditional mathematical computation with computers and calculators is based on arithmetic of fixed-length integers and fixed-precision floating-point numbers, otherwise known as numeric computer algebra. In contrast, modern symbolic computation systems support exact rational arithmetic, arbitrary-precision floating-point arithmetic, and algebraic manipulation of expression containing undetermined values (symbols), such as variable `x` in `(x+1)*(x-1)`. Several commercial symbolic computer algebra systems are available on the market; Maple [13] and Mathematica [14] are most widely used.

The algebraic object that we are interested to manipulate symbolically is a multivariate polynomial that represents a (portion of) data path of our design. Most symbolic polynomial manipulations that we find interesting in data-path synthesis are based on Gröbner bases [9-12]. Gröbner bases and Buchberger's algorithm generalize the division and GCD algorithms of univariate polynomials to multivariate polynomials. Therefore, it is the heart of symbolic polynomial factorization. Gröbner bases also solve variable elimination in a set of polynomials and ideal membership problems, which is the core of simplification modulo set of polynomials. We will not dive into the details of Gröbner bases and Buchberger's algorithm and how they are applied to symbolic polynomial manipulation. We refer the reader to the literature available in mathematics [9-12].

In order to show the power of symbolic algebra, namely the *simplification modulo set of polynomials* routine, let us consider a simple example. The built-in function that implements this routine is called *simplify* in Maple [13]. In order to comply with Maple terminology, we call the set of polynomials the *side relations.* Consider a data flow implementing `x^2-y^2` and a library that includes add, multiply subtract and square components. Using Maple syntax we have:

```
> a:=x^2-y^2: siderels:={b=x-y, c=x+y}
> simplify(a, siderels,[x,y,b,c]);
  b*c
```

This is equivalent to the implementation shown in Figure 1.a. Note that `siderels` is a subset of our library. Maple computes the Gröbner basis *G* of `siderels` and uses that internally to eliminate `x` and `y` variables from the polynomial. The result indicates that:

```
a:=x^2-y^2:=b*c:=(x-y)*(x+y)
```

If the side relation set is changed, other possible solutions for the specification may be found, for example:

```
> a:=x^2-y^2: siderels:={b=x^2, c=y^2}
> simplify(a, siderels,[x,y,b,c]);
  b-c
```

results in the implementation shown in Figure 1.b.

Choosing the side relation set is a non-trivial task. In previous work [5], an algorithm was introduced to select the side relation set such that the implementation is minimal in term of library component count. In this paper, we describe an algorithm to exploit the library components such that the CPD of the data path implementation is minimized.

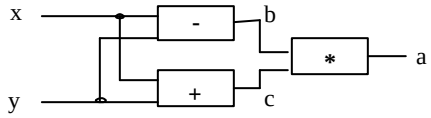

Figure 1.a. An implementation of x^2-y^2

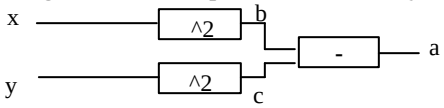

Figure 1.b. Another implementation of x^2-y^2

3. Expression Manipulation Techniques

Previously [5], an algorithm was introduced that maps a polynomial representation of a (portion of) data flow to minimal number of complex arithmetic library elements. This algorithm was implemented in the Symbolic Synthesis tool, *SymSyn*. To enhance SymSyn such that it enables critical path delay minimization, a guideline is necessary for side-relation selection. Such guideline should facilitate mapping for maximum parallelism. We have chosen different symbolic polynomial manipulation techniques as such guidelines. The intent of this section is to describe the manipulation techniques through simple examples.

3.1 Tree-height Reduction

Tree-height reduction (THR) was introduced long ago [6,7] as an optimization method for parallel software compilers. It is a technique to reduce the height of an arithmetic expression tree, where the height of the tree is the number of steps required to compute the expression. In the best case, it achieves the tree height of $O(\log n)$ for an expression with n operations. Tree-height reduction uses commutativity, associativity, and distributivity properties of addition, subtraction and multiplication. In the classical case, tree-height reduction is achieved at the expense of adding more resources to obtain maximum parallelism in the expression. In previous work, THR has been proven useful in high-level synthesis of data-intensive circuits such as DSP and multimedia applications [15-17].

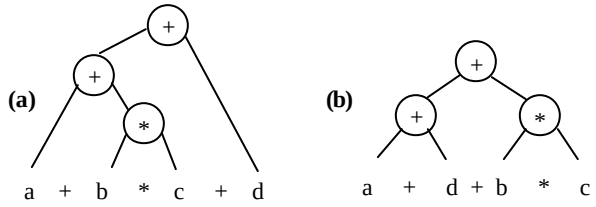

Figure 2. Performing THR on (a) produces (b)

In our work, we use THR as an expression tree manipulation technique. THR will achieve the best

execution time when using unlimited number of two input adders, subtracters and multipliers is allowed. Since we are focusing on libraries that have more complicated blocks, THR may or may not result in the optimal execution time. The result is dependant on the library components available. Figure 2 shows an example of how THR can reduce the critical path delay. Figure 2 (b) is obtained after applying THR on Figure 2 (a).

3.2 Factor and Expand

As mentioned previously, traditional tree-height reduction [6, 7] only uses associativity, commutativity, and distributivity to transform expressions. Since we have access to a symbolic manipulation tool in SymSyn, we can benefit from other transformations as well. One such transformation is common sub-expression factorization. Factorization can reduce the number of components used as well as the tree height of a given expression. An example is shown in Figure 3.

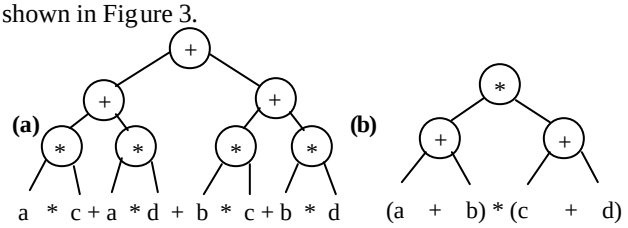

Figure 3. Factor may reduce number of components and CPD

Another useful symbolic manipulation technique is expansion. This manipulation technique changes the polynomial into its sum of products format. Meanwhile, it is capable of straightforward simplification techniques that can save both delay and area. A small example of such simplification is transforming `a+a+a` to `3*a`.

3.3 Horner form

Horner form of a polynomial is a nested normal form with minimal number of multiplications and additions. Any polynomial can be rewritten in Horner, or nested, form. The general univariate case is defined as follows [14]:

$$p(x) = a_0 \cdot x^n + \cdots + a_{n-1} \cdot x + a_n$$
$$= (\cdots((a_0 \cdot x + a_1) \cdot x + a_2) \cdot x + \cdots a_{n-1}) \cdot x + a_n$$

Assume that x^n can be calculated using only $\log_2(n)$ multiplications for integer n. For a polynomial of degree n, the Horner form requires n multiplications and n additions. The expanded form, however, requires:

$$\sum_{i=1}^{n} \log_2(i) = \log_2(n!)$$

multiplications, which is more than twice as expensive for a polynomial of degree 10. Thus, one advantage of Horner form is that the work involved in exponentiation is distributed across addition and multiplication, resulting in savings of some basic arithmetic operations. Another advantage is that Horner form is more stable to evaluate numerically when compared with the expanded form. The reason for this is that each sum or product involves quantities which vary on a more evenly distributed scale. For hardware implementation, Horner form has a distinct advantage. It effectively maps a univariate polynomial to cost effective multiplier-accumulators (MAC).

Horner form is generalized for multivariate polynomials by specifying an ordered list of variables. As a simple example consider the following polynomial in which the number of multiplications is reduced from 32 to 13:

```
› S:=x^3+3*x^2*y+x^2+3*x*y^2+2*x*y+2*x^2*z+
     y^3+y^2+2y^2*z+2*y*z+z^2*x+z^2*y+z^2;
› convert(S, 'horner', [x,y,z];
  z^2+((2+z)*z+(2*z+1+y)*y)*y+((2+z)*z+
  (2+4*z+3*y)*y+(2*z+1+3*y+x)*x)*x
```

3.4 Substitution

Substitution is defined as replacing a subexpression by a previously computed variable [1]. It reduces complexity of a function by using an additional variable that was not previously in its support set. This transformation creates a new dependency between expressions, but may also eliminate previous dependencies. Substitution has been previously used in multi-level combinational logic optimization [18,19]. Elimination theory [12] based on the Gröbner basis formalizes substitution and variable elimination for multivariate polynomials. We refer the interested reader to the reference [12] for the detailed mathematical proof. Note that for arithmetic polynomials, use of a more general decomposition model is necessary as compared to the algebraic division modeled in combinational logic synthesis. This is due to the fact the Boolean idempotence property does not hold in arithmetic polynomials and exponents are valid. Therefore, there is no restriction on the support set of the divisor and quotient of an expression. For example:

$\frac{x^2 - y^2}{x - y} = x + y$ is a legitimate division.

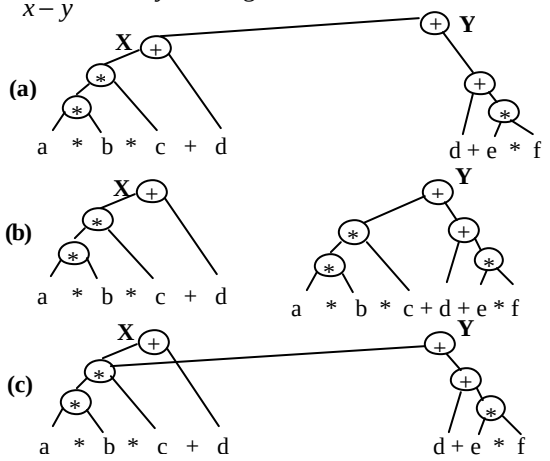

Figure 4. Substitution with THR can maximize parallelism

Substitution can be combined with THR in order to select a subexpression that maximizes parallelism. As a

simple example let us consider a basic block which consists of two arithmetic expressions:

```
X:= a*b*c+d;
Y:= X+e*f;
```

It can be seen that Y is dependent on X, therefore Y is calculated after the value of X is known as shown in Figure 4.a. However, if we re-substitute X in Y, Y:=a*b*c+d+e*f, Y can be evaluated in parallel with X. Figure 4.b shows the results of tree-height reduction on both X and Y expressions. In order to achieve maximum parallelism between X and Y, we must substitute only subexpression a*b*c in Y, this is shown in Figure 4.c.

4. Timing Driven Decomposition Algorithm

In this section, we introduce a new polynomial decomposition algorithm that in conjunction with classical high-level synthesis algorithms can be used for efficient algorithmic-level DSP synthesis. The core of this algorithm is *simplification modulo set of polynomials* and Gröbner basis fundamentals described in Section 2.1. After extracting the CDFG of an algorithmic level DSP model, Algorithm 4.1 intelligently decomposes the data flow to library components such that the critical path delay of the implemented data path is minimized.

As seen in Section 2.1, different side relation sets result in different implementations of the data-flow specification. Therefore, to find the best possible implementation, the side relation set should be set equal to all subsets of the library. Since this is exponentially expensive, a guided architectural exploration is necessary. The algorithm presented in this section, uses the branch-and-bound method to reduce the search space. We define the bounding function as the best critical path delay of implementations seen so far. The lower bound computed at each decision branch is the critical path delay of components in the side relation set in view of data dependencies. If this lower bound is greater than the best critical path delay of implementations seen so far, the corresponding decision branch is pruned.

In general, the branch-and-bound algorithm is practically applicable to most problems. However, introducing heuristics that lead quickly to promising solutions can improve the execution time without hampering the quality of the solution. We use the expression manipulation techniques presented in Section 3 as heuristic guidelines for choosing the side relation set. As all branch-and-bound algorithms, the worst-case complexity remains exponential.

Algorithm 4.1 shows the pseudo-code of the proposed timing-driven algorithm. Let *S* be the polynomial representation of the data flow. Our goal is to decompose *S* into the elements of the library *L* such that the critical path delay of *S* is minimized. Decomposing *S* is synonym to simplifying *S* modulo elements of the library *L* as side relations. In order to decide which library elements should be used as the side relations, we use a decision tree (*solution_tree*) to implement the branch-and-bound algorithm. The bounding variable is initialized to the critical path delay of mapping the polynomial solely to adders and multipliers, the lexicographical mapping.

The *simplify* results are also saved in the tree data structure. If a simplification result is identical (or within an acceptable tolerance) to the polynomial representation of a library element, a possible solution is found and the corresponding tree node is marked accordingly. If the critical path delay of the solution is less than previously encountered solutions, we set the bounding variable to the current delay. In case the simplification result stored in a tree node does not correspond to any library elements, we recursively apply the same steps to the new tree node.

Algorithm 4.1 Decompose *S* into elements of library *L*

```
function GuidedDecomposition(exp_tree, max_CPD){
  # initialize a solution tree
  solution_tree ← tree(exp_tree);
  depth ← 0
  bound ← max_CPD
  for all n ∈ in tree with depth depth do{
    if depth ==0 then
      choose all sr ∈ L that preserve the expression tree structure
    else for all sr ∈ L do{
      if cost of sr + cost of node n < bound then {
        result = simplify(n, sr);
        # make result a child of node n
        addchild(n, result);
        add cost of sr to cost of result;
        depth ← depth +1
        if result ∈ L then {
          # solution is found
          bound = cost of node result; }}}}
  return the best solution in solution_tree
end

int function CalcMaxCPD(expression_tree){
  CPD = the critical path delay of expression_tree assuming
        the expression is mapped to adders and multipliers only.
  return(CPD)
end

procedure main(S, L)
  # Given a polynomial representation of the spec S
  # and a set of polynomials L as component library,
  # decompose S into elements of library L such that
  # the critical path delay (CPD) of S is minimized.
  # perform expression manipulation techniques
  exp_tree[1..NumberOfManipulations]=AllManipulations(S);
  for i= 1 to NumberOfManipulations do{
    maxCPD[i]=CalcMaxCPD(exp_tree[i]);
    solution[i]=GuidedDecomposition(exp_tree[i], maxCPD[i]);
  }
  report the best solution in solutions[i]
end
```

In order to reduce the execution time, we introduce heuristics to select the initial side relation. The heuristics used are based on the symbolic polynomial manipulations described in Section 3. Initially, we apply tree-height reduction, factorization, expansion, and Horner-based transform on *S*. As a result, we have several polynomials (*exp_tree*) representing the same data flow. Each of these representations can result in the desirable implementation based on the available library elements. Starting with the

primary inputs, we try covering the expression tree with the library elements. We choose all library elements that cover the primary inputs and a portion of the expression tree as a side relation. If the result of simplify modulo side relation is not a library element, we decompose the result without further guidance from the expression tree. Algorithm 4.1 in conjunction with substitution and tree-height reduction can be generalized to several polynomials in basic block or across basic blocks.

As an example, consider a data-flow segment of a Gabor filter with the following polynomial representation:

$$S = 1 - a^2 - b^2 + a^2b^2 + \frac{1}{2}a^4 + \frac{1}{2}b^4 - \frac{1}{6}a^6 - \frac{1}{2}a^4b^2 - \frac{1}{2}a^2b^4 - \frac{1}{6}b^6 + \frac{1}{24}a^8 + \frac{1}{6}a^6b^2 + \frac{1}{4}a^4b^4 + \frac{1}{6}a^2b^6 + \frac{1}{24}b^8$$

Assume we would like to map S to a library consisting of functions implementing add, multiply, MAC, square, exp. After factorization S will be converted to:

$$S = \frac{1}{24}(a^2 + b^2)\,(a^6 - 4a^4 + 3a^4b^2 + 12a^2 - 8a^2b^2 + 3a^2b^4 + 12b^2 + 24 - 4b^4 + b^6) + 1$$

The factored form of `S` guides us to use `c=a^2+b^2` as an initial side relation and sets an initial bound by mapping the factored form lexicographically to adders and multiplier. SymSyn makes a call to Maple and requests result of the following `simplify` operation.

```
> siderel := {c=a^2+b^2};
> result:=simplify(S, siderel, [a,b,c]);
  result=1-c+1/2*c^2-1/6*c^3+1/24*c^4
```

The last line is the result reported to SymSyn by Maple. As it can be seen, the result is a Taylor series expansion of `exp(c)`. Therefore, the data flow can be implemented using two square blocks, an adder, and an `exp` block, as shown in Figure 5. The bounding function is now changed to the new implementation found. By exploring the other branches of the decision tree (solution tree).

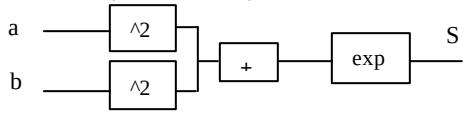

Figure 5. Mapping S to 4 components

Now, assume that there is no `exp` block in our library. In order to show the power of other polynomial transformations, we perform Horner transform on the result polynomial, we obtain:

$$result = 1 + (-1 + (\frac{1}{2} + (-\frac{1}{6} + \frac{1}{24}\cdot c)\cdot c)\cdot c)\cdot c$$

The formula given above can be implemented using a chain of 4 MACs, or one MAC in 4 cycles. Figure 6 demonstrates one possible implementation.

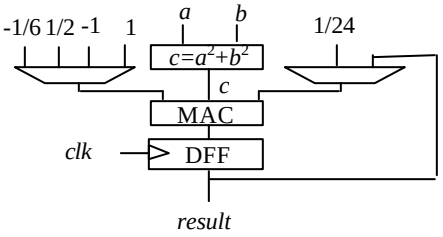

Figure 6. A possible implementation for e^C

5. Implementation and Experimental Results

We have implemented Algorithm 4.1 described in this paper in C programming language with calls to Maple V [13] for the symbolic manipulations in the SymSyn framework. The program input is the data flow of a high level description of the design and a database of polynomial representations of library elements. Output reported is components used in the data-path implementation and their interconnection such that the critical path delay is minimized.

Table 1. SymSyn results for some examples

Data flow Examples	Lexicographical Mapping		SymSyn Output	
	#of comps	CPD	#of comps	CPD
a^2-b^2	3	2.35	3	2.05
b^3+ba^2c	9	5.05	2	4.69
IDCT	9	3.7	2	3.34
1/2tanh(a-1)+ 1/2tanh(a+1)	12	8.75	3	5.63
PSK	33	7.75	2	7
Geometric-transform	12	11.45	5	7.92
Gabor-transform	79	13.8	6	9.4

We have tested the efficiency of SymSyn with a number of data-path examples. The results are shown in Table I. In this table, the critical path delay reported is normalized by the critical path delay of an adder. For example, the critical path delay of an adder is 1 and critical path delay of a multiplier is 1.35. This number is calculated from the critical path delay reported by DC for a 16-bit multiplier divided by the critical path delay reported by DC for a 16-bit adder. The normalized critical path delay calculation is done for all library components available in the DesignWare arithmetic component library [2].

In the first set of results, we assume that the polynomial representation is mapped only to multipliers and adders. This is same as lexicographical component inference that is typical in commercial behavioral synthesis tools. The number of components refers to the numbers of adders and multipliers in the data-path polynomial. The critical path delay (CDP) reported is an accumulative delay of components on the critical path. The second set of results is derived by SymSyn. The cost is sum of cost of the components used in data path to implement the polynomial

representation as recommended by SymSyn. The library used for the examples is the DesignWare library [2] plus the *tanh(x)* and *exp(x)* operations.

The first two data flows in Table I are simple benchmark polynomials. The third polynomial is a basic block in a one-dimensional inverse discrete cosine transform (IDCT). IDCT is widely used in audio and video compression standards such as JPEG, MPEG, and MP3. The fourth example comes from the digital communication field. It performs phase shift keying (PSK) modulation. The fifth example used in graphics for image rotation. And the last example is a data flow segment of the Gabor transform used in neural systems.

Table 2. Reported by Design Compiler using tsmc.35 library

Data flow Examples	DC Results without SymSyn		DC Results with SymSyn	
	Timing	Area	Timing	Area
a^2-b^2	11.21	66760	9.42	54815
b^3+ba^2c	29.09	285926	25.44	166303
IDCT	29.29	323185	20.52	130753

With the intention of achieving more precise measurement of the critical path delay of the set of examples, we used Behavioral Compiler (BC) and Design Compiler (DC) to produce the set of results shown in Table 2. Due to the limitation of library components available and DC, this experiment was restricted to the first three examples. The first set of results is the output of DC on the standard behavioral synthesis flow. The second set of results is the output of DC when mapping directives suggested by SymSyn are incorporated in the HDL input to BC. It can be observed that actual performance improvements in these examples are inline and better than estimated by SymSyn in Table 1.

In summary, the results show that we can achieve an average performance improvement of 17.6 percent over commercial behavioral synthesis flow. It is also shown that even though we are aiming for performance improvement, we also achieve significant area improvement by using less number of components than lexicographical mapping.

6. Conclusion

In this paper we have introduced a timing driven architectural decomposition algorithm in order to map data flow to a set of complex arithmetic library components. This algorithm fits seamlessly in the high-level synthesis flow and enhances the quality of result of data intensive circuit synthesis. Our method takes advantage of previously developed concepts; polynomial representation of library blocks, symbolic computer algebra, and transformations previously used in the fields of multi-level combinational logic minimization and parallel computing.

Polynomial representation is used to represent the functionality of library components and the data flow segment of the chip under design. Symbolic computer algebra is used to decompose the data flow into a set of library components. From a practical standpoint, the contribution of this paper is to make performance constrained arithmetic library binding an automated process, and eliminate the need for synthesis directives.

Symbolic computer algebra is a powerful set of algorithms not previously used in the field of synthesis. We believe these algorithms open a new set of opportunities in algorithmic-level synthesis research. Even though algebraic manipulations are best suited for combinational arithmetic designs, scheduling, resource sharing, and retiming algorithms can be applied to the data-path output to achieve optimized/pipelined designs.

7. Acknowledgments

This research is supported by ARPA/MARCO Gigascale Research Center and Synopsys Inc. We would like to thank both organizations for their support.

8. References

[1] G. De Micheli, "Synthesis and Optimization of Digital Circuits", Mc Graw Hill, Hightstown, NJ, 1994.
[2] DesignWare Library, http://www.synopsys.com/, 1994.
[3] J. Smith and G. De Micheli, "Polynomial Methods for Component Matching and Verification", *Proceedings of the International Conference on Computer Aided Design*, 1998.
[4] J. Smith and G. De Micheli, "Polynomial Methods for Allocating Complex Components", *Proceedings of the Design, Automation, and Test in Europe Conference*, 1999.
[5] A. Peymandoust and G. De Micheli, "Using Symbolic Algebra in Algorithmic Level DSP Synthesis", *Proceedings of the Design Automation Conference*, pp. 277-282, 2001.
[6] D. J. Kuck, "The Structure of Computers and Computations. Vol. I", *John Wiley and Sons*, New York, NY, 1978.
[7] D. J. Kuck, Y. Muraoka, and S. C. Chen, "On the Number of Operations Simultaneously Executable in Fortran-like Programs and Their Resulting Speedup", *IEEE Trans. On Computers*, C-21, 12, December 1972.
[8] J. F. Hart et al., "Computer Approximations", New York: Wiley, 1968.
[9] B. Buchberger, "Some Properties of Gröbner Bases for Polynomial Ideals", *ACM SIG-SAM Bulletin*, 1976.
[10] K. Geddes, S. Czapor, and G. Labahn, *Algorithms for Computer Algebra*, Kluwer Academic Publishers, 1992.
[11] T. Becker and V. Weispfenning, *Gröbner Bases*, Springer-Verlag, New York, NY, 1993.
[12] D. Cox, J. Little, and D. O'shea, "Ideals, Varieties, and algorithms", Springer-Verlag, New York, NY, 1997.
[13] Maple, Waterloo Maple Inc., www.maplesoft.com, 1988.
[14] Mathematica, Wolfram Research Inc., www.wri.com, 1987.
[15] A. Nicolau and R. Potasman, "Incremental Tree Height Reduction for High Level Synthesis", *Proceedings of the Design Automation Conference*, pp. 770-774, 1991.
[16] D. Kolson, A. Nicolau, and N. Dutt, "Integrating Program Transformations in the Memory-Based Synthesis of Image and Video Algorithms", *Proceedings of the International Conference on Computer Aided Design*, November 1994.
[17] H. Wang, A. Nicolau, and K. Siu, "The Strict Time Lower Bound and Optimal Schedules for Parallel Prefix with Resource Constraints", *IEEE Trans. On Computers*, November 1996.
[18] R. Brayton and C. McMullen, "The Decomposition and Factorization of Logic Synthesis", *IEEE Int. Symp. Circuits Syst.*, May 1982.
[19] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A. Wang, "MIS: A Multiple-level Logic Optimization and the Rectangular Covering Problem", Proceedings *of the International Conference on Computer Aided Design*, 1987.