Statistical On-Chip Interconnect Modeling: An Application of Automatic Differentiation

Christian Bischof and Lucas Roh
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
{bischof,roh}@mcs.anl.gov

Norman Chang, Ken Lee, Valery Kanevsky, O. Sam Nakagawa, and Soo-Young Oh
Hewlett Packard Laboratory
3500 Deer Creek Rd.
Palo Alto, CA 94304

Argonne Preprint ANL/MCS-P698-1097

ABSTRACT

Automatic differentiation is a technique for computing derivatives accurately and efficiently with minimal human effort. We employed this technique to generate derivative information of FCAP2 (2-D) and FCAP3 (3-D) programs that simulate the parasitic effects of interconnects and devices. This derivative information is used in the statistical modeling of worst-case interconnect delays and on-chip crosstalks. The ADIC (Automatic Differentiation in C) tool generated new versions of FCAP2 and FCAP3 programs that compute both the original results and the derivative information. Given the ANSI C source code for the function, ADIC generates new code that computes derivatives of the model output with respect to the input parameters. We report on the use of automatic differentiation and divided difference approaches for computing derivatives for FCAP3 programs. The results show that ADIC-generated code computes derivatives more accurately, more robustly, and faster than the divided difference approach.

1 Introduction

As the geometry of VLSI/ULSI chips shrinks, the parasitic effects of interconnects become very important. The resulting interconnect delay can dominate the critical path delay of the circuits (which in turn determines the operating frequency of the chips). Because of process variations, the critical path delay varies with the set of interconnects and devices. Thus, we need accurate models of the sensitivities of the delay with respect to these interconnects and devices in order to determine the worst-case behaviors. For this purpose, the modeling of 3-sigma delays is more desirable than determining the traditional skew-corner worst cases. Since a chip has millions of interconnects, a fast method is necessary to generate statistical-based worst case modeling of interconnects and devices.

FCAP2 and FCAP3 (Fast Capacitance Extraction 2-D and 3-D simulators, respectively) [9] have been developed at Hewlett Packard Laboratory to study parasitic electrical effects of interconnects and devices. These codes are based on the finite-difference method. Given a set of interconnect geometry and bias conditions, the codes can, for example, compute the capacitance among conducting wires. The statistical modeling of interconnect methodology relies on accurate derivatives of the FCAP-generated results. Specifically, we are interested in computing the derivatives of the output variables (such as the capacitance between the wires) with respect to input variables (such as the spacing between the wires on the same or different layers). A single run of on-chip statistical modeling takes on the order of a week on a fast workstation, and most of this time is spent in computing the derivatives.

The traditional approach to obtaining derivatives is to estimate them by using divided difference schemes. For example, with central divided differences,

$$\mathbf{f'(x) \approx (f(x + \Delta x) - f(x - \Delta x)) / 2\Delta x.}$$

The advantage of this approach is that the function (in this case, the simulator) can be treated as a black box. The disadvantage is that the time required to compute derivatives grows linearly with the number of independent variables, and the accuracy of derivatives may be compromised severely as a result of truncation and cancellation errors [16].

Instead of approximating derivatives by using divided differences, we can use hand coding or symbolic differentiation. These techniques, however, are generally not feasible for large codes such as the FCAP programs. Recently, the automatic differentiation technique has been gaining popularity because of its ability to produce accurate derivatives for general codes in an automated fashion. Several tools that incorporate automatic differentiation techniques have been developed that take a computer program comprising the function to be differentiated, such as FCAP2/3, and generate a new program that evaluates the derivative of the function with respect to the specified independent variables, in addition to computing the original function. No limits are imposed on the length or the complexity of the program. Hence, general techniques that rely on the output of computer simulation models, such as optimal design and sensitivity or reliability analysis, can all benefit from using automatic differentiation. See, for example, the work in [4, 6, 8].

In this paper, we describe how we applied our automatic differentiation tool, called ADIC (Automatic Differentiation in C), to generate derivative codes for FCAP2 and FCAP3 programs. We also present runtime performance results of the derivative code for FCAP3. The results show that ADIC-generated code computes derivatives more accurately and up to twice as fast as the divided-difference methodology. We also describe our postoptimization steps that have improved the performance by a factor of two, on the average, over the straight ADIC-generated code.

2 The ADIC Automatic Differentiation Tool

In this section, we briefly review automatic differentiation techniques and describe our tool that implements the techniques. Every function, no matter how complicated, is executed on a computer as a (potentially very long) sequence of elementary operations (addition, multiplication, etc.) and elementary functions (sine, cosine, etc.). By applying the chain rule of differential calculus over and over again to the composition of those elementary operations, one can compute the derivative information exactly (up to the machine precision) and in a completely mechanical fashion [17].

We illustrate the idea with a trivial example. Assume a function $f\colon x \in R^n \Rightarrow y \in R^m$ and that we wish to compute the derivative of y with respect to x. Here, x is called an independent variable and y the dependent variable.

$$a = x[1] + x[2] \qquad (1)$$

$$y[0] = a / x[2] \qquad (2)$$

By means of the chain rule, derivatives can be propagated forward in a mechanical fashion. Let us denote the derivatives of a variable t with respect to a chosen set of independent variables by ∇t. Then the statement (1) implies

$$\nabla a = \nabla x[1] + \nabla x[2],$$

and the chain rule, applied to the statement (2), yields

$$\nabla y[0] = \frac{\partial y[0]}{\partial a} \times \nabla a + \frac{\partial y[0]}{\partial x[2]} \times \nabla x[2],$$

which evaluates to

$$\nabla y[0] = 1.0/x[2] \times \nabla a + (-a/x[2] \times x[2])) \times \nabla x[2].$$

This mode of differentiation, where the derivatives are maintained with respect to the independent variables, is called the forward mode of automatic differentiation: derivatives are accumulated in the same order as the original execution order. The best-known alternative to the forward mode is the reverse mode, which maintains derivatives of intermediate values with respect to the final results and is a discrete analog of the adjoint: derivatives are accumulated in the reverse order of program execution. This mode is attractive for the computation of derivatives of few dependent variables with respect to many independent variables, but requires extensive runtime tracing to store or recompute intermediate quantities that are required in the backward pass. Because of the associativity of the chain rule of differential calculus, many other ways of computing derivatives are possible. Each way may differ significantly in complexity, with respect to both floating-point operations and memory, depending on the code. These issues are discussed in some of the contributions in the proceedings edited by Griewank and Corliss [18] and Berz et al. [1]. Derivative accuracy is a built-in feature of automatic differentiation although in some cases care must be taken to distinguish between the mathematically derived continuous derivatives and the ones defined by the discretized algorithm; see, for example, Eberhard and Bischof [14].

Several tools have been developed to handle the automatic differentiation process. They include ADIFOR [2,3], ODYSSEE [21], and ADOL-F [22] for Fortran programs and ADOL-C [19] and ADIC [7] for C programs. For an up-to-date account, readers are referred to the documentation available on the World Wide Web under URL http://www.mcs.anl.gov/autodiff/adtools/.

In our work, we employed the ADIC (Automatic Differentiation in C) tool. Given an ANSI C routine or a collection of routines describing a function, ADIC uses a source-to-source program transformation technique to produce a new, portable C code that computes derivatives of the output variables with respect to any independent variables. ADIC has the following features:

- **Ease of Use**: The user supplies the set of ANSI C source files to be differentiated. ADIC then produces a new ANSI C code that computes first-order derivatives with respect to the specified independent variables. After specifying independent variables in a process typically referred to as "derivative seeding" (see [3,7]), the user invokes the derivative code. ADIC also allows the user to guide the derivative generation process by specifying problem- and domain-specific knowledge in an optional control script.
- **Generality and Portability**: ADIC provides AD functionality for all ANSI C constructs, including, for example, subroutine and function calls, structures, and pointers. The generated code is generally portable across different platforms and compilers.
- **Extensibility**: AD technology is still in its infancy. The development of better algorithms for exploiting chain rule associativity and their incorporation into AD tools promise significant improvement in the performance of generated derivative codes. ADIC incorporates a component architecture called AIF (AD Interface Form) that simplifies implementation of AD algorithms and enables sharing across different language front-ends [7].

3 Differentiated Models: FCAP2/FCAP3

The development of the FCAP2/FCAP3 codes was started at HP Lab in the 1980s. These codes have served many purposes in simulating capacitance, resistance, and thermal modeling of simple as well as complicated devices and on-chip/off-chip interconnects at HP. HP also licensed FCAP2/3 to TMA (Technology Modeling Associate) Inc. and the codes were incorporated in the industry-standard capacitance/inductance extraction tool called Raphael^{TM}.

For a given set of geometries and bias condition, FCAP2/FCAP3 solve the Poisson equation using the finite difference method with self-adjusting rectangular grid and the incomplete Cholesky conjugate gradient (ICCG) method [15]. FCAP codes embody a modeling language with which users write scripts to specify the geometry, bias conditions, and the parasitic effects to be simulated.

4 Using ADIC to Generate Derivative Codes for FCAP

In this section, we describe how we have differentiated FCAP codes using ADIC and the hand optimizations that we performed to improve the performance of the derivative codes.

4.1. Basics of ADIC Processing

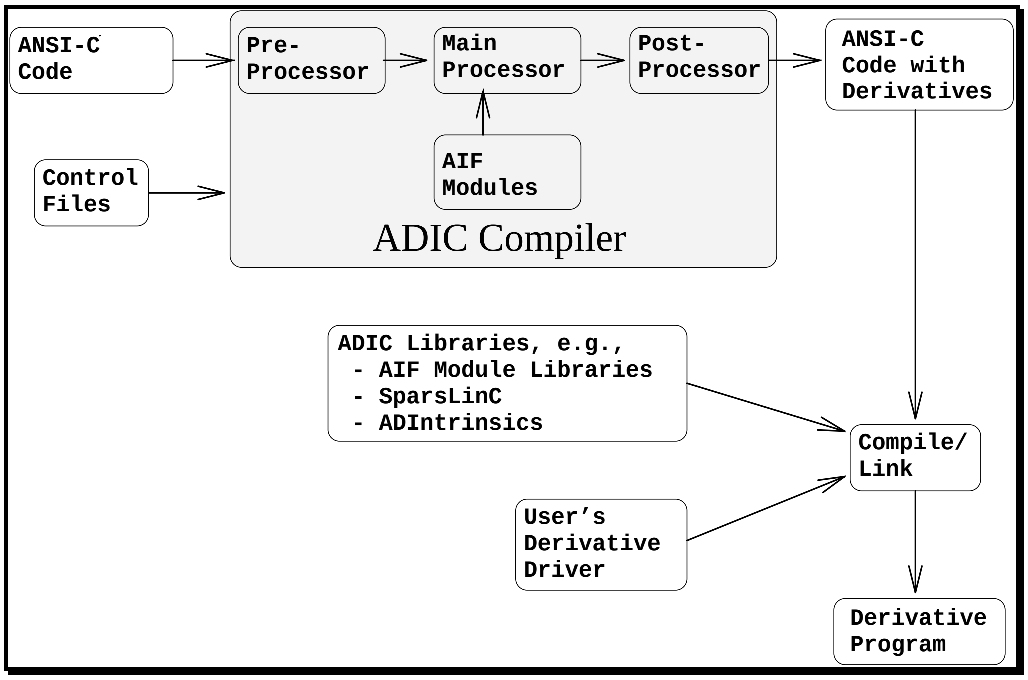

Figure 1: Generating derivative code with ADIC

Figure 1 illustrates the process employed to generate derivative codes with ADIC. The user submits a set of C source files along with an optional control script to ADIC. The control script is used to fine-tune the behavior of ADIC; for example, we can specify that a particular function should not be differentiated or the naming scheme of the generated functions. ADIC goes through a number of stages to generate the derivative code. For each source file, ADIC generates a new file representing its differentiated version. The generated sources are linked with ADIC runtime libraries and a driver written by the user. The ADIntrinsics library, for example, addresses potential nondifferentiability in intrinsic functions such as *sqrt(0)*, whereas SparsLinC provides support for sparse derivatives as they occur, for example, in large-scale optimization.

A typical driver has three main tasks: (1) It specifies and initializes the input variables with respect to which derivatives actually must be computed. In fact, with proper initialization, we can compute *directional* derivatives (this process is termed "derivative seeding'' [3,7]). (2) The driver then calls the derivative function to compute the derivatives. (3) The driver passes the derivatives to another program fragment that makes use of the derivative values.

To illustrate the transformation process, we consider the program fragment consisting of statements 1 and 2 in Section 2. From this fragment, ADIC generates the following code:

```
ad_loc_0 = DERIV_val(x[1]) + DERIV_val(x[2]);      (3)
ad_grad_axpy_2(DERIV_grad(a), 1.0,DERIV_grad(x[1]),(4)
               1.0, DERIV_grad(x[2]));
DERIV_val(a) = ad_loc_0;                           (5)

ad_loc_0 = DERIV_val(a) / DERIV_val(x[2]);         (6)
ad_adj_0 =  - ad_loc_0 / DERIV_val(x[2]);          (7)
ad_adj_1 = 1.0 / DERIV_val(x[2]);                  (8)
ad_grad_axpy_2(DERIV_grad(y[0]), ad_adj_1,         (9)
               DERIV_grad(a), ad_adj_0,
               DERIV_grad(x[2]));
DERIV_val(y[0]) = ad_loc_0;                       (10)
```

Statements 3-5 represent the differentiated version of statement 1; statements 6-10 represent the differentiated version of statement 2. ADIC redeclares floating-point variables and type declarations to be of type DERIV_TYPE:

```
typedef struct {
    double val;
    double grad[ad_GRAD_MAX];
} DERIV_TYPE;
#define DERIV_val(x) (x.val)
#define DERIV_grad(x) (x.grad)
```

DERIV_val is a macro that denotes the original floating-point value for each variable of type DERIV_TYPE. DERIV_grad is a macro that denotes the vector of total derivatives with respect to the chosen independent variables that is associated with an original floating-point variable. These macros and other are defined in a header file automatically generated by ADIC. Different definitions of these macros may be generated by ADIC depending on the mode of operation. Currently, ADIC associates derivative objects with all floating-point variables, unless otherwise specified by the user in a control script. The maximum length of the derivative object (ad_GRAD_MAX) is equal to the number of design parameters considered in the formulation.

4.2. Issues with FCAP Processing

We would like to run the entire source through ADIC and generate a new program that computes and displays not only the original results but also their derivatives with respect to various parameters. To make this possible, we modified the original yacc grammar to handle the specification of independent variables. The FCAP grammar reads the scripts written in the modeling language, parses it, then builds the appropriate data structures used in the simulation. The script contains many “param” variables that represent the design parameters (e.g., thickness of a plane) or some functions of other design parameters. To compute derivatives with respect to such a variable, the user need only change the “param” keyword to an “indepen” keyword. The modified grammar calls the appropriate derivative initialization routine automatically. In addition, we provided utilities for printing and extracting the derivatives. These modifications affected 2 out of the 30 FCAP3 input files. These files are then processed with ADIC, and the resulting output files linked together to produce the FCAP3.AD program.

4.3. Postoptimization

To explore the benefits of using smarter AD algorithms and better analysis capability, we optimized the use of ADIC by hand. The following two approaches were used.

- Identification of functions whose output does not impact the final result. For example, the results of certain test functions may affect only control flow. Once the user has identified such functions, they can be specified in the control script so that ADIC will not augment derivative computations for those functions.
- Derivative optimization for a section of the iterative linear solver for the Poisson equation. Employing a technique illustrated in [5], we applied the reverse mode over an entire code section by hand. The resulting code was then substituted for the ADIC-generated piece. Since ADIC employs a consistent naming and calling scheme, it is easy to provide optimized derivative codes for performance-critical sections.

5 Statistical Modeling Using FCAP2.AD and FCAP3.AD

Statistically based worst-case modeling of devices has been extensively studied [13], but little work has been done for on-chip statistical interconnect modeling [20]. Our methodology addresses the problem of quantifying the impact of process-induced interconnect variations on resistance (R) and capacitance (C) and circuit performance. In deep-submicron technologies, the on-chip interconnect delay can easily be more than 70% of the total delay. Furthermore, new planarization processes such as chemical-mechanical polishing (CMP) can cause variations of over 40% in the interlayer dielectric

(ILD) thickness, which has a large impact on the variation of interconnect capacitances. Variations in interconnect R and C cause variations in the circuit delay and crosstalk. If the circuit delay or crosstalk exceeds the specification of a critical circuit path, the chip will have slower performance or fail completely. Therefore, it is important to get an accurate estimate of the circuit delay and crosstalk spread for performance and yield tuning. In this application of FCAP2.AD and FCAP3.AD, we developed a novel methodology for obtaining statistically based worst-case (i.e., 3-sigma) R (resistance), C (capacitance), crosstalk, and delay given variations in interconnect-related process parameters [11,12].

This methodology is divided into three phases. In the first phase, the 3-sigma values of capacitance, resistance, and partial derivatives of capacitances with respect to selected interconnect process parameters are generated in batch-mode computation as part of an enhanced version of HIVE [10], which is a parameterized interconnect model generator and library for R and C, and the Derivative HIVE Generator. Most of time is spent in derivative calculation in this phase. In the second phase, randomized but correlated R and C are generated via a Monte Carlo method in a distributed N-Pi network for a given net via the randomized RC generator. In the last phase, the randomized RC net and nominal/3-sigma device models can be combined to characterize delay or crosstalk variation based on device and interconnect variations.

Using this methodology for a long critical net analysis on a 0.35 um process, we realized a more than 70% improvement in 3-delay delay estimation compared with the traditional skew-corner worst case delay. The 3-sigma crosstalk calculation for coupling nets can also be calculated in the similar manner.

6 Results

In this section we present the computational requirements of FCAP3.AD and compare them with the divided-differences technique. A particular run of the statistical modeling methodology with FCAP2 takes about 5-10 days of CPU time on an HP9000/755 workstation. The computational complexity increases by an order of magnitude when FCAP3 is employed. Most of this time is spent in computing derivatives; therefore, any method that reduces the derivative computation cost is significant.

For our experiments, we use two input models that compute (1) capacitances of two layers of 5-trace signals routed orthogonally between two ground layers, and (2) potentials of two vertically parallel signals routed between two ground layers. The experiments were performed on a Hewlett Packard 9000/780 workstation running HP-UX 10.20 and compiled using the Softbench C compiler with full optimizations.

Table 1 shows the runtime performance using divided-difference approximations versus ADIC “out of the box” and postoptimized derivative code for FCAP3. Central differences, which, unlike one-sided differences, usually deliver acceptable derivative approximations for FCAP2/3, would have required 2p+1 function evaluations to compute p derivatives plus the function values. Comparing the derivative values computed via central differences and automatic differentiation, we found that these values agreed to within one half of a percent.

The measurements are made for 2, 5, and 10 independent variables. The columns **AD/Func.** represent the runtime ratio of FCAP3.AD over FCAP3. The columns **DD/AD** represent the runtime ratio of using central divided difference approximations versus FCAP3.AD. We see that derivative code generated by ADIC out of the box (AD) is 1.2 to 2.0 times faster than the divided-difference method (DD); and as the number of independent variable increases, the speedup increases. This is as expected, since non-floating-point computations as well as certain derivative preaccumulation computations are amortized over larger number of derivative computations [3,7]. In the case of postoptimized FCAP3.AD, the results are an additional factor of 1.7 to 2.4 times faster than ADIC out of the box. Hence, the postoptimization steps can significantly improve the runtime performance. In either case, since the statistical modeling of on-chip interconnect properties is dominated by the cost of computing derivatives, considerable improvements are realized in the overall modeling process.

Since ADIC currently augments all floating-point variables with an array for the gradient object, memory requirements of the ADIC-generated code scale linearly with the number of independent variables.

# of Indep. Variables	**2**		**5**		**10**	
Model 1 **Function = 63.6 sec.**	**AD/Func.**	**DD/AD**	**AD/Func.**	**DD/AD**	**AD/Func.**	**DD/AD**
AD	**4.08**	**1.23**	**6.92**	**1.59**	**12.1**	**1.74**
AD Post-optimized	**2.10**	**2.38**	**3.52**	**3.13**	**4.93**	**4.26**
Model 2 **Function = 31.3 sec.**	**AD/Func.**	**DD/AD**	**AD/Func.**	**DD/AD**	**AD/Func.**	**DD/AD**
AD	**3.58**	**1.40**	**6.28**	**1.75**	**10.4**	**2.02**
AD Post-optimized	**2.08**	**2.40**	**3.71**	**2.96**	**4.99**	**4.21**

Table 1. Comparison of FCAP3.AD versus central divided differences for two different input models and three different sets of independent variables (3, 5, and 7). AD/Func. represents the runtime ratio of FCAP3.AD over function evaluation (FCAP3). DD/AD represents the runtime ratio of central divided differences over FCAP3.AD.

6 Conclusions

FCAP2 and FCAP3 are 2-D and 3-D, respectively, simulators to measure the parasitic electrical effects of interconnects and devices. In the statistical modeling of interconnects, the evaluation of gradients of FCAP-generated results are required. Conventional techniques cannot be relied upon to deliver fast and accurate derivatives. Divided differences may not be accurate and are obtained slowly, symbolic approaches do not appear to be feasible, and hand coding of derivatives is impractical. In contrast, automatic differentiation can be used to obtain fast and accurate derivatives for functions defined by large codes.

In this paper, we have described how the ADIC (Automatic Differentiation in C) tool has been used to generate derivative codes for FCAP programs. This has been done with minimal changes to the original source code. The experiments show that ADIC-generated derivatives reduce dependence on grid variations compared with the central divided difference method that had been employed before, while at the same time executing up to twice as fast. By postoptimizing ADIC-generated derivative codes, (namely, identifying inactive functions and employing the reverse mode of differentiation across a code block of the linear solver), performance was improved by roughly another factor of two. Since the cost of interconnect modeling is dominated by derivative computation, these derivative improvements result in considerable speedup overall.

Automatic differentiation is a field in its infancy. Improvements in the complexity of AD-generated derivative codes are driven by using smarter ways to exploit the associativity of the chain rule of differential calculus, by exploiting mathematical insight concerning the algorithms governing the underlying program, and by improving the program analysis capabilities of AD tools. Automatic differentiation also can be generalized to derivatives of arbitrary order, and we have developed prototype second-order capabilities for ADIC and ADIFOR.

Acknowledgments

We thank E. Fetzer, B. Queen, A. Meyer, R. Meitzler, K. Rahmat, and W. Xie for stimulating discussions on statistical modeling of interconnects, and S. Y. Chiang and H. Stork for management support. The work of Bischof and Roh was supported by the Mathematical, Information, and Computational Sciences Division Subprogram of the Department of Energy, under contract W-31-109-Eng-38, and by Technical Service Agreement No. 85G50 with the Hewlett-Packard Corporation.

References

[1] M. Berz, C. Bischof, G. Corliss, and A. Griewank, “Computational Differentiation: Techniques, Applications, and Tools,” SIAM, Philadelphia, 1996.

[2] C. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland, “ADIFOR: Generating Derivative Codes from FORTRAN Programs,” Scientific Programming, 1(1):11-29, 1992.

[3] C. Bischof, A. Carle, P. Khademi, and A. Mauer, “ADIFOR 2.0: Automatic Differentiation of FORTRAN 77 Programs,” IEEE Computational Science & Engineering, 3(3):18-32, 1996.

[4] C. Bischof, G. Corliss, L. Green, A. Griewank, K. Haigler, and P. Newman, “Automatic Differentiation of Advanced CFD Codes for Multidisciplinary Design,” Journal on Computing Systems in Engineering, 3(5):625-638, 1993.

[5] C. Bischof and M. Haghighat, “On Hierarchical Differentiation,” [1], pp. 83-94, 1996.

[6] C. Bischof , W. Jones, A. Mauer, and J. Samareh, “Experiences with the Application of the ADIC Automatic Differentiation Tool to the CSCMDO 3-D Volume Grid Generation Code,” in Proceedings of the 34th AIAA Aerospace Sciences Meeting, AIAA Paper 96-0716, American Institute of Aeronautics and Astronomics, 1996.

[7] C. Bischof , L. Roh, A. Mauer, “ADIC: An Extensible Automatic Differentiation Tool for ANSI-C,” Argonne Preprint ANL/MCS-P626-1196, 1996. To appear, Software: Practice and Experience.

[8] C. Bischof, G. Whiffen, C. Shoemaker, A. Carle, and A. Ross, “Application of automatic differentiation to groundwater transport models.” in Computational Methods in Water Resources X, ed. Alexander Peters, Kluwer, Dordrecht, 173-182, 1994.

[9] K. Cham, S. Y. Oh, D. Chin, J. L. Moll, K. Lee, and P. V. Voorde, “Computer-Aided Design and VLSI Device Development,” second edition, Kluwer, 1988.

[10] K. J. Chang, N. Chang, Ken Lee, and Soo-Young Oh, “Parameterized SPICE Subcircuits for Multilevel Interconnect Modeling and Simulation,” IEEE Special Issue on Interconnect Modeling and Design, Dec. 1992.

[11] N. Chang, V. Kanevsky, O. S. Nakagawa, K. Rahmat, and S. Y. Oh, “Fast Generation of Statistically Based Worst Case Modeling of On-Chip Interconnects,” International Conference on Computer Design, Oct. 1997.

[12] N. Chang, V. Kanevsky, B. Queen, O. S. Nakagawa, K. Rahmat, and S. Y. Oh, “3-sigma Worst-case Calculation of Delay and Crosstalk for Critical Nets,” Timing Specification Workshop, Dec. 1997.

[13] J. Chen, C. Hu, D. Wan, P. Bendix, and A. Kapoor, “E-T Based Statistical Device Modeling and Compact Statistical Circuit Design Methodologies,” University of California, Berkeley, and LSI Logic Corporation, Milpitas, California, *IEDM* Dec. 1996.

[14] P. Eberhard and C. Bischof, “Automatic Differentiation of Numerical Integration Algorithms,” Argonne Preprint ANL/MCS-P621-1196, 1996.

[15] D. S. Kershaw, “The Incomplete Cholesky - Conjugate Gradient Method for the Iterative Solution of Systems of Linear Equations,” Journal of Computational Physics, 26:43-65, 1978.

[16] P. E. Gill, W. Murray, and M. H. Wright, “Practical Optimization,” Academic Press, New York, 1981.

[17] A. Griewank, “On Automatic Differentiation.” Mathematical Programming: Recent Developments and Applications, ed. A. L. Norwell, Kluwer, pp. 83-108, 1989.

[18] A Griewank and G. F. Corliss, “Automatic Differentiation of Algorithms: Theory, Implementation, and Application,” SIAM, Philadelphia, 1991.

[19] A. Griewank, D. Juedes, and J. Utke, “ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++,” ACM Transactions on Mathematical Software, 22(2), 131-167, 1996.

[20] O. S. Nakagawa, K. Rahmat, N. Chang, S. Y. Oh, P. Nikkel, and D. Crook, “Impact of CMP ILD Thickness Variation on Interconnect Capacitance and Circuit Performance,” in Proceedings of Second International Chemical-Mechanical Polish for ULSI Multilevel Interconnection Conference (CMP-MIC), p. 251, Feb., 1997

[21] N. Rostaing, S. Dalmas, and A. Galligo, “Automatic Differentiation in ODYSSEE,” Tellus, 45a(4), 558-568, 1993.

[22] D. Shiriaev and A, Griewank, “ADOL-F: Automatic Differentiation of Fortran codes,” in [1], pp. 375-384, 1996.