COSMOS:
A Compiled Simulator for MOS Circuits*

Randal E. Bryant
Derek Beatty†
Karl Brace
Kyeongsoon Cho
Thomas Sheffler
Carnegie Mellon University

Abstract

The COSMOS simulator provides fast and accurate switch-level modeling of MOS digital circuits. It attains high performance by preprocessing the transistor network into a functionally equivalent Boolean representation. This description, produced by the symbolic analyzer ANAMOS, captures all aspects of switch-level networks including bidirectional transistors, stored charge, different signal strengths, and indeterminate (X) logic values. The LGCC program translates the Boolean representation into a set of machine language evaluation procedures and initialized data structures. These procedures and data structures are compiled along with code implementing the simulation kernel and user interface to produce the simulation program. The simulation program runs an order of magnitude faster than our previous simulator MOSSIM II.

1 Overview

COSMOS (COmpiled Simulator for MOS circuits) provides a new level of performance in switch-level simulation. While providing the same network model and functionality as MOSSIM II [2], COSMOS operates an order of magnitude faster.

Unlike programs that operate directly on the transistor level description during simulation, COSMOS preprocesses the transistor network into a Boolean description. This description, produced by the symbolic analyzer ANAMOS, captures all aspects of switch-level networks including bidirectional transistors, stored charge, different signal strengths, and indeterminate (X) logic values.

COSMOS consists of a set of C language programs configured as shown in Figure 1. ANAMOS reads in the switch-level representation of a MOS circuit and partitions it into a set of channel-connected subnetworks. It then derives a Boolean representation of the behavior of each subnetwork. A second program, LGCC, translates the Boolean representation into a set of C language evaluation procedures plus declarations of data structures describing the network interconnections. Finally, the code produced by LGCC, together with the simulation kernel and user interface code, are compiled to generate the simulation program. The resulting program appears to the user much like an ordinary simulator, except that the network is already loaded at the start of execution. The simulator implements a block-level, event-driven scheduler, with blocks corresponding to subnetworks. Processing an event involves calling the appropriate procedure to compute the new state and output of a block. Future enhancements include a translator, SIMBLK, from a behavioral modeling language into the same Boolean representation produced by ANAMOS, allowing mixed behavioral and switch simulation.

The ANAMOS output could also be mapped onto any of a number of hardware simulation accelerators, since only simple, Boolean evaluation is required. Other applications, such as fault simulation, test generation, and symbolic verification, could also utilize the Boolean representation of a switch-level network produced by ANAMOS.

*This research is supported in part by the Defense Advanced Research Projects Agency, ARPA Order Number 4976, and in part by the Semiconductor Research Corporation under Contract 86-01-068.

†Supported by a National Science Foundation Graduate Fellowship

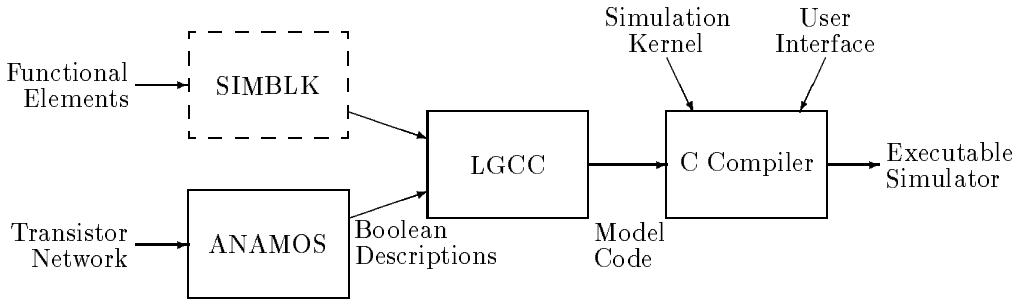

Figure 1: COSMOS Program Structure

2 Related work

Other researchers have developed symbolic analyzers for MOS circuits [5, 6, 7, 11], but these programs have poor worst case performance (exponential in the circuit size), as well as modeling limitations, especially with respect to indeterminate values and attenuating transistors. The resulting symbolic representations produced by these programs are often too large for efficient evaluation and lead to unreliable simulation results. In addition, most rely on algebras that extend standard Boolean algebra, or that involve a combination of integer and Boolean operations. Some of these algebras cannot be mapped onto existing hardware accelerators.

All analysis programs (including ANAMOS) partition a network into subnetworks prior to analysis. One can argue that even an exponential algorithm may have acceptable performance for the small subnetworks typically found. However, we have encountered subnetworks with over 5000 transistors. A practical symbolic analyzer must also have good worst case performance.

The SLS program developed at IBM [1] also preprocessors a switch-level network to generate compiled evaluation procedures for each subnetwork. The evaluation procedure implements one iteration in a strength propagation algorithm similar to that used by MOSSIM II [2]. The simulator evaluates a subnetwork by calling the appropriate procedure repeatedly until convergence. Heuristic methods order the nodes so that most subnetworks require only a small number of iterations to converge [9]. In the worst case, however, SLS requires a number of iterations proportional to the number of subnetwork nodes. Furthermore, this technique cannot be mapped onto existing simulation accelerators.

In contrast to its predecessors, ANAMOS guarantees polynomial performance while providing a more detailed and accurate model. In fact, it produces a description linear in the circuit size for the MOS circuits encountered in actual designs, with the exception of certain dense pass transistor networks. Furthermore, it requires nothing more than conventional Boolean algebra to express circuit behavior.

3 Network Model

ANAMOS accepts as input a list of parameterized nodes and transistors. The topology of a switch-level network matches that of the actual circuit, but circuit operation is modeled in a highly idealized way.

Nodes of type *input* represent strong signal sources originating off-chip. The user can designate an input node as a constant source of 1 or 0 to represent a power supply connection. The remaining nodes are of type *storage*, able to store charge dynamically in the absence of other signal connections. Charge sharing is modeled in a simplified way by assigning each storage node a discrete *size*, denoted by the integer values $\{1, \ldots, k\}$. The state resulting when a set of nodes share charge is determined by the initial states of the largest node(s). An input node is indicated by size $w > k$. Node voltages are represented by the discrete states 0, 1, and X, where X indicates an indeterminate or invalid logic level.

A transistor has terminals labeled "gate", "source", and "drain". It acts as a resistive switch connecting the source and drain nodes controlled by the state of the gate node. All transistors are modeled as bidirectional devices with no predetermined direction of information flow. A transistor has a *type* (n-type, p-type, or depletion) to represent the devices found in both NMOS and CMOS circuits. Ratioed circuits are modeled in a simplified way by assigning each transistor a discrete *strength*, denoted by integer values $\{k+1, \ldots, w-1\}$. During circuit operation, the con-

ducting transistors in a network form paths between the nodes, where each path has strength equal to that of its weakest transistor. The state on a node resulting due to paths from inputs nodes depends on the state(s) of those input nodes connected by maximum strength paths.

The overall ranking of sizes and strengths indicates a signal precedence with input nodes highest, paths from input nodes through conducting transistors next, and stored charge least. Few circuits require more than a total of $w = 6$ signal strengths to describe their operation.

Circuit operation is modeled according to a simple, unit-delay timing model. The behavior of a subnetwork is characterized by its *steady state response*, giving a mapping from initial input and storage node states to new storage node states. Informally, this response can be viewed as the set of logic states that would form on the storage nodes if the transistors were held fixed according to the initial states of their gate nodes. The simulator operates by repeatedly computing the steady state response and setting the nodes to their responses until a stable state is reached.

The abstractions made by our switch-level model greatly facilitate the task of representing the circuit behavior symbolically. Discrete node states allow the circuit behavior to be expressed in a simple, discrete algebra. Discrete signal strengths and their strict precedence allows the circuit behavior to be derived by computing the behavior for each strength level from strongest to weakest. The simplified timing model eliminates the need to capture circuit delays in the symbolic description.

4 Symbolic Analysis

The most novel aspects of COSMOS are found in the symbolic analyzer ANAMOS. Space precludes a complete description of the algorithms. Other reports [3, 4] give a detailed presentation.

4.1 Network Partitioning

ANAMOS partitions the network into channel-connected subnetworks and separately derives the steady-state response of each subnetwork. Each subnetwork corresponds to a component in the undirected graph having as vertices the storage nodes and as edges the pairs of nodes connected by transistor sources and drains. This partitioning describes the static connections in the network, i.e., those independent of transistor state.

Performing the partitioning during preprocessing leads to a simulator with static partitioning, less able to exploit locality than simulators employing dynamic partitioning [2]. We have not yet determined the degree to which this reduced locality penalizes simulation performance. Should it prove significant, we could break the Boolean descriptions of the subnetworks into smaller pieces and evaluate them by event-driven scheduling.

Within a subnetwork, the behavior can be complex and difficult to analyze due to the bidirectional transistors and the many ways state is formed in a MOS circuit. The interactions between subnetworks, however, are much more straightforward. Each subnetwork acts as a sequential logic element having as inputs the input nodes connected to transistor sources and drains, plus the gate nodes of the transistors. The subnetwork state is stored as charge on the storage nodes. Its outputs are those nodes that are gate nodes of transistors in other subnetworks. Hence, the overall operation of our simulator is similar to that of a logic gate simulator—changing values on the subnetwork inputs require updating the state and outputs, and these changing output values in turn affect other subnetworks.

4.2 State Encoding

To cast the switch-level model in terms of Boolean operations, a logic value $y \in \{0, 1, X\}$ is represented by a "dual rail" Boolean encoding, $y.1, y.0 \in \{0, 1\}$ as follows

y	$y.1$	$y.0$
1	1	0
0	0	1
X	1	1

With a Boolean encoding of the state values, the symbolic analysis problem can be defined as follows. For each node `n`, introduce Boolean variables $n.1$ and $n.0$ to represent the possible encoded node state values. Then, for each node `n`, derive Boolean formulas $N.1$ and $N.0$ to represent the encoded value of the steady state response at the node as a function of the initial node states. Through careful formulation, the analyzer can produce formulas that fully characterize the three-valued logic behavior of the circuit while retaining the simplicity of Boolean algebra.

4.3 Systems of Boolean Equations

Switch-level networks resemble classical contact networks in that both consist of bidirectional switching elements. Shannon [8] first developed techniques for analyzing a contact network symbolically. His method labels each contact with a Boolean literal

and formulates the conditions under which a path may form between designated pairs of terminals as a Boolean expression. This idea serves as the conceptual basis of ANAMOS, although MOS circuits require a more complex analysis method. Furthermore, most of the methods presented in the contact network literature do not lend well to computerized application on large circuits.

The contact network analysis problem can be formulated as the solution of a system of Boolean equations [3]. A generalized form of Gaussian elimination can solve such a system of equations, with Boolean operations $\wedge$ and $\vee$ replacing the conventional arithmetic operations [10].

Gaussian elimination has a distinct advantage over iterative methods for symbolic analysis. Being a direct method, it requires no testing for convergence. Symbolic analysis can proceed by simply constructing Boolean formulas in terms of operations $\wedge$ and $\vee$ in accordance to the elimination steps. A direct method avoids the need to test formulas for equivalence, an NP-hard problem.

The efficiency of Gaussian elimination depends on the sparseness of the graph and the total number of fill-in edges added during elimination. For solving a system of Boolean equations, pivots can be selected in an order that keeps fill-in low, without concern for numerical accuracy. Experience indicates that most graphs arising from MOS circuit analysis can be expressed as acyclic connections of series-parallel graphs. Gaussian elimination requires only a linear number of operations for such graphs. By always eliminating a vertex of minimum degree, no vertex has degree greater than 2 upon elimination. Even many non-series-parallel circuit graphs can be solved with a linear number of operations. However, certain pass transistor networks have very dense circuit graphs. For example, a pass transistor barrel shifter forms a complete bipartite graph. For these, Gaussian elimination achieves its $O(N^3)$ worst-case complexity for an N node subnetwork.

4.4 Application to MOS Circuits

Switch-level networks require a more complex analysis than contact networks. Signals have different strengths due to the varying node sizes and transistor strengths. The steady state response on a node must be expressed as a pair of formulas to describe ternary network behavior. However, these problems can be dealt with by solving a series of systems of Boolean equations.

For each strength level s and node $\mathtt{n}$, formulas $N.1_s$ and $N.0_s$ are derived describing the effects by paths of strength greater than or equal to s. These formulas arise as the solution of two systems of Boolean equations, where the initial vertex and edge labels contain terms representing the following effects: paths of strength greater than s (given by formulas $N.1_{s+1}$ and $N.0_{s+1}$), transistors of strength s (for $s > k$), nodes of size s (for $s \leq k$), and formulas representing the conditions under which no blocking path of strength greater than s is present. This latter term is given for each node $\mathtt{n}$ by a formula $N.c_{s+1}$, computed by solving a third system of equations at strength $s+1$.

Thus, switch-level network analysis involves solving 3 systems of equations at each strength level starting from $w-1$ and working downward. The final result for each node $\mathtt{n}$ is given by formulas $N.1_1$ and $N.0_1$. For typical circuits where w ranges from 3 to 6, this involves setting up and solving between 8 and 14 systems (formula $N.c_1$ is not needed). Hence, the complications of switch-level networks add only a constant complexity factor to the contact network analysis problem.

4.5 Boolean Manipulation

ANAMOS represents a Boolean formula as a directed acyclic graph (DAG). A DAG resembles a parse tree with leaves representing variables and constants and with internal vertices representing Boolean operations. In a DAG, however, a given subgraph may be shared by several branches, yielding a more compact representation. A formula is given by the pointer to some DAG vertex, where the formula denoted consists of the vertex and all of its descendants. During the analysis of a subnetwork, ANAMOS constructs a single DAG to represent all formulas for the subnetwork. Figure 3 shows an example of such a DAG.

A Boolean operation can be applied to two formulas to yield a new formula by simply adding a new vertex to the DAG with branches to the argument vertices. The total number of vertices produced by this method cannot grow larger than the total number of algebraic operations required to set up and solve the systems of Boolean equations. No other data structure for representing Boolean functions matches this efficiency.

As an optimization, ANAMOS attempts to simplify each formula as it is created, using techniques developed for optimizing compilers. It only applies simplifications that maintain or decrease the number of vertices, preserving the upper bound on DAG size. After applying the simplification rules, ANAMOS looks for an existing vertex with the desired operation and arguments. Only if the match fails does the program

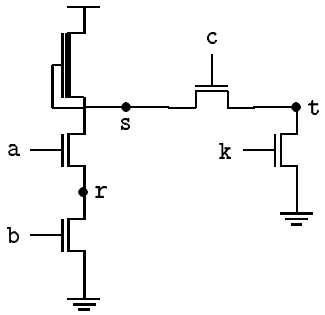

Figure 2: Example Network

create a new vertex, thereby avoiding the creation of common subexpressions.

Most simplifications involve straightforward graph transformations, implementing such rules as constant propagation and elimination, associativity, and commutativity. A test to detect redundant terms, however, involves (non-combinatorial) search applied to the descendants of the argument vertices. Consequently, the total time spent in simplification can grow worse than linearly with the number of operations. To control its computational effort, the program can limit the maximum depth to which the searches for redundancies may proceed before giving up.

Measurements show that ANAMOS expends most of its computational effort on Boolean manipulation, particularly on redundancy testing. As in an optimizing compiler, the program must trade off the time spent for simplification versus the size of the resulting DAG's. Smaller DAG's yield more compact and consequently faster simulation code. The appropriate balance depends on the extent to which the simulation code will be run. Therefore ANAMOS lets the user control the degree of optimization.

4.6 Interpretation of Results

Figure 2 shows an NMOS circuit consisting of a single subnetwork. This circuit provides an interesting case for symbolic analysis, because the transistor controlled by `c` operates bidirectionally, either passing the NAND gate output to node `t`, or as part of a pulldown chain for node `s`. ANAMOS produces the DAG shown in Figure 3, assuming all storage nodes have size 1, depletion transistors strength 2, and enhancement transistors strength 3.

Examining this DAG highlights several properties of ANAMOS. The leaves represent Boolean encodings of the control signals `a`, `b`, `c`, and `k`, as well as the initial charge on node `t`. The roots $S.0$ and $S.1$ indicate formulas for the steady state response on node `s`, while $T.0$ and $T.1$ indicate formulas for node `t`. Some terms are shared by several formulas in the DAG. More complex formulas typically share many terms, enhancing the efficiency of the analyzer.

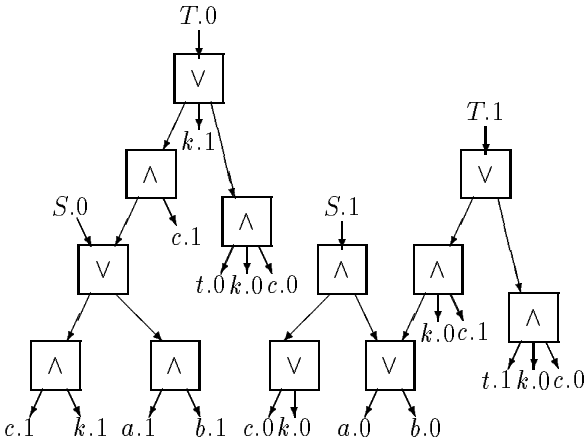

Figure 3: DAG Representation of Example Network

Observe that node `r` is not represented in the DAG. During an optimization phase, ANAMOS determines that the formulas for nodes `s` and `t` contain no reference to node `r`. That is, they have no leaves of the form $r.0$ or $r.1$, indicating that the stored charge on `r` can never affect the values on `s` or `t`. Since `s` and `t` are the only nodes observable from outside the subnetwork, the formulas for node `r` can be pruned away. In general, the optimizer can eliminate all nodes in the pullup or pulldown chains of standard NMOS and CMOS logic gates by this method.

In Figure 3 the formula $T.0$ is formed as the OR of 3 terms. Studying these terms separately gives insight into how the formulas produced by ANAMOS describe circuit operation. Each represents a separate source of logic level 0 to node `t`. The left hand term describes the case where the gate output is pulled low (given by the formula $S.0$) and the pass transistor is conducting (indicated by $c.1$). The center term represents a direct connection to ground through the transistor with gate `k`. The final term represents the case where `t` is isolated ($c.0 \wedge k.0$) and hence retains its stored charge ($t.0$). A similar analysis can be applied to the other 3 formulas. Perhaps less obvious is the fact that these formulas also characterize the subnetwork behavior when some nodes have state X. Much of the apparent complexity of the formulas arises from the fact that they hold for all 243 ternary combinations of input and storage node states, not just the 32 Boolean combinations.

5 Compilation

Compiling the Boolean description of a logic circuit into simulation code presents no major difficulties. However, many details must be resolved about data structures, calling conventions, and efficient coding.

5.1 LGCC

The LGCC program converts the Boolean descriptions produced by ANAMOS into executable C code. It has as input a hierarchical representation describing the behavior of subnetworks in the leaves of the hierarchy, and the interconnections among them in the root of the hierarchy. LGCC in turn produces C source code containing procedures describing the behavior of each subnetwork, and a set of initialized data structures describing the interconnection of subnetworks.

The hierarchical Boolean descriptions are given in a notation, *lgc*, similar to a programming language. An *lgc* file first describes subnetworks, and then gives their interconnection. Dual-rail encoding is explicit in the *lgc* representation. That is, each node is described by two Boolean variables, called *simulation variables*. Subnetworks are described by *modules*, looking much like procedures with formal parameters corresponding to the simulation variables of the input, output, and storage nodes of the subnetwork. The interconnection of subnetworks looks much like a program that declares variables, then instantiates these procedures.

The descriptions of subnetwork behavior form the lowest level in the hierarchy. Each subnetwork is described by the DAG resulting from its analysis. A DAG is listed in the file according to a topological sort of its nodes, where each node is listed as its operation and arguments.

The declaration of subnetwork interconnections forms the highest level in the hierarchy. This portion of the *lgc* file begins with the declaration of the network nodes that have not been eliminated during analysis of their containing subnetworks. It also contains a set of *module instances*, which map the formal parameters of the modules onto the declared nodes.

The transformation of a subnetwork description into a C procedure is straightforward, complicated only by the need to store temporary values. Since the DAG nodes in the *lgc* file are already ordered for evaluation, most transformations are merely syntactic. The generated procedures use only a few features of the C language. Each procedure takes two arguments, which are pointers through which the procedure accesses the formal parameters of the original *lgc* module. The only operations required in a procedure are pointer dereferencing, array indexing, assignment, and, of course, Boolean operations.

The initialized data structures represent the overall network structure. These data structures define the circuit nodes, their membership in subnetworks, and their controlling effects on other subnetworks. Their key features are the node array and the module instance array, which refer to each other. In addition to the node array and module instance array, LGCC emits array declarations which allocate (at compile time) storage for the simulation kernel's event lists. As a consequence, the simulator requires very little dynamic allocation during execution.

Each entry in the node array declares a node with fields indicating its name and two simulation variables (for dual-rail encoding of node state). A simulation variable is represented by its old and new values, and its fanout list. The old and new values are Boolean values used to implement a strict unit-delay timing model. The fanout list is a sequence of references to the module instances which are affected when the value of the variable changes. Each entry in the module instance array declares a subnetwork instance. The fields for an instance indicate the procedure describing subnetwork behavior, the lists of state and input variables, and flags used by the simulation kernel's event scheduler.

5.2 Simulation Kernel

The simulated system appears to the kernel as a set of Boolean state variables connected by procedural modules. Its design does not depend on the correspondence between pairs of variables and circuit nodes nor between module instances and subnetworks.

The kernel implements the simulation of a *phase* as the basic simulator operation. During a phase, the program holds all data and clock inputs fixed and simulates unit steps until either it reaches a stable state or exceeds a user-specified step limit. Each unit step consumes one event list and produces another, where the initial event list indicates any new values on input nodes. The program makes one pass through the event list, calling module procedures to compute new values of the module output variables. It then makes a second pass to update the state variables and schedule all modules affected by the changing variables. Two passes are required to implement a strict, unit-delay model. The kernel requires only two event lists at any time, neither of which can be larger than the number of modules in the network.

In comparison to MOSSIM II, the COSMOS simulation code is far simpler and more modular. ANAMOS handles the complexities of switch-level analysis.

LGCC produces most of the data structures required during simulation. Hence, the kernel need only handle the tasks of module evaluation, event scheduling, and state variable updating. However, having reduced the computation required for network evaluation, the kernel becomes a critical factor in the simulator performance. At one point during the development, we doubled the simulation speed by careful crafting of the kernel code.

5.3 User Interface

The current, line-oriented, user interface operates in either interactive or batch mode. The user starts a simulation session starts by defining a clocking scheme and declaring which nodes are to be observed during simulation. He or she then gives commands to simulate the network for a number of clock cycles, to probe or set the state of any node in the network, and to save and restore the network state.

Generating an executable simulator for a circuit requires either compiling or linking the user interface code. This provides a natural opportunity for a user to customize the simulator by linking different interface or kernel code. By this means, a user can adapt the simulator to reflect the particular circuit characteristics, simulation requirements, and personal preferences. We organized the interface code to facilitate the inclusion of new command procedures. We plan to implement one or more alternate interfaces to accommodate different user preferences. One styled after a programming language interpreter seems especially desirable, as it would provide powerful capabilities for further customizing the simulation environment.

6 Experimental Results

COSMOS became operational in September, 1986. We have simulated a variety of small circuits to gain a better understanding of the programs, particularly of ANAMOS. We continue to refine the programs with a goal of developing a simulator suitable for true VLSI circuits.

Figure 4 compares the performance of COSMOS and MOSSIM II. All times were measured on a Digital Equipment Corporation MicroVax-II. A 64-bit NMOS ALU operated as a counter serves as the benchmark. This circuit contains several nontrivial MOS circuit structures, including a precharged Manchester carry chain, and pass transistor multiplexors. Its modest size (1664 transistors) hides the degree to which it stresses simulation performance. Since every bit position of the ALU is precharged every clock cycle, each cycle involves many events.

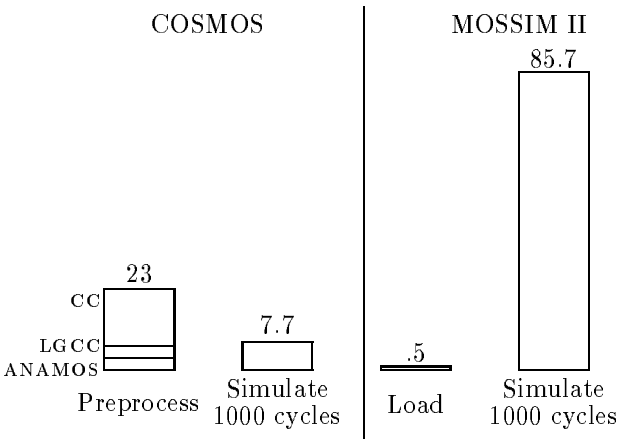

Figure 4: COSMOS vs. MOSSIM II Performance Comparison. Time measured in Microvax-II CPU minutes.

Comparing the simulation speed to that of MOSSIM II, we see that COSMOS operates 11.7 times faster. However, substantial time is required to create the compiled network description. Whereas MOSSIM II requires only 30 seconds to read in the network and set up the simulation data structures, a total of 23 minutes is required by the 3 programs ANAMOS, LGCC, and the C compiler CC. This time can be tolerated if the ensuing simulation run involves many patterns, but not for short simulation runs. Furthermore, it indicates that our programs are not yet ready for very large circuits.

Of this preprocessing time, the C compiler consumes the majority (70%). LGCC could be modified to generate assembly code directly without much difficulty. This, however, would limit program portability, and hence we have chosen not to take this step yet.

7 Future Work

COSMOS has already demonstrated the feasibility of compiled switch-level simulation, achieving far better simulation performance than its more traditional counterparts. Much work remains, however, to improve preprocessing speed, and to exploit the full potential of symbolic analysis.

As the performance data indicates, we must reduce the time to compile a circuit into simulation code. We plan to improve the performance through *incremental analysis*. Each time ANAMOS identifies a sub-

network, it will compare the subnetwork structure to those already analyzed. This comparison can be performed quickly by generating a hash signature for the subnetwork through wirelist comparison techniques, comparing only those subnetworks with matching signatures. For each unique subnetwork, files will be generated containing the circuit structure, the DAG representation, and both source and object code for the evaluation procedure. For circuits with many repeated structures, the time savings in the 3 ensuing preprocessing steps will greatly outweigh the added cost of comparison. The subnetwork will also be compared to subnetworks analyzed during previous executions of ANAMOS. As a result, rerunning the preprocessor on a circuit following a design modification will mostly involve analyzing the modified portions. Through this technique COSMOS can be practical even in the early stages of a chip design despite frequent modifications and short simulation runs.

We are presently adding fault simulation to COSMOS. Initially, only stuck-at faults at the subnetwork boundaries will be allowed. In the longer term, ANAMOS will be extended to characterize the behavior of a circuit with more detailed fault models, including stuck-open and stuck-closed transistors. Our implementation combines both parallel and concurrent fault simulation techniques. Since the subnetwork evaluation procedures involve only machine-level Boolean operations, they can be used without modification for parallel fault simulation.

For execution on hardware accelerators, the ANAMOS and SIMBLK outputs can be mapped into sets of Boolean elements. Whatever methods are provided to accelerate logic gate simulation can then perform switch-level simulation. COSMOS requires no special hardware for switch-level simulation. In fact, many of the costly features found in existing simulation accelerators such as bidirectional elements and multi-valued logic modeling are not needed. Preprocessors such as ours encourage a "RISC" approach to hardware design that implement only basic operations on a limited set of data types. The preprocessor performs the mapping between the complex models required by the simulator and the simple operations implemented by the hardware. For switch-level simulation, the payoff in terms of greater flexibility and performance clearly favors this approach.

Acknowledgements

Gary York provided valuable assistance in implementing ANAMOS.

References

[1] Z. Barzilai, *et al*, "SLS—a Fast Switch Level Simulator for Verification and Fault Coverage Analysis", *23rd Design Automation Conf.*, ACM, 1986, pp. 164–170.

[2] R. E. Bryant, "A Switch-Level Model and Simulator for MOS Digital Systems," *IEEE Trans. on Computers* Vol. C-33, No. 2 (February, 1984), pp. 160–177.

[3] R. E. Bryant, "Algorithmic Aspects of Symbolic Switch Network Analysis", *IEEE Trans. on Computer-Aided Design of Integrated Circuits*, accepted for publication, 1987.

[4] R. E. Bryant, "Boolean Analysis of MOS Circuits", *IEEE Trans. on Computer-Aided Design of Integrated Circuits*, accepted for publication, 1987.

[5] E. Cerny, and J. Gecsei, "Simulation of MOS Circuits by Decision Diagrams", *IEEE Trans. on Computer-Aided Design of Integrated Circuits*, Vol. CAD-4, No. 4 (October, 1985), pp. 685–693.

[6] G. Ditlow, W. Donath, and A. Ruehli, "Logic Equations for MOSFET Circuits", *International Symposium on Circuits and Systems*, IEEE, 1983, pp. 752–755.

[7] I. N. Hajj, and D. Saab, "Symbolic Logic Simulation of MOS Circuits", *International Symposium on Circuits and Systems*, IEEE, 1983, pp. 246–249.

[8] C. E. Shannon, "A Symbolic Analysis of Relay and Switching Circuits", *Trans. of the AIEE*, Vol. 57 (1938), pp. 713–723.

[9] I. Spillinger, and G. M. Silberman, "Improving the Performance of a Switch-Level Simulator", *IEEE Trans. on Computer-Aided Design of Integrated Circuits*, Vol. CAD-5, No. 3 (July, 1986), pp. 685–693.

[10] R. E. Tarjan, "Fast Algorithms for Solving Path Problems", *J. ACM*, Vol. 23, No. 3 (July, 1981), pp. 594–614.

[11] C. J. Terman, *Simulation Tools for Digital LSI Design*, PhD Thesis, MIT Dept. Elec. Eng. and Comp. Sci., October, 1983.