Fault Grading of Large Digital Systems

Daniel G. Saab, Robert B. Mueller-Thuas, David Blaauw
Center for Reliable and High Performance Computing
Coordinated Science Laboratory
University of Illinois
Urbana, IL 61801

Joseph T. Rahneh and Jacob A. Abraham
Computer Engineering Research Center
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, TX 78712

ABSTRACT
Fault simulation is an integral part of developing high quality tests for integrated circuits. Due to its high time and memory requirements the size of systems that can be simulated cost-effectively is limited. This paper discusses a fault simulation approach that can be used to fault grade large digital designs on engineering workstations. The hierarchical approach reduces memory requirements drastically by storing the structure of common repeated subcircuits only once and allows flexible multilevel simulation. The simulation algorithms are at the switch level so that general MOS digital designs with bidirectional signal flow can be handled, and both stuck-at and transistor faults are treated accurately. Our fault simulation algorithms have been implemented as a prototype that was used to fault grade a model of the Motorola 68000 microprocessor on SUN Microsystems workstations.

1. Introduction
Integrated circuits are being fabricated with increased complexity allowing the implementation of large system on a single chip. This increase in the level of integration has exceeded the capability of current Computer Aided Design (CAD) tools which are crucial for the design and verification of large systems. In particular, the effort spent on simulation has grown sharply. Simulation tools are employed both to help verify the functionality of a design (logic simulation) and to evaluate the quality of a set of test patterns (fault simulation).

Fault simulation has traditionally been performed at the gate level [1] where failures are approximated using the stuck-at-fault model [2]. For Metal Oxide Semiconductor (MOS) technology this is an inappropiate level of abstraction. MOS circuits may contain both logic and pass transistors exhibiting bidirectional signal flow and charge sharing effects. Furthermore, the gate level stuck-at-fault model does not model realistic physical failures in MOS circuits adequately [3].

Switch-level fault simulators[4] are effective in simulating transistor level effects and failures; however, they tend to require large amounts of memory when the circuits are represented at the flat level and the circuit hierarchy is not utilized. This memory requirement becomes a limiting factor of the performance of switch level logic and fault simulators when large circuits are considered. The limitations are due to the severe performance penalties of paging in a virtual memory environment. Therefore, to effectively perform logic and fault simulation in a reasonable amount of time and with acceptable memory requirements, circuit regularity and hierarchy must be exploited.

From the point of view of simulation, a hierarchical circuit description offers many advantages over a flat description. A hierarchical description allows compact representation of the circuit by exposing

† This work was supported in part by the Semiconductor Research Corporation Contract 87-DP-109 at the University of Illinois, in part by the Semiconductor Research Corporation Contract 88-DJ-142 at the University of Texas, and in part by Motorola Inc., Austin, Texas.

repetitive† used blocks. Consider, for example, an N-bit binary adder; it can be n times the interconnection of N identical submodules each consisting of a full adder which, in turn, consists of an interconnection of elementary logic gates. This contrasts sharply with a flat description, where the whole N-bit adder is given at the gate level. The difference in the size of the description becomes even more visible when the lowest level of the circuit is at least in part described in terms of transistor networks, as is the case for MOS designs. This reduction in memory requirement is pivotal for dealing with large circuits, where paging activity during simulation degrades performance and makes analysis of a complete system practically impossible.

A hierarchical description can also be used to speed up the simulation. It facilitates the replacement of complex modules with functionality equivalent but computationally cheaper modules. For instance, a collection of gates can be emulated by one software function. Also, for blocks of moderate size, one has the option of constructing a table by carrying out an exhaustive low-level simulation or generating logic expressions [5-7], or generating a behavioral description [8] to replace the block and expedite its evaluation while maintaining its function.

Hierarchical logic and fault simulation was first implemented in the program CHIEFS [9]. CHIEFS uses a gate-level description at the lowest level, and thus cannot model transistor-level effects. Moreover, gate level simulation requires unidirectional signal flow across subcircuit boundaries. This is insufficient for MOS design. Similarly, multilevel simulation has typically been performed from the gate level upwards.

In this paper, an approach is given for hierarchical multilevel fault simulation. The approach is based on representing the circuit in a hierarchical fashion where the lowest level primitives consist of transistor interconnections. A key point of this work is in its application to large systems. The approach has been implemented as a prototype fault simulator, F_CHAMP, which has the following features:

(1) It is switch level based. Hence general MOS designs are handled. Transistor level stuck-open/stuck-close faults can be modeled in addition to the classical gate level faults.

(2) It allows mixed mode simulation: parts of the circuit can be simulated faster at a behavioral level by supplying a high level software description.

The simulator has been used to fault grade the M68000 [10] microprocessor design obtained from Motorola Inc., Austin. This is the first program that can perform fault simulation for large systems with reasonable requirements of CPU time and memory.

The remainder of the paper is organized as follows: Section 2 describes the data structures necessary for hierarchical simulation. Section 3 details the various stages of the simulation algorithm. Section 4 outlines our implementation and presents results and observations from our experiments. Section 5 offers conclusions and gives directions for future research.
2. Circuit description and data structures

2.1. Circuit description

The circuit is described in a hierarchical format as an interconnection of building blocks. A building block can be an interconnection of other building blocks or as an interconnection of primitives.

Two types of primitives are used in our framework: behavioral models (also called functional models) and transistor networks. A behavioral model explicitly specifies the input/output relationship of a piece of circuitry. The behavioral description is specified in a high-level software function (also referred to as C-function, since the C programming language is used for implementation); it is either generated automatically using compiled simulation techniques[8] or supplied by the user. A behavioral model may contain just a logic gate (for example using table lookup or bit manipulation functions), a collection of gates or a whole functional block (see for instance [8]). Note that a behavioral model can appear at any level of the hierarchy. In particular, a complete block that was previously described in terms of subcells can be replaced by a software function.

The second type of primitive consists of a network of transistors. A transistor network is given as a netlist of MOS transistors. A transistor is modeled as a three node device (source, gate, and drain). All transistors act as voltage-controlled switches which can be in one of three states: on (high conductance), off (open circuit), and undefined (on or off or intermediate). The state of the circuit may assume one of three values: high (1), low (0), or undefined (X). An nMOS (pMOS) transistor is on (off) when its gate is high, off (on) when its gate is low, and undefined when its gate is undefined. All transistors are bidirectional elements (i.e., no distinction is made between the source and the drain). Using switch-level transistor models, the circuit is represented by an undirected weighted, edge-weighted switch graph G(V,E) similar to the graphs described in [7,11]. Switch-level simulation techniques are applied to the switch graph G(V,E) to evaluate the corresponding transistor network.

2.2. Hierarchical data structures and operations

Storing a circuit description hierarchically demands more involved data structures than a flat representation: besides avoiding the replication of structural information one need to be able to traverse the hierarchy top-down and bottom-up to propagate signal changes. In this section we outline the basic data structures.

A data structure for the circuit description must essentially have two components: one that is concerned with the circuit topology (describing the recursive macro composition of the circuit) and the other maintaining the state of the circuit during the simulation. As discussed earlier, repeated structures need to be stored only once and may be referenced many times. However, different references of one structure will be connected to different parent cells in the hierarchy. Thus, there is interconnection information that is specific to each reference. Clearly, the state (the current set of signal values) is also specific to each reference. Therefore, we distinguish two types of data structure: a base structure, or class, which carries the structure of a module, its fault information etc. and an instantiation structure, or instance, which holds information specific to each reference of a base structure, such as current state and fault lists.

The base structure is implemented through the two data structures cell and node given in pseudo C code in Figure 2.1.

In cell, pin_count and node_count store the number of electrical nodes on the boundary of the cell and the total number of electrical nodes in the cell respectively; nodes is an array of electrical nodes containing the boundary nodes first followed by the internal nodes. The distinction is necessary because in order to reevaluate a certain block of the circuit one needs to load a new set of signal values (also called environment) into the input pins on the boundary of the block; similarly new output values need to be passed upwards in the hierarchy through the output pins. In contrast, internal nodes are recomputed using the standard switch-level operation of selecting the signal value corresponding to the least upper bound of all paths feeding into the node (also called the consensus operation here).

struct cell
{
  string name;
  integer pin_count;
  integer node_count;
  integer_vector nodes;
  integer_vector subcell_count;
  cell_vector subcells;
  function c_function;
  transistor_pointer transistor_net;
  integer fault_count;
  fault_descriptor local_faults;
}

struct node
{
  integer fanout_count;
  integer fanin_count;
  integer_vector fanouts;
  integer_vector fanins;
  integer_vector fain_nodes;
  char type;
}

Figure 2.1. Data structures cell and node.

The field subcell_count contains the number of children cells and subcells is an array of pointers to the subcells. This corresponds to the down links in a hierarchical representation. Primitives need to have either a reference to a software function or to a transistor network (fields c_function and transistor_net).

For fault simulation one needs to keep track of the number of faults that can occur in the cell (fault_count). Fault_descriptor is a pointer to an array of fault descriptors each containing bit encoded information about the local faults in the cell (i.e. the pin number and the signal value under fault).

Node is a substructure of cell and represents electrical nodes at different levels of the hierarchy. Fanin_count (fanout_count) stores the number of fanins (fanouts) to the node. Fanins and fanouts are arrays of cell indices; fanin_nodes store the indices of the fanin nodes relative to the fanin cells. Fanins and fanin_nodes are used to perform the consensus operation for a node after all instances feeding into it have been evaluated.

Furthermore, the structure node contains a type field. In addition to the conventional input/output data types for signals (denoted INPUT, OUTPUT) entering a subcircuit, we introduce a bidirectional type (denoted IOPUT). This captures the notion of bidirectional signal flow and it is used to represent nodes on the boundary of a subcircuit that can both receive a value from the outside and be modified by the operations inside the subcircuit.

The instantiation structure instance is shown in Figure 2.3. Base is the base structure after which the instance is patterned. The state of the

struct instance
{
  cell base;
  state state;
  bit_vector activity;
  instance_pointer parent;
  instance_pointer children;
  integer_vector contacts;
  integer time;
  integer rank;
  integer fault_offset;
  fault_list propagated_faults;
  fault_list local_faults;
}

Figure 2.3 Instance structure.
nodes of the instance are kept in the state vector state. The state of a node consists of its logical value (‘0’, ‘1’, or ‘X’) and strength information (conductance of the path driving the node). In addition, the array of bits activity keeps track of whether a new consensus needs to be computed for a node: this is the case when output signal values of instances connected to the node change. The arrays state and activity have node_count elements.

The pointers parent and children provide up and down links between instances in the tree representation of the hierarchy. Each boundary node of the instance is connected to a node in the parent of the instance. The array contacts keeps the index of the connection in the parent of each boundary node of the instance. It is needed for loading a new environment into the instance whenever it is scheduled for evaluation. The environment is loaded from the parent as will be explained in the next section.

Each instance keeps track of the time of its last evaluation in the field time. This helps avoiding unnecessary reevaluations. Rank gives the level of the instance at the respective level of hierarchy computed by performing a topological sort.

For fault simulation, different copies of the same cell will receive individual fault identification numbers (called fault ids). The first fault id is stored in the field fault_offset. For each instance we maintain two lists of fault records, one holding faults local to the instance (local faults) and the other holding faults that have been propagated to the instance (propagated faults).

Each fault record contains the respective fault id and the corresponding state and activity information. We note therefore that the size of fault records critically depends on the number of nodes at the particular level of hierarchy and can become very large towards higher levels in the hierarchy.

3. Evaluation algorithm

The benefits of hierarchical fault simulation come at the cost of increased complexity of the event scheduler. Scheduling, retrieving, and propagating events become non trivial when an event must travel up and down the hierarchy in order to propagate to all the affected modules. The difficulties are due to the following:

1. Event propagation: The propagation of events is not limited to a single level of the hierarchy.
2. State variables: In the presence of faults proper updating of nodes is very crucial to ensure correct fault effect at higher levels.
3. Delays: Delayed events resulting from faulty and fault-free circuit need to be processed differently at different levels.
4. Consistency: Checks are needed to assure consistency across levels for node description (delays, node type etc.).

In this section we describe the hierarchical evaluation algorithm that is at the heart of F.CHAMP. The evaluation algorithm operates on a single stack. A stack element consists of an instance needing evaluation and flag indicating the direction of evaluation (top or bottom-up). The algorithm uses the following operations:

1. push (instance, flag) to push an instance-flag pair on the stack,
2. pop (instance, flag) to pop from the top of the stack an instance-flag pair
3. top (instance, flag) to get a copy of the instance-flag pair currently on the top of the stack.

3.1. Algorithm

The evaluation procedure is outlined in Figure 3.1. It starts by updating the state of delayed elements. This involves changing node states and propagating their effect to the respective next higher level in the hierarchy and to the rest of the circuit (process_delay). Faulty delayed signals are processed before the corresponding fault free signals, since the previous fault free machine is a reference for all faulty machines, modifying the necessary state variables. After the delayed signal have been processed, the instance top_instance, which corresponds to the root instance of the hierarchy, is pushed on the stack and procedure eval_top() is called to evaluate the effect of a new input pattern.

eval_top() {
    process_delay (current_time);
    push (top_instance, top_down);
    eval inst();
}

Figure 3.1. Procedure eval_top.

eval inst is called to evaluate the effect of a new input pattern.

Procedure eval inst is shown in Figure 3.2. Initially, instance top inst = top is placed on the stack (in eval top) and the evaluation flag is set to top down. Eval inst first evaluates all primitives found on the top of the stack (if any) by either calling the switch-level evaluation procedure if the primitive is described at the transistor level or the associated behavioral C-function. Next, the evaluation flag for instance inst on top of the stack is tested to determine in which direction the evaluation is to proceed. If the direction of evaluation is top-down, the environment is loaded in Procedure load_environment, which copies the value of all signals connected to the boundary. If the environment is being loaded for fault free evaluation, then all faulty copies of inst are evaluated to compute the next state of inst in the presence of each of the faults. After the procedure load_environment the direction flag associated with inst is switched to bottom-up, if the new environment is different from the previous one. In the case of the top instance the environment is given by the new set of primary inputs; else it consists of all signals on the boundary of the instance. To propagate the changed signals downward in the hierarchy all affected subblocks (children instances) are pushed on the stack. Their children in turn are pushed if signals on their boundary have changed, and so forth until the primitive instances are reached.

If the evaluation, on the other hand, is bottom-up, the state of the nodes in inst are updated due to changes coming from the next lower level; this step is accomplished by a call to procedure sense children which computes the new state of nodes in inst resulting from a change in node states at the lower level. In case the new state of a node differs from its old state all instances the node fans out to need to reevaluated and are pushed onto the stack. If the modified node lies on the boundary of the instance the contact node in the parent is notified. If no new events are caused in sense children , the instance inst on top of the stack is popped. The procedure then examine inst to see if it was evaluated due to fault free events. In this case, faults local to children of inst are considered for injection. The fault injection is accomplished in the procedure eval inst() {
    while (! empty (stack)) {
        top (inst, flag);
        while (type (inst) == primitive) {
            pop (inst, flag);
            eval_lowest_level (inst);
            top (inst, flag);
        }
        if (flag == bottom_up) {
            event_count = sense_children (inst, fault_id);
            if (event_count == 0)
                pop (inst, flag);
            else
                inject_faults (inst);
        } else {
            event_count = load_environment (inst, fsl)
            if (event_count == 0)
                pop (inst, flag);
            else
                set_stack_flag_of_inst (bottom_up);
        }
    }
}

Figure 3.2. Procedure eval inst.
inject faults. The effect of inject faults is the activation of faults local to children of test that were evaluated with the fault free machine. In addition, inject faults propagates the effects of faults that have been activated at lower levels.

4. Implementation and Application

4.1. Programming environment

The algorithms described in this paper have been implemented in a prototype software program in the C programming language. The program comprises a total of about 15000 lines of code and runs under UNIX on SUN Microsystems SUN 3 and SUN 4 workstations. The simulator accepts a simple hierarchical description language in which the user specifies circuits by defining primitives and building macros from them hierarchically. A primitive consists of either an interconnection of transistors or the name of a software function. Other description languages are supported through front end translators. The good machine simulation of F_CHAMP was validated by simulating an entire microprocessor with around 250000 input vectors and comparing the outputs with those of a commercial software simulator. We verified the fault simulation by 'hardwiring' selected faults and checking that the ones flagged as detected by the simulator cause corresponding errors and check that faults that are not detected by the simulator do not cause any errors at the output pins of the circuit.

4.2. Performance Experiments

In this section, we describe experiments performed with small to medium sized circuits. In particular, we are interested in how some performance parameters vary as we scale the circuits up.

In Table 1, a summary is given which contains the statistics of simulations for typical MOS designs. The circuits were simulated on a SUN3 workstation. Circuit 1 is a four phase clock generator which contains 101 transistors. Circuits 2 to 5 are 4-, 8-, 16-, and 32-bit full adders respectively. This table show that the CPU time and memory requirement do not grow exponentially with circuit size.

<table>
<thead>
<tr>
<th>Circuit</th>
<th># Elements</th>
<th># Flts</th>
<th># Test</th>
<th>CPU Sec</th>
<th>Pex</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTK 1</td>
<td>100</td>
<td>144</td>
<td>310</td>
<td>84%</td>
<td>58.40</td>
</tr>
<tr>
<td>CTK 2</td>
<td>176</td>
<td>80</td>
<td>10</td>
<td>100%</td>
<td>5.40</td>
</tr>
<tr>
<td>CTK 3</td>
<td>352</td>
<td>160</td>
<td>8</td>
<td>100%</td>
<td>7.76</td>
</tr>
<tr>
<td>CTK 4</td>
<td>704</td>
<td>320</td>
<td>8</td>
<td>100%</td>
<td>14.06</td>
</tr>
<tr>
<td>CTK 5</td>
<td>1408</td>
<td>640</td>
<td>8</td>
<td>100%</td>
<td>51.74</td>
</tr>
<tr>
<td>CTK 6</td>
<td>2300</td>
<td>1620</td>
<td>70</td>
<td>84%</td>
<td>700.74</td>
</tr>
</tbody>
</table>

5. Conclusions

In this paper we introduced an approach for the cost-effective and accurate fault simulation of very large digital designs. It is based on storing and processing the circuit in a hierarchical manner. This way, memory requirements are reduced and faster behavioral descriptions can replace subcircuits at any level of the hierarchy.

After pointing out shortcomings of previous work we described the data structures used and explained the essential steps of the simulation algorithms in Sections 2 and 3 respectively. In Section 4, we presented some experimental results; in particular, we demonstrated the usefulness of the methods by fault grading an entire microprocessor on engineering workstations.

Current and future work concentrates on the use of efficient high level function, integrating test generation algorithms into the fault simulator, and distributed implementation.

6. References


