Sequential Circuit Fault Simulation Using Logic Emulation

Shih-Arn Hwang, *Member, IEEE*, Jin-Hua Hong, *Student Member, IEEE*, and Cheng-Wen Wu, *Senior Member, IEEE*

***Abstract*—A fast fault simulation approach based on ordinary logic emulation is proposed. The circuit configured into our system that emulates the faulty circuit's behavior is synthesized from the good circuit and the given fault list in a novel way. Fault injection is made easy by shifting the content of a fault injection scan chain or by selecting the output of a parallel fault injection selector, with which we get rid of the time-consuming bit-stream regeneration process. Experimental results for ISCAS-89 benchmark circuits show that our serial fault emulator is about 20 times faster than HOPE. The speedup grows with the circuit size by our analysis. Two hybrid fault emulation approaches are also proposed. The first reduces the number of faults actually emulated by screening off faults not activated or with short propagation distances before emulation, and by collapsing nonstem faults into their equivalent stem faults. The second reduces the hardware requirement of the fault emulator by incorporating an ordinary fault simulator.**

***Index Terms*—CLB, fault emulation, fault injection, fault simulation, FPGA, logic emulation, logic testing.**

I. Introduction

FAULT simulation has been heavily used in test pattern generation, fault coverage evaluation, fault dictionary construction, post-test diagnosis, faulty circuit analysis, etc. [1]. In fault simulation, each test pattern is applied to the good circuit as well as every faulty circuit. If the output of any faulty circuit differs from that of the good one, the corresponding fault is said to be detected. Fault simulation is very time consuming since we are simulating many faulty copies of the circuit as well as the original one. Complete fault simulation for large circuits usually requires an unacceptably long period of time, even though many efficient simulation techniques have been reported recently (see, e.g., [2]–[16]). These techniques can generally be classified as follows: 1) parallel pattern single fault propagation (PPSFP), e.g., PARIS [2], PSF [3], and ZAMBEZI [13], which parallelize sequential test vectors by grouping them into packets and applying multiple runs for each packet; 2) single pattern parallel fault propagation (SPPFP), e.g., PROOFS [4], [6] and HOPE [7]–[9], which simulate several faults simultaneously by grouping them into a packet; and 3) distributed fault simulation by fault-set or test-set partition, executed on multiple computer systems [10]–[12], [14]–[16].

Logic design verification can be carried out by a hardware logic emulator or accelerator [17]–[22] rather than a conventional simulator. Unlike simulation, a hardware emulator performs gate evaluation in parallel, which can provide real-time logic operation and fast design verification, hence greatly reduce the design turnaround time. As shown in Fig. 1, a current popular hardware emulator consists of several emulation boards, while each board is composed of numerous RAM-based field programmable gate arrays (FPGA's), possibly connected by field programmable interconnect chips (FPIC's). An FPGA can contain thousands of configurable logic blocks (CLB's), which are connected by programmable interconnects, e.g., an XC4062 FPGA by Xilinx contains 2304 CLB's [23]. The lookup tables (LUT's) and flip-flops (FF's) in a CLB are used to model the module functions of the given circuit. Programming the logic emulator to emulate the target hardware requires circuit *compilation* and FPGA *configuration*. The compilation involves circuit partitioning, placement, and routing for FPGA's, and generation of the bit stream representing the design. The configuration is the download of the bit stream to the FPGA's. When the circuit netlist is modified, it is necessary to *reprogram* (i.e., recompile and reconfigure) the FPGA's.

A hardware emulator or accelerator also can be used to boost the speed of fault simulation [21], [22], [24]–[26]. Both good-value simulation and fault simulation are performed on the emulator. The difficulty in fault emulation is mainly in fault injection. In our previous approach [21], [22], a programmable two-dimensional (2-D) cellular automata (CA) was developed as a dedicated logic simulation and fault simulation accelerator. Although the system was highly parallel and very fast, fault injection was done by reprogramming, and was the performance bottleneck. In the approach used by the University of California, Santa Barbara and Quickturn [24], fault injection was carried out by modifying the behavior of the CLB's to model the faulty function; hence, injecting each fault requires reconfiguring the FPGA. Since they had to reconfigure an entire FPGA even if only one CLB function needs to be changed, the fault injection also was the performance bottleneck of their approach. They then used the independent fault identification [27] and dynamic fault injection techniques to reduce the number of FPGA reconfigurations. Although some new RAM-based FPGA architectures allow reconfiguring only part of the chip, real implementation requires high overhead,

Manuscript received January 12, 1998; revised March 22, 1998. This work was supported in part by the National Science Council, R.O.C., under Contract NSC87-2215-E007-018. This paper was recommended by Associate Editor T. Cheng.

The authors are with the Department of Electrical Engineering, National Tsing Hua University, Hsinchu, Taiwan, R.O.C.

Publisher Item Identifier S 0278-0070(98)05827-8.

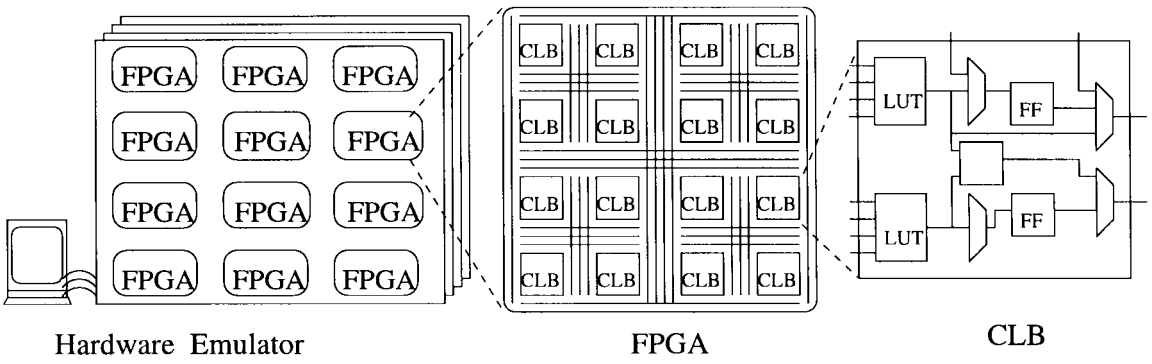

Fig. 1. Typical hardware logic emulator.

and the speedup is very limited. For example, in the recent Meta Systems approach [25], fault injection was done as in [24], but they used a different FPGA architecture so that they were able to modify only a small portion of the FPGA bit stream without the need for entire reconfiguration; hence, the fault injection time was reduced. In their report, an average of 0.8 ms is required to inject a fault, which is still significant in many cases. It can be omitted only when the average number of patterns for each fault is over 10 K. They also implemented a comparator on the emulator to concurrently compare the prestored fault-free output with the emulated output. Their experimental results showed 8–20 times speedup as compared with HOPE. However, reconfiguration time is still visible even for as many as 50K patterns.

In this paper, we present a new fault injection method without FPGA reconfiguration, thus greatly increasing the fault emulation performance. In order to inject faults, we convert the original circuit into a *fault-injectable circuit*, and map the fault-injectable circuit onto the emulator. Injecting faults is made easy by enabling the corresponding *fault injection elements* in the fault-injectable circuit. Efficient methods to control the fault injection elements are then proposed. Experimental results show that the proposed serial fault emulator is about 20 times faster than HOPE on average for the ISCAS-89 benchmark circuits. The speedup grows with the circuit size according to our analysis. Two hybrid fault emulation approaches are also presented. The first approach eliminates the overhead associated with undetected faults, i.e., faults not activated or with short propagation distances are screened off before fault emulation. Also, nonstem faults are collapsed into their equivalent stem faults, reducing the number of faults actually emulated. The second approach reduces the hardware requirement of the configured circuit in the fault emulator by simulating the small proportion of multiple-event faults using an ordinary fault simulator. Experimental results for the hybrid approaches show that they perform much better than the serial fault emulator when the percentage of multiple-event faults is low.

II. Fault Injection

A. CLB Reconfiguration

The CLB's in an FPGA model the module functions of the given circuit. Thus, fault injection can be made by changing the content of a CLB [24], [25]. As shown in Fig. 2, the CLB implements the function $g = (ab)'(cd)'$ when no fault is injected. When the fault $e/1$ (for e stuck at 1) is injected, the function of the CLB changes to $g = (cd)'$. Likewise, if now $f/1$ is injected, the function changes to $g = (ab)'$.

In some cases, we may have to modify more than one CLB when injecting a fault. This is because the module function may be replicated when doing technology mapping to reduce the overall routing cost of the FPGA. Changing the behavior of a CLB requires reprogramming the FPGA. When partial reconfiguration is not possible, fault injection requires reconfiguring the entire FPGA, and hence is very time consuming [24]. If partial reconfiguration is available, then the fault injection time can be reduced, i.e., reconfiguration affects only a few CLB's [25]. However, fault injection may still be too slow as compared with the pattern application rate, and can be the bottleneck. In [24], a *dynamic fault injection* approach to reduce the number of FPGA reconfigurations is proposed. Each CLB implements both the fault-free and faulty functions, selected by a fault activation signal, i.e., one fault can be injected into one CLB without reconfiguration. The fault activation signals are controlled by a circular shift register to inject faults to CLB's one by one. Reconfiguration thus is not required for the current group of faults stored in the CLB's, but is still needed when the next group of dynamic faults is to be loaded.

B. Fault Injection Scan Chain

In our approach, a *fault-injectable circuit* is synthesized from the given circuit netlist and the fault list. Fault injection using the fault-injectable circuit is done by three types of built-in *fault injection element* (FIE) as shown in Fig. 3, where FIE0 (FIE1) performs the fault-free function or models the stuck-at-0 (stuck-at-1) fault when the test-mode input (T) is 0 or 1, respectively, and FIE10 combines both FIE0 and FIE1, i.e., it can model stuck-at-0 and stuck-at-1 faults. The FIE's are placed at the fault sites. The control of the FIE's is made easy by adding a *fault injection scan chain* into the fault-injectable circuit. The test-mode inputs of the FIE's are then connected to the outputs of the FF's in the scan chain. The circuit is fault free when the values of the FF's are all zero. We then initialize the content of the scan chain to $100\cdots0$, and faults can be injected into the circuit one by one by shifting the data in the scan chain.

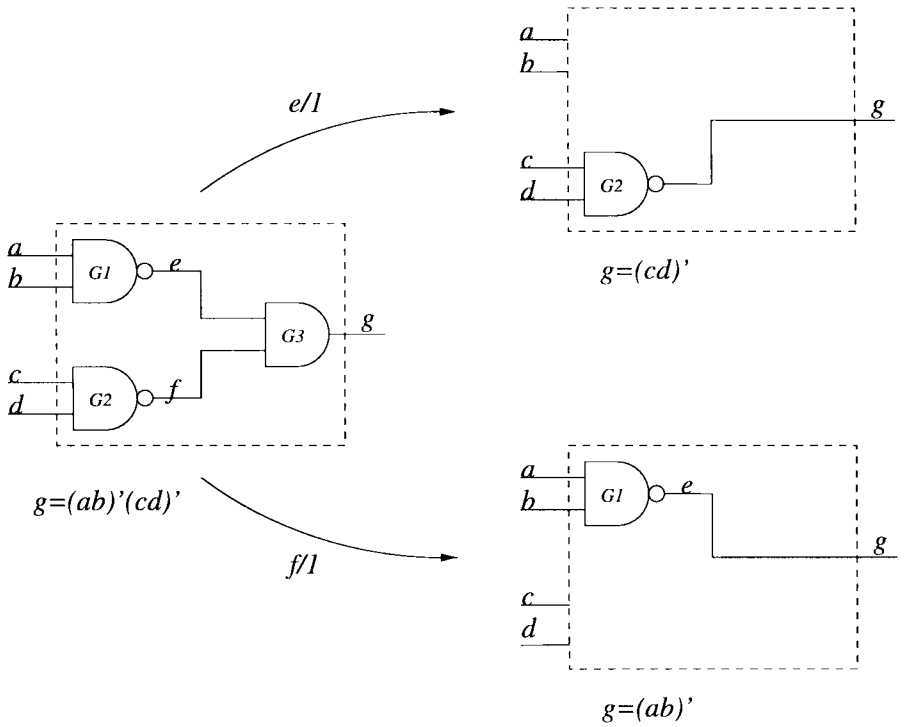

Fig. 2. Fault injection by changing the content of a CLB.

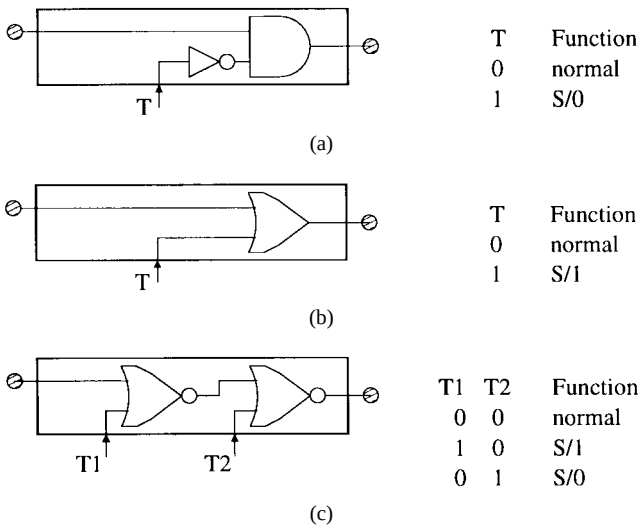

Fig. 3. Three fault injection elements (FIE's).

For example, referring to Fig. 4(a), suppose the faults to simulate are $e/0, e/1, f/1$, and $g/0$. We add FIE10 to node e, FIE1 to node f, and FIE0 to node g, respectively. We then connect all of these FIE's to the fault injection scan chain, and the resulting fault-injectable circuit is as shown in Fig. 4(b). When the content of the FF in the scan chain is 0000, the circuit is fault free. If the content is 1000 instead, then the circuit has the fault $e/1$. Likewise, 0100 represents $e/0$, 0010 represents $f/1$, and 0001 represents $g/0$.

We also want to reduce the routing cost of connecting the FIE's into a fault injection scan chain. We can connect the FIE's in the order of their levels in the circuit and their positions in the level. Another way is to take advantage of independent faults [27]. The average size of an independent fault group in the ISCAS-89 benchmark circuits is about 1.38 [24], [27], so we can reduce the number of FIE's to 1/1.38 of its original size on average with this method.

As opposed to [24], where a fault is injected by modifying the content of the CLB's, we perform fault injection directly on the original circuit; thus, all faults to be simulated can be stored in the fault-injectable circuit before the FPGA bit stream is generated, i.e., bit-stream regeneration is not necessary. Also, in [24], only one fault can be injected into one CLB at a time, while in our approach at least two faults can be injected into one CLB according to the experimental results. However, the comparison of efficiency between these two methods is difficult since the CLB's used are different—our CLB can implement two arbitrary 4-to-1 functions, while theirs can implement one arbitrary 5-to-1 function. Moreover, we are unable to compare the hardware overhead per fault between these two methods since there are no data about the number of faults stored in the emulator per reconfiguration as reported in [24]. It is reasonable to assume that the hardware efficiency of the two methods is close since both use a scan chain to inject faults.

C. Parallel Fault Injection Selector

The underlying logic emulator uses XC4000-series FPGA's. Therefore, in the fault-injectable circuit with the fault injection scan chain, the number of CLB's required is approximately half the number of faults emulated [26]. This is because one FF is required for injecting one fault, and one CLB in the XC4000-series FPGA can implement only two FF's [23]. The fault injection scan chain thus consumes the most CLB's. The scan chain controls the FIE's independently. However, only one FIE can be activated (i.e., the test-mode input be set to 1)

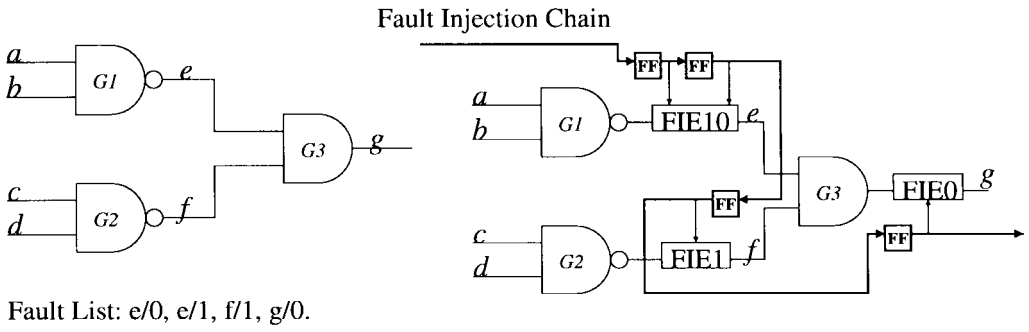

Fig. 4. Fault injection scan chain example.

TABLE I
DECODER FUNCTION WHERE THREE FAULTS ARE TO BE EMULATED

	Input	Output
Fault-free	00	000
Fault #1	01	001
Fault #2	10	010
Fault #3	11	100

during each fault injection step (since a single stuck-at fault is assumed) so the control scheme can be simplified. A simpler way to control the FIE's is to use a *parallel fault injection selector*, which consists of a decoder and an optional counter. The input of the decoder is the binary representation of the location of every fault in the fault list, e.g., the first fault is represented as $0 \cdots 01$, the second fault $0 \cdots 010$, the third fault $0 \cdots 011$, etc. The output of the decoder activates only the corresponding FIE for the input fault. Note that when the input of the decoder is the all-zero vector, the circuit is fault free, and the decoder disables all FIE's. The decoder size is $\lceil \log(N_f + 1) \rceil \times N_f$, where N_f is the number of faults to be emulated. Table I shows an example in which three faults are to be emulated in the fault-injectable circuit.

No FF is required in the parallel fault injection selector. However, the extra pins needed for the parallel fault injection selector are $\lceil \log(N_f + 1) \rceil$, which slightly increases with the number of faults. If the faults are arranged in a fixed order, then we can use a counter to control the decoder and enable the fault injection elements one by one. The only extra pin required in this structure is the Scan_Clock of the counter. For example, consider the previous example circuit as shown in Fig. 4(a). The corresponding fault injectable circuit using the parallel fault injection selector is depicted in Fig. 5. Using the parallel fault injection selector, the number of FF's required in the fault-injectable circuit is reduced from N_f to $\lceil \log(N_f + 1) \rceil$. The number of CLB's, however, is not reduced by the same amount—only about 25% according to our experimental results (to be shown in Table II). The reason is that the decoder of the parallel fault injection selector consists of many large-fan-out gates, and a CLB can only implement two-output functions. Moreover, the bit-stream generation time becomes longer, and the memory required for FPGA optimization also increases. When the number of FIE's is high, bit-stream generation may not be possible due to memory limitation.

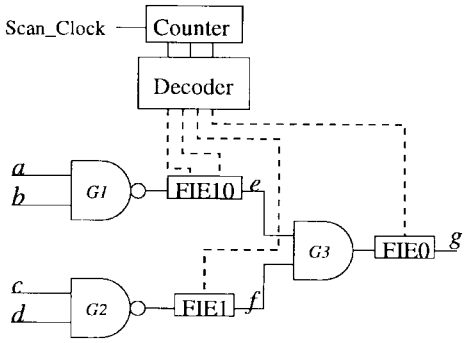

Fig. 5. Example of using parallel fault injection selector to control the FIE's.

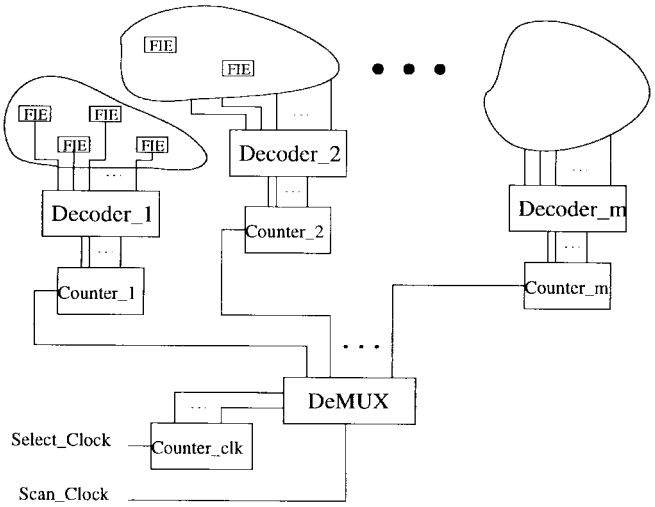

Fig. 6. Distributed parallel fault injection selector architecture.

To solve the memory problem, we propose a distributed parallel fault injection selector architecture as shown in Fig. 6. Instead of using a single parallel fault injection selector, we use several smaller parallel fault injection selector modules to control the FIE's. Suppose the decoder size of the parallel fault injection selector module is k-to-$(2^k - 1)$, i.e., $2^k - 1$ test-mode inputs can be controlled by a selector module. Then, $\lceil N_f/(2^k - 1) \rceil$ selector modules are required in the fault injectable circuit, and $k \times \lceil N_f/(2^k - 1) \rceil$ FF's are needed for implementing the counters. Also, a 1-to-$\lceil N_f/(2^k - 1) \rceil$ demultiplexer is used to pass the Scan_Clock to the selector module which contains the FIE to be activated. The

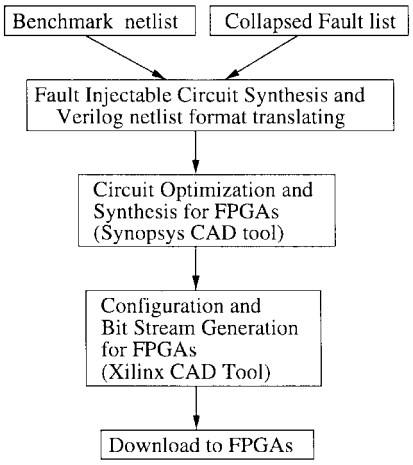

Fig. 7. Preprocessing.

demultiplexer is controlled by a $\log\lceil N_f/(2^k-1)\rceil$-bit counter, which can reduce the pin overhead of the fault-injectable circuit. The disadvantage of the distributed scheme is that its hardware overhead is higher than the original parallel fault injection selector scheme. We will show, in our experimental results, that the smaller the decoder size, the higher the hardware overhead.

III. Serial Fault Emulation

To determine whether a fault can be detected by the test patterns, the test patterns are run on the good circuit as well as all faulty circuits. If the output of any faulty circuit differs from the good one, the corresponding fault is said to be detected. Thus, in this serial fault emulation scheme, the emulator has to model the faulty circuit for each intended fault. We have developed a preprocessing tool for synthesizing the fault-injectable circuit from the given circuit netlist and fault list. The preprocessing flow diagram is shown in Fig. 7. The fault-injectable circuit is optimized for FPGA mapping by a Synopsys tool. After that, we use the Xilinx FPGA technology mapping tool, XMAKE, to generate the corresponding FPGA bit stream. Finally, we download the bit stream to the FPGA's. The flow of our serial fault emulation is as follows:

```
Serial_Fault_Emulation( )
  {disable all FIE's;
   for(each test pattern)
     {perform fault-free emulation;
      store the primary-output data;
     }
   for(each fault in the fault list)
     {initialize circuit FF's;
      inject the fault by enabling the
      corresponding FIE;
      for(each test pattern)
        {perform fault emulation;
         compare outputs with fault-free
         outputs; if detected then break;
        }
     }
  }
```

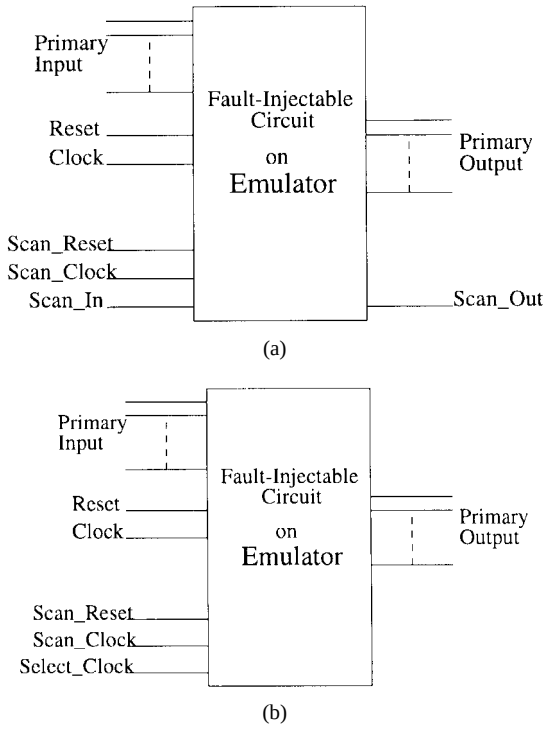

Fig. 8. Pin overhead for fault-injectable circuit: (a) with fault injection scan chain and (b) with parallel fault injection selector.

The fault list is arranged such that fault injection time (i.e., the shifting time of the fault injection scan chain or the count-up time of the parallel fault injection selector) is minimized. When a fault is detected, it is dropped immediately, and the next fault is injected. The number of extra pins required in the fault-injectable circuit is constant regardless of the number of faults. As depicted in Fig. 8, we only need three extra pins to do fault injection. Scan_Reset is used to reset the FF's controlling the FIE's, i.e., to map the fault-free circuit to the emulator. For the fault injection scan chain [see Fig. 8(a)], Scan_In and Scan_Clock are used to shift data in the chain to inject the faults, while Scan_Out is optional, which can be used if we want to monitor the internal circuit. For the parallel fault injection selector [see Fig. 8(b)], Select_Clock is used to activate the injection modules, and then Scan_Clock is used to select the FIE of the active module.

The performance estimation of the serial fault emulation is as follows. Let N_t be the number of test patterns, N_g the gate count, N_f the fault count, and T_{cpu} the single-gate evaluation time of the simulator, then the serial fault simulation time is

$$T_{FS} = N_t N_f N_g T_{cpu}. \tag{1}$$

The simulation time for HOPE and PROOFS is assumed to be reduced by a constant factor since they simulate several faults simultaneously by grouping them into a packet. Let T_{emu} be the single-gate evaluation time of the emulator and L_g the circuit depth, i.e., the number of gates (or CLB's) in the critical path of the configured circuit, then the serial fault emulation time is

$$T_{SFE} = N_t N_f L_g T_{emu}. \tag{2}$$

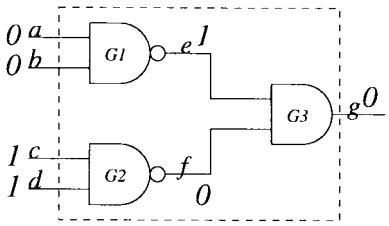

Fig. 9. Circuit under emulation—all internal values of the good circuit are known.

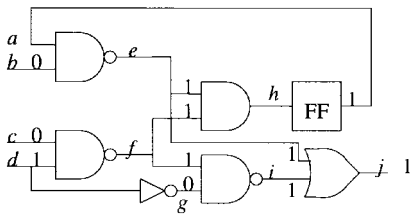

Fig. 10. Example of single- and multiple-event faults.

In [24] and [25], where reconfiguration is required for every fault (or fault group) injection, the fault emulation time thus becomes

$$T'_{SFE} = \frac{N_f}{F_g} T_R + N_t N_f L_g T_{emu} \tag{3}$$

where F_g is the average number of faults stored in the emulator per reconfiguration, and T_R is the reconfiguration time. The first term in (3) usually degrades the overall performance since T_R can be as high as several milliseconds [24], [25]. Our approach needs no reconfiguration. The speedup of our serial fault emulator over the serial fault simulator is

$$\frac{T_{FS}}{T_{SFE}} = \frac{N_g T_{cpu}}{L_g T_{emu}}. \tag{4}$$

For example, consider a circuit with $N_g = 10\text{K}$, and the clock cycle of the configured circuit is 1 μs, i.e., $L_g T_{emu} = 1$ μs. If $T_{cpu} = 10$ ns, then the speedup becomes $(10\text{K} \times 10\ \text{ns}/1\ \mu\text{s}) = 100$. Higher speedup can be expected for large circuits since N_g/L_g grows with the gate count.

Circuits of the same size may not have the same speedup since they may have a different percentage of faults that are not detected by the applied test patterns. The faults that are not detected by the applied test patterns consume a large proportion of the emulation time since we need to apply all test patterns before we know that they are not detected. However, in fault simulation, faults not detected by the applied test patterns can be identified with relatively less effort, so they will not go through the real simulation process. We will explore speedup techniques to tackle this problem in the next section.

IV. Hybrid Fault Emulation

A. Integration with Logic Simulator

To improve the speedup of the fault emulator, we use a logic simulator to perform the fault-free simulation, which needs to be done only once for each pattern. When the good values of all gate outputs in the circuit are known, heuristics [4], [7], [8] can be used to further reduce the number of faults needed to be emulated. If we perform good-value emulation, we obtain only the primary-output (PO) values since internal values cannot be obtained without a high cost. For fault emulation, PO values are not enough because we need to know as early as possible if a fault effect cannot be propagated to the PO to avoid unnecessary work. For example, consider the circuit shown in Fig. 9. Assume that the only good value we can obtain is that of the PO g. Now, assume that the fault $e/1$ is being simulated with the pattern 0011. As we can see from the figure, the fault $e/1$ is not activated, but the emulator does not know this, and will proceed until the PO value of the faulty circuit is produced. If we perform the fault-free simulation first, then good values for e and f will be known in advance, and we will be able to stop immediately as soon as we find that $e/1$ is not activated by this test pattern. Now, if $e/0$ is being simulated, it will be activated but not propagated through $G3$ since the off-path value of $G3$ is a controlling value, so we can stop the process right there, too. Therefore, the number of faults to be emulated can be reduced by obtaining the internal values from a fault-free logic simulator.

Faults in the sequential circuits can be classified into two types, i.e., *single-event fault* and *multiple-event fault*. If a fault has not been detected but its fault effect was propagated to the FF's in the previous time frame, and therefore in the present time frame the fault effect originates from the faulty FF's as well as the fault site, then it is a multiple-event fault; otherwise, it is a single-event fault. For example, consider the circuit shown in Fig. 10. Assume that the test pattern is 001 and the content of the FF is 1. When the fault $b/1$ is present, it is not detected by the PO j, but the input value of the FF will be faulty, i.e., the fault effect of $b/1$ will be propagated to the next time frame. Therefore, $b/1$ is marked as a multiple-event fault in the next time frame. Note that a single-event fault in the present time frame may become a multiple-event fault in the next time frame, and vice versa.

For a single-event fault, the internal values for the present pattern obtained from logic simulation can be used to check whether its fault effect can be propagated to its gate output or several levels further by this pattern [4], [7], [8]. If the single-event fault is declared unpropagatable, then further emulation is unnecessary. A more efficient method is the use of critical path tracing [5], [28] on every fan-out-free region (FFR) of the circuit. Critical path tracing screens off the single-event faults with short propagation paths and collapses them simultaneously. We map the single-event faults which can be propagated out of the corresponding FFR to the equivalent stem fault, and only emulate the stem faults. Thus, we have to add an FIE10 to every stem to model the stem fault, which can be the stuck-at-1 or stuck-at-0 fault depending on the collapsed single-event faults. The fault-injectable circuit in hybrid fault emulation is as shown in Fig. 11. Note that multiple-event faults cannot be collapsed to stem faults by logic simulation because the current fault effect may originates from the fault site as well as the FF's. The proposed hybrid fault emulation

procedure is shown as follows:

```
Hybrid_Fault_Emulation()
 {
  each fault is initially marked 1;
  for(each test pattern Vi, i = 1 to Nt)
    {perform fault-free logic simulation
     and emulation;
     for(each single-event fault marked i)
       collapse to stem fault;
     for (each stem fault activated)
      {perform fault emulation;
       compare output with fault-free
           output;
       if detected then report the
           equivalence group and continue
           to the next fault.
       else check the FF output;
       if changed then
        {mark the equivalence group as
         multiple-event faults;
         for(each multiple-event fault in
             the group)
           {for(each remaining test
                pattern Vj, j=i+1 to Nt)
              {perform fault-free
                  emulation;
               perform fault emulation;
               compare output with fault-
                   free output;
               if detected then break;
               else check the FF output;
                 if not changed then
                   {mark it as single-event
                    fault with j+1;
                    break;}
              }
                  }
              continue to the next
              activated stem fault;
           }
          else mark the equivalent
              single-event faults with
              i+1;
         }
     }
 }
```

As depicted in the procedure, we collapse single-event faults to stem faults for each test pattern, so the internal node values need not be stored for the next pattern. We emulate the selected stem faults one by one for each pattern. If the fault effect is propagated to FF's (i.e., become multiple-event faults), we immediately apply the remaining patterns in the test sequence for each fault in the group until it is detected or turned into a single-event fault again. After emulating all multiple-event faults in the group, we continue to the next single-event fault. For example, suppose a single-event fault f_x becomes a multiple-event fault after we have entered the test pattern V_i. We will then apply $V_j, j = i+1, i+2, \cdots, N_t$ in series until it is detected or turned into a single-event fault again. If f_x becomes a single-event fault after we have entered $V_k, i+1 \leq k < N_t$, then f_x is marked $k+1$, and it will not be simulated (collapsed) in the single-event fault loop until $i = k+1$. Note that every single-event fault is tagged with a pattern number. Only those single-event faults having the same pattern number as the present pattern can be collapsed to stem faults.

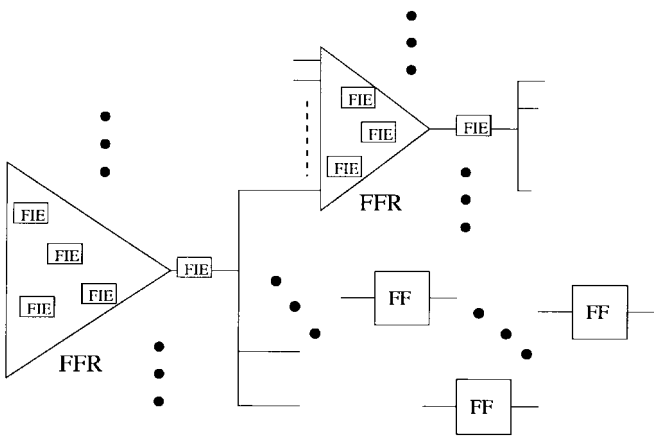

Fig. 11. Fault-injectable circuit for hybrid fault emulation with logic simulator.

In hybrid fault emulation, the faults to be injected are determined during emulation; hence, we cannot know the fault activation order in advance. We use a parallel fault injection selector without counter to inject faults in constant time. This, however, slightly increases the pin count. Also, the FF's shown in Fig. 11 are modified as in Fig. 12. In the figure, Qa's and Qb's are used to store the FF contents of the single-event faults, while Qc's, Qd's, and Qe's are used to store the FF contents of the multiple-event faults. When we emulate single-event faults, we set $Smef$ to 0, and Qa's and Qb's store the fault-free FF contents of the present time-frame and that of the previous time frame, respectively. After each fault-free emulation, we store the FF values into Qa's by applying one clock pulse to $Fclk1$, and at the same time, the FF contents of the previous time frame (originally in the Qa's) are updated into Qb's. If the fault effect is propagated to FF's in the current cycle, *FF_Faulty* will be set to 1. In that case, we apply one clock pulse to $Fclk3$ to update the faulty FF values into Qe's, which store the faulty FF values of the current multiple-event fault group. Then, we apply one clock cycle to $Fclk2$, so Qc's and Qd's store the present faulty and fault-free FF values, respectively. $Smef$ is subsequently set to 1. For each multiple-event fault, we perform logic and fault emulation for each pattern. After logic emulation, we apply one clock pulse to $Fclk2$, so the present fault-free FF contents are updated to Qc's and the previous faulty FF contents (originally in Qc's) are updated to Qd's. We then perform fault emulation. If *FF_Faulty* $= 1$ after fault emulation and the fault is not being targeted, then we apply one clock pulse to $Fclk2$, such that the present faulty FF values are updated to Qc's, and the present fault-free FF values are updated to Qd's. We continue the process for the next pattern. If *FF_Faulty*$= 0$, then the

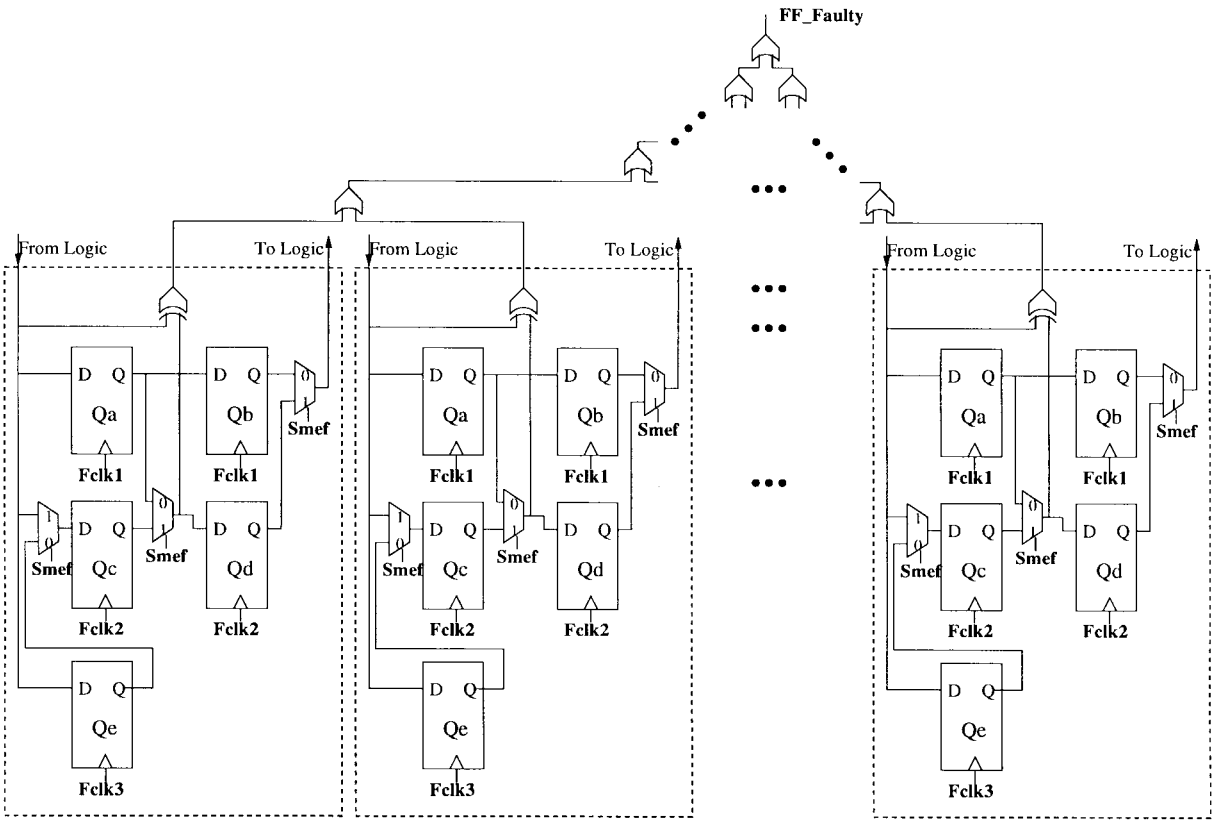

Fig. 12. Restoring and checking FF values.

multiple-event fault is turned back into a single-event fault. Before moving to the next multiple-event fault, we need to restore the faulty and fault-free FF values of the current group to Qc's and Qd's, respectively. This is done by first letting $Smef = 0$ and applying one clock cycle to $Fclk2$. Then we let $Smef = 1$ and emulate the next multiple-event fault. After emulating all multiple-event faults of the current group, let $Smef = 0$, and continue to the next single-event fault.

During the process, we need to check whether the fault effect is propagated to FF's for every selected stem fault and every multiple-event fault. The checking time is proportional to the depth C_g of the OR tree as shown in Fig. 12, where $C_g = \lceil \log N_{FF} \rceil$ and N_{FF} is the number of FF's. Note that C_g is normally small as compared with the depths (L_g) for large circuits. For example, for *s*38471, $L_g = 48$, while $N_{FF} = 1636$, so $C_g = \lceil \log 1636 \rceil = 11$. The gap between C_g and L_g rapidly increases with the circuit size. The checking time for FF's therefore can be omitted in hybrid emulation.

For combinational circuits, since there is no multiple-event fault, only the FIE's for the stems are required. The corresponding fault-injectable circuit is shown in Fig. 13, where the FIE's inside the FFR's are removed. Suppose that the average number of gates in an FFR is $N_{g_{FFR}}$; then the number of FIE's is reduced to $1/N_{g_{FFR}}$ of that in the serial fault emulator.

We now analyze the performance of the hybrid fault emulator. Let the number of faults emulated be $N_{f_{HFE}}$. Since the emulated faults include multiple-event faults and collapsed stem faults (single-event faults), and we perform logic and

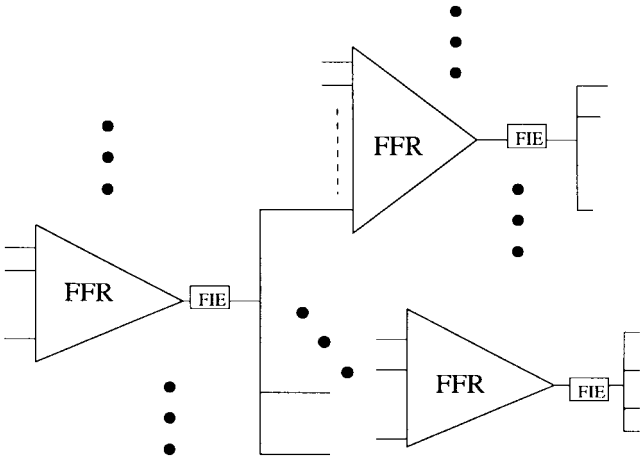

Fig. 13. Fault-injectable circuit for a combinational circuit.

fault emulation for each multiple-event fault, we have

$$N_{f_{HFE}} = 2P_m N_f + \frac{(1 - P_m)N_f}{R_s} \tag{5}$$

where P_m is the percentage of the multiple-event faults, and R_s is the ratio of the number of single-event faults to the number of collapsed stem faults emulated. The hybrid fault emulation time thus is

$$T_{HFE} = N_t N_g T_{cpu} + N_t N_g T_{cpu} + N_t N_{f_{HFE}} L_g T_{emu} \tag{6}$$

where the first, second, and third terms are the logic simulation time, fault collapsing (i.e., critical path-tracing analysis) time,

and fault emulation time, respectively. Therefore,

$$\frac{T_{HFE}}{T_{SFE}} = \frac{N_t(2N_g)T_{cpu} + N_t N_{f_{HFE}} L_g T_{emu}}{N_t N_f L_g T_{emu}} = \frac{2N_g T_{cpu}}{N_f L_g T_{emu}} + \frac{N_{f_{HFE}}}{N_f} = \frac{2T_{cpu}}{L_g T_{emu}} + \frac{1}{R_{HFE}} \quad (7)$$

where $R_{HFE} = N_f/N_{f_{HFE}}$. For example, suppose the emulation speed is 1 MHz, i.e., $L_g T_{emu} = 1\ \mu$s, and $T_{cpu} = 10$ ns; then $2T_{cpu}/L_g T_{emu} = 0.02$. Also, R_{HFE} is about 6 (to be shown in Table VII), so the speedup of the hybrid fault emulation is about $1/(0.02 + 0.1) = 8.33$. Since L_g increases with the circuit size, $2T_{cpu}/L_g T_{emu}$ can be neglected for large circuits, so the speedup of hybrid fault emulation becomes R_{HFE}.

B. Integration with Fault Simulator

We found that almost all FIE's, except those attached to stems, are used only when we emulate multiple-event faults. Since only a small portion of faults are multiple-event faults, we can simulate them on a fault simulator instead of the emulator to reduce the hardware overhead, i.e., the number of FIE's is reduced to $1/N_{g_{FFR}}$ of that in the serial fault emulator. The resulting fault-injectable circuit is similar to that for combinational circuits as shown in Fig. 13. The only difference is that we still need to check the FF contents for the fault effect. The modified FF's shown in Fig. 12 can be used for this purpose, but only Qa's, Qb's, and the checking circuit (i.e., the XOR and OR gates) are required since we only emulate single-event faults.

Since multiple-event faults are simulated while the collapsed single-event faults are emulated, the total time required for the hybrid fault emulation with a fault simulator is

$$T_{HFE\text{-}F} = N_t N_g T_{cpu} + N_t N_g T_{cpu} + N_t P_m N_f N_g T_{cpu} + N_t \frac{(1-P_m)N_f}{R_s} L_g T_{emu} \quad (8)$$

which includes the logic simulation time, fault collapsing time, fault simulation time, and fault emulation time, respectively. Note that for large circuits, the fault simulation time will dominate, and the benefit gained from fault emulation can disappear. Whether we should use the fault simulator depends on the difference between the simulator speed and emulator speed, as well as the emulator capacity. However, this scheme is much faster then the conventional fault simulator if P_m is small since the speedup is $1/P_m$.

V. Experimental Results

We have constructed a small experimental fault emulator as shown in Fig. 14, in which the emulator board (FPGA board) contains six Xilinx XC4000-series FPGA chips. A PC with an i8255 I/O card is used to send test patterns and fault injection data to the emulator and receive the responses from the emulator. The data transmission rate between the PC and emulator is 100K patterns/s using this low-cost interface card. However, since the emulator can perform logic emulation at several million patterns per second and a high-speed interface can be used for real applications, we assume a very conservative 1M patterns/s application rate in our performance evaluation.

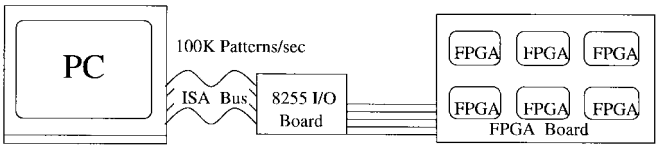

Fig. 14. Our emulation environment.

TABLE II
Hardware Overhead for the Serial Fault Emulator

Circuit	N_g	N_f	Cir	FISC (α)	PFIS (α)
s298	136	308	15	188 (12.53)	133 (8.87)
s344	184	342	18	200 (11.11)	141 (7.83)
s382	182	399	23	244 (10.61)	175 (7.61)
s400	186	424	22	263 (11.95)	183 (8.32)
s444	205	474	23	296 (12.87)	196 (8.52)
s526	217	555	24	329 (13.71)	235 (9.79)
s713	447	581	31	336 (10.83)	299 (9.64)
s820	312	850	53	473 (8.92)	369 (6.96)
s832	310	870	54	484 (8.96)	376 (6.96)
s953	440	1079	71	611 (8.61)	448 (6.31)
s1196	561	1242	91	744 (8.17)	740 (8.13)
s1238	540	1355	94	772 (8.21)	568 (6.04)
s1423	748	1515	73	831 (11.38)	631 (8.64)
s1488	667	1486	112	855 (7.63)	628 (5.61)
s1494	661	1506	112	865 (7.72)	644 (5.75)
s5378	2993	4603	233	2775 (11.91)	2090 (8.97)
s9234	5844	6927	252	3864 (15.33)	2898 (11.50)
s13207	8651	9815	369	5101 (13.82)	4140 (11.22)
s38417	23843	31180	1225	16501 (13.47)	11748 (9.59)
Average				(10.93)	(8.22)

Cir: original circuit.
FISC: fault injectable circuit using fault injection scan chain.
PFIS: fault injectable circuit using parallel fault injection selector.

ISCAS-89 benchmark circuits have been used to evaluate our serial fault emulator. Table II lists the number of CLB's required for the benchmark circuits, the corresponding fault-injectable circuits using a fault injection scan chain, and those using a parallel fault injection selector. The symbol α represents the ratio of the CLB count used in the fault-injectable circuit to that in the original circuit. As shown in the table, the number of CLB's used in the fault-injectable circuit using a fault injection scan chain is approximately half the number of faults emulated. This is because one FF is required for one fault and one CLB can implement two FF's [23]; hence, the fault injection chain in the fault-injectable circuit occupies the most CLB's. The use of the parallel fault injection selector is more cost effective, and the CLB count is reduced by about 25% as shown in the table. Also, the size of the parallel fault injection selector used is 7-to-127. A lower α can be expected for a larger selector.

Table III lists the hardware overhead of the distributed parallel fault injection selector with respect to the module size. From the table, the larger the module size, the lower the hardware overhead. Also, as shown in Fig. 15, the number of

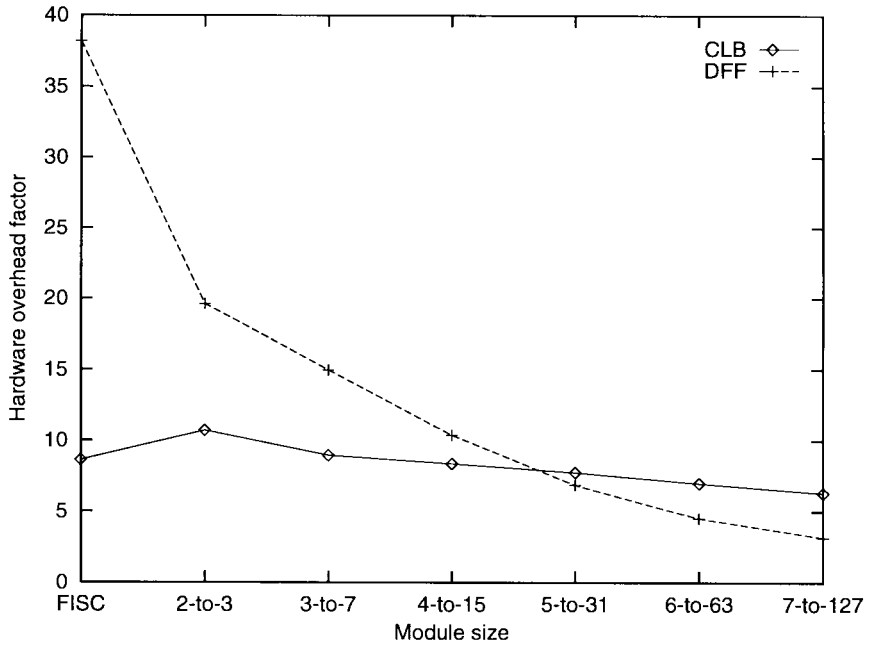

Fig. 15. Comparison of CLB overhead and FF overhead for parallel fault injection selector modules of different sizes.

TABLE III
DISTRIBUTED PARALLEL FAULT INJECTION SELECTOR WITH MODULES OF DIFFERENT SIZES (*s*953)

Module size	#CLB	#FF	α_{CLB}	α_{FF}
Fault injection scan chain	611	1108	8.61	38.21
2-to-3	760	569	10.70	19.62
3-to-7	634	434	8.93	14.97
4-to-15	593	301	8.35	10.38
5-to-31	550	199	7.75	6.86
6-to-63	496	131	6.99	4.52
7-to-127	448	92	6.31	3.17

FF's required decreases rapidly with the increase of the module size, but the CLB overhead decreases much more slowly. This is because the hardware overhead in the parallel fault injection selector is dominated by high-fan-out gates in the decoder, while that in the fault injection scan chain is dominated by the number of FF's. The computer memory required to compile the FPGA bit stream also increases with the module size. In our experiment, when the module size is increased to 8-to-255, the compilation of the FPGA bit stream is aborted due to insufficient memory.

The number of CLB's needed does not always reflect the hardware cost, which is the number of FPGA's required, because the number of gates which can be put into an emulator may depend on the number of pins rather than CLB's needed. The number of extra pins needed in the fault-injectable circuit is constant, so we are using the FPGA's more efficiently in the fault-injectable circuit than in the original circuit, and the actual hardware overhead is less than α normally. Also, the hardware overhead is less an issue in fault emulation since the emulator is reusable.

The two largest ISCAS-89 circuits (i.e., *s*13207 and *s*38417) cannot be configured due to the capacity limit of our small experimental serial fault emulator, but we have estimated the emulation time. Our estimation model, as shown in the previous section, has been justified by the circuits actually emulated, and is shown to be accurate. The emulation time covers all steps in the serial fault emulation, including the bit-stream configuration (download) time, pattern application time, communication time, and output comparison time. The bit-stream compilation time is not counted since the compilation is done only once for each circuit, regardless of the test set and fault set. We compare our serial fault emulator with HOPE [7], [8]. When running HOPE on a Pentium-100 PC, we reset the FF's to zero, which is the same condition as our serial fault emulator. This is because, presently, our fault emulator implements two-valued logic, and thus cannot handle potentially detected faults, which are unknown values at the primary outputs (PO's)—they may or may not be detected, depending on the initial values of the FF's. For sequential circuits, the percentage of potentially detected faults is usually very low [24]. Also, our fault emulator can easily be modified to implement three-valued logic to emulate potentially detected faults by applying the dual-railed logic proposed in [24]. Table IV lists the results using 100K random patterns, and the average speedup is 22.876. Although the speedup for circuits of about the same size varies, the speedup increases with the gate count in general. Table V lists the estimated results for the two largest circuits *s*13207 and *s*38417, which reveal that higher speedup can be obtained for larger circuits.

Table VI lists the average gate count of an FFR (i.e., $N_{g_{FFR}}$) and the CLB count for the hybrid fault emulator emulating only stem faults. The average $N_{g_{FFR}}$ is 5.728, as shown in the table. We have discussed in Section IV-B that the number of FIE's will be reduced to $1/N_{g_{FFR}}$ (i.e., 1/5.728) of that of the serial fault emulator if we simulate multiple-event faults by a fault simulator. However, the average α is 2.89,

TABLE IV
PERFORMANCE COMPARISON USING 100K RANDOM TEST PATTERNS

Circuit	FC (%)	HOPE (sec)	SFE (sec)	Speedup (SFE/HOPE)
s298	87.338	19.950	2.272	8.781
s344	97.840	25.967	0.916	28.348
s382	18.546	218.733	16.960	12.897
s400	18.396	229.950	18.017	12.763
s444	13.924	285.467	20.482	13.937
s526	12.613	298.767	24.446	12.221
s713	82.272	85.583	5.245	16.318
s820	50.000	195.700	21.971	8.907
s832	48.996	207.417	22.925	9.048
s953	99.073	51.500	1.683	43.533
s1196	99.275	114.167	2.141	53.324
s1238	94.244	145.683	5.199	28.020
s1423	67.525	549.333	30.572	17.968
s1488	79.475	169.150	17.978	9.409
s1494	78.685	176.533	18.847	9.367
s5378	66.956	1685.200	77.829	21.653
s9234	5.904	26858.433	325.978	82.393
Average				22.876

TABLE V
ESTIMATED PERFORMANCE FOR LARGE CIRCUITS

Circuit	FC (%)	HOPE (sec)	SFE (sec)	Speedup (SFE/HOPE)
s13207	31.055	19647.800	353.496	55.581
s38417	13.047	483057.194	1360.333	355.101

TABLE VI
HARDWARE OVERHEAD FOR HYBRID FAULT EMULATOR

Circuit	N_{gFFR}	#CLB	α
s298	5.667	40	2.67
s344	4.848	57	3.17
s382	4.647	56	2.43
s400	4.263	63	2.86
s444	3.620	77	3.35
s526	5.676	67	2.79
s713	4.912	120	3.87
s820	8.500	80	1.51
s832	8.441	82	1.52
s953	2.599	230	3.24
s1196	3.550	223	2.45
s1238	3.215	238	2.53
s1423	5.568	203	2.78
s1488	9.328	147	1.31
s1494	9.243	147	1.31
s5378	3.786	1078	4.62
s9234	6.455	1066	4.23
s13207	8.904	1415	3.83
s38417	5.623	5512	4.49
Average	5.728		2.89

which is 1/3.782 of that for the serial fault emulator, i.e., α is not reduced accordingly. This is because the routing cost is not reduced in proportion to the decrease of the number of FIE's. Nevertheless, the hardware overhead reduced is still significant.

Table VII lists the percentage of multiple-event faults (P_m), the ratio of the number of single-event faults to the number of collapsed stem faults (R_s), and the results obtained by (5) for random test patterns (R_{HFE}). The average values for R_s and P_m are 36.991 and 6.729%, respectively. Therefore, by (5), $R_{HFE} = 6.27$, and the expected speedup for hybrid fault emulation over the serial fault emulation is 6.27 as discussed in Section IV-A.

TABLE VII
HYBRID FAULT EMULATION PERFORMANCE ESTIMATION

Circuit	R_s	P_m (%)	R_{HFE}
s298	87.119	0.149	69.25
s344	36.085	0.093	33.84
s382	25.048	12.586	3.49
s400	26.829	12.142	3.63
s444	32.145	12.437	3.62
s526	26.716	8.138	5.07
s713	37.185	0.009	36.94
s820	43.330	0.002	42.60
s832	44.957	0.001	44.91
s953	19.931	0.109	19.12
s1196	18.625	3.534	8.17
s1238	21.347	1.094	14.66
s1423	17.620	13.391	3.15
s1488	70.762	0.001	70.66
s1494	71.859	0.001	71.76
s5378	23.859	6.485	5.92
s9234	27.801	14.329	3.15
s13207	34.115	20.751	2.28
s38417	37.508	22.609	2.11
Average	36.991	6.729	6.27

VI. CONCLUDING REMARKS

We have shown that a hardware emulator can be used to boost the speed of fault simulation. We convert the original circuit into a fault-injectable circuit. Injecting faults is made easy by enabling the corresponding fault injection elements (FIE's) in the circuit. As opposed to previous methods [24], [25], we perform fault injection directly on the original circuit instead of the corresponding CLB, so all faults to be simulated can be stored in the fault-injectable circuit before the FPGA bit stream is generated, i.e., bitstream regeneration is not necessary, and the emulation time is greatly reduced. Moreover, our approach is independent of the underlying emulator architecture. The FIE's can be controlled by either the fault injection scan chain or the parallel fault injection selector. The parallel fault injection selector is more cost effective than the fault injection scan chain—the former's hardware overhead is about 25% lower than the latter. The hardware overhead factor (α) is about constant with respect to the circuit size because the extra circuitry required for the fault-injectable circuit is proportional to the number of faults emulated. Current commercial hardware emulators can handle designs up to about three million gates, so our fault emulator can handle circuit modules up to about 300K gates, assuming $\alpha = 10$. Beyond that, partition is required and reconfiguration time is increased. We can partition faults into F_g sets for fault injection so that the hardware overhead factor is reduced to α/F_g and the emulation capacity is increased to $F_g\times$ 300 K gates.

Note that F_g should be small to prevent the reconfiguration time from becoming the bottleneck. Experimental results show that our serial fault emulator is about 20 times faster than HOPE, and that higher speedup can be obtained for larger circuits.

We also proposed two hybrid fault emulation approaches: the first is faster, and the second has a lower hardware requirement as compared with serial fault emulation. In fault emulation, apart from the PO values, we also want to know as early as possible if the fault effect cannot be propagated to a PO. In that case, early screening of the faults that are not detected by the applied patterns can be done, and time can be saved. To eliminate the overhead from such faults and to reduce the number of faults to be emulated, we have used a logic simulator in our first hybrid fault emulation approach. We have shown that the performance overhead of the logic simulator is negligible for large circuits, and that the hybrid fault emulation is about six times faster than the serial fault emulation according to the time-complexity analysis and experimental results. As compared with the serial fault emulator, the hardware requirement and pin count slightly increase due to the modified FF's and parallel fault injection selector. In the second hybrid emulation approach, we have shown that if we simulate the normally small proportion of multiple-event faults using a fault simulator, we can greatly reduce the hardware requirement. However, this scheme is slower than serial emulation since the simulation time for multiple-event faults dominates when the circuit is large. Of course, this scheme is still much faster than conventional fault simulators since only multiple-event faults are simulated. Constructing experimental hybrid fault emulators is our future work.

Recently, some products have been developed, such as the Xilinx XC6200 chips, which can be partially reconfigured. However, fault injection by FPGA reconfiguration—even partial reconfiguration—is still not feasible since it is much slower than the pattern application rate. Using fault-injectable circuit to inject faults is more suitable for fault emulation. Also, internal values of the emulator usually cannot be obtained without a high cost. Identifying the faults that are not detected by the applied test patterns directly on the emulator will be much more efficient than on the simulator, and seeking efficient ways to do this also remains to be done in the future.

The limitation of our current fault emulator is that we are unable to handle potentially detected faults because it handles only two-valued logic. It can be modified to handle three-valued logic and potentially detected faults by using the dual-railed logic proposed in [24].

ACKNOWLEDGMENT

The authors would like to thank the Associate Editor and the anonymous referees for their valuable comments.

REFERENCES

[1] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*. New York: Computer Science Press, 1990.

[2] N. Gouders and R. Kaibel, "PARIS: A parallel pattern fault simulator for synchronous sequential circuits," in *Proc. IEEE Int. Conf. Computer-Aided Design (ICCAD)*, 1991, pp. 542–545.

[3] C.-P. Kung and C.-S. Lin, "Parallel sequence fault simulation for synchronous sequential circuits," in *Proc. European Design Automation Conference (EDAC)*, 1992, pp. 434–438.

[4] T. M. Niermann, W.-T. Cheng, and J. H. Patel, "PROOFS: A fast, memory efficient sequential circuit fault simulator," *IEEE Trans. Computer-Aided Design*, vol. 11, pp. 198–207, Feb. 1992.

[5] O. Y. Song and P. R. Menon, "3-valued trace-based fault simulation of synchronous sequential circuits," *IEEE Trans. Computer-Aided Design*, vol. 12, pp. 1419–1424, Sept. 1993.

[6] C. R. Graham, E. M. Rudnick, and J. H. Patel, "Dynamic fault grouping for PROOFS: A win for large sequential circuits," in *Proc. Int. Conf. VLSI Design*, Jan. 1997, pp. 495–501.

[7] H.-K. Lee and D.-S. Ha, "HOPE: An efficient parallel fault simulator for synchronous sequential circuits," in *Proc. IEEE/ACM Design Automation Conf. (DAC)*, 1992, pp. 336–340.

[8] ——, "New methods of improving parallel fault simulation in synchronous sequential circuits," in *Proc. IEEE Int. Conf. Computer-Aided Design (ICCAD)*, 1993, pp. 10–17.

[9] C.-P. Kung and C.-S. Lin, "HyHOPE: A fast fault simulator with efficient simulation of hypertrophic faults," in *Proc. IEEE Int. Conf. Computer-Aided Design (ICCAD)*, 1994, pp. 714–718.

[10] T. Markas, M. Royals, and N. Kanopoulos, "On distributed fault simulation," *IEEE Computer*, vol. 23, pp. 40–52, Jan. 1990.

[11] P. A. Duba, R. K. Roy, J. A. Abraham, and W. A. Rogers, "Fault simulation in a distributed environment," in *Proc. IEEE/ACM Design Automation Conf. (DAC)*, 1988, pp. 686–691.

[12] S. Parkes, P. Banerjee, and J. Patel, "A parallel algorithm for fault simulation based on PROOFS," in *Proc. Int. Conf. Computer Design (ICCD)*, 1995, pp. 616–621.

[13] M. B. Amin and B. Vinnakota, "ZAMBEZI: A parallel pattern parallel fault sequential circuit fault simulator," in *Proc. IEEE VLSI Test Symp. (VTS)*, 1996, pp. 438–443.

[14] ——, "Zamlog: A parallel algorithm for fault simulation based on Zambezi," in *Proc. IEEE Int. Conf. Computer-Aided Design (ICCAD)*, 1996, pp. 509–512.

[15] E. M. Rudnick and J. H. Patel, "Overcoming the serial logic simulation bottleneck in parallel fault simulation," in *Proc. Int. Conf. VLSI Design*, Jan. 1997, pp. 542–544.

[16] D. Krishnaswamy, E. M. Rudnick, J. H. Patel, and P. Banerjee, "SPITFIRE: Scalable parallel algorithms for test set partitioned fault simulation," in *Proc. IEEE VLSI Test Symp. (VTS)*, Apr. 1997, pp. 274–281.

[17] S. Walters, "Computer-aided prototyping for ASIC-based systems," *IEEE Design Test Comput.*, vol. 8, pp. 4–10, 1991.

[18] J. Varghese, M. Butts, and J. Batcheller, "An efficient logic emulation system," *IEEE Trans. VLSI Syst.*, vol. 1, pp. 171–174, June 1993.

[19] J. Dunlop, D. Girma, and P. Lysaght, "Alternative approach to ASIC design methodology based on reconfigurable logic devices," *Proc. Inst. Elect. Eng.*, vol. 139, pt. G, pp. 217–221, Apr. 1992.

[20] S. M. Trimberger, "A reprogrammable gate array and applications," *Proc. IEEE*, vol. 81, pp. 1030–1041, July 1993.

[21] Y.-L. Li and C.-W. Wu, "Cellular automata for efficient parallel logic and fault simulation," *IEEE Trans. Computer-Aided Design*, vol. 14, pp. 740–749, June 1995.

[22] Y.-L. Li, Y.-C. Lai, and C.-W. Wu, "VLSI design of a cellular-automata based logic and fault simulator," *Proc. Nat. Sci. Council Part A: Phys. Sci. Eng.*, vol. 21, pp. 189–199, May 1997.

[23] Xilinx, *The Programmable Gate Array Data Book*, Xilinx, Inc., San Jose, CA, 1996.

[24] K.-T. Cheng, S.-Y. Huang, and W.-J. Dai, "Fault emulation: A new approach to fault grading," in *Proc. IEEE Int. Conf. Computer-Aided Design (ICCAD)*, 1995, pp. 681–686.

[25] L. Burgun, F. Reblewski, G. Fenelon, J. Bariber, and O. Lepape, "Serial fault emulation," in *Proc. IEEE/ACM Design Automation Conf. (DAC)*, Las Vegas, NV, June 1996, pp. 801–806.

[26] J.-H. Hong, S.-A. Hwang, and C.-W. Wu, "An FPGA-based hardware emulator for fast fault emulation," in *Proc. Midwest Symp. Circuits Syst.*, Ames, IA, Aug. 1996.

[27] V. S. Iyengar and D. T. Tang, "On simulating faults in parallel," in *Proc. Int. Symp. Fault Tolerant Computing (FTCS)*, 1988, pp. 110–115.

[28] M. Abramovici, P. R. Menon, and D. T. Miller, "Critical path tracing—An alternative to fault simulation," in *Proc. IEEE/ACM Design Automation Conf. (DAC)*, 1983, pp. 214–220.

Shih-Arn Hwang (S'96–M'98) received the B.S. and Ph.D. degrees, both in electrical engineering, from National Tsing Hua University, Hsinchu, Taiwan, in 1994 and 1998, respectively.

His research interests include VLSI testing and design of high-performance application-specific VLSI circuits and systems. He is currently with the Taiwan Army for a two-year military service.

Jin-Hua Hong (S'98) received the B.S. degree in physical sciences in 1988 from National Central University, Chungli, Taiwan.

Since 1991, he has been with the Department of Electrical Engineering, National Tsing Hua University, Hsinchu, Taiwan, where he is currently a graduate student working toward the Ph.D. degree. His research interests include VLSI design and testing, with emphasis on design for testibility, boundary-scan-based testing, built-in self-test, and the hardware emulator.

Cheng-Wen Wu (S'86–M'87–SM'95) received the B.S.E.E. degree in 1981 from National Taiwan University, Taipei, Taiwan, and the M.S. and Ph.D. degrees, both in electrical and computer engineering, in 1985 and 1987, respectively, from the University of California, Santa Barbara.

From 1981 to 1983, he was an Ensign Instructor at the Chinese Naval Petty Officers' School of Communications and Electronics, Tsoying, Taiwan. From 1983 to 1984, he was with the Information Processing Center of the Bureau of Environmental Protection, Executive Yuan, Taipei, Taiwan. From 1985 to 1987, he was a Postgraduate Researcher at the Center for Computational Sciences and Engineering at UCSB. Since 1988, he has been with the Department of Electrical Engineering, National Tsing Hua University, Hsinchu, Taiwan, where he is currently a Professor. He also has served as the Director of the Computer and Communications Center of NTHU from February 1996 to February 1998.

Dr. Wu was the Technical Program Chair of the 5th IEEE Asian Test Symposium (ATS'96). He received the Teaching Award from NTHU in 1996, and the Outstanding Electrical Engineering Professor Award from the Chinese Institute of Electrical Engineers (CIEE) in 1997. He is interested in the relationship between architectures and algorithms with respect to the design and testing of high-performance VLSI circuits and systems. He is a member of CIEE.