Simulation of embedded memories by defective hashing

by Leendert M. Huisman

Because logic designs are becoming more complex and extensive, they increasingly tend to contain embedded memories. In the simulation (particularly fault simulation) of these designs, the embedded memories may be found to require large amounts of storage unless a carefully designed simulation strategy is adopted. This paper describes a technique that drastically reduces the storage required in the fault simulation of such large designs. The required amount of storage can be fixed at compile time or at load time, and can almost always be made to fit in the available storage at the cost of only a small decrease in the predicted exposure probabilities.

Introduction

During fault simulation in the combinational logic of designs containing embedded memories, a fault in the combinational logic may cause the input to the write port or the read port of some memory to be incorrect. The contents of that memory may become corrupted when the write port is affected; i.e., the contents may diverge from those of the memory in the fault-free design. How they diverge depends on the details of the fault effect on the write port: Sometimes only one address is affected, sometimes two [1]. An incorrect read address or an incorrect read enable may produce faulty data in the read buffer, although it does not corrupt the contents of the memory.

A fault that affects the write port of the memory may be exposed if the contents of the memory become incorrect and are later read. Since it is not known beforehand when such incorrect data will be written into or read from the memory, a fault simulator must maintain a complete copy of the faulty design's memory, in addition to the copy for the fault-free design. When only the read port of a memory can be affected by a fault, incorrect data may still appear in the read buffer and cause an exposure of the fault, but this will not affect the contents of the memory. Therefore, no separate copy of the memory is required for such a fault.

In many fault-simulation algorithms [2], such as deductive simulation or concurrent simulation, all faults are processed simultaneously to take advantage of the logical similarity of most faulty designs: For many input patterns, faults will produce no fault effects at all, or will produce fault effects only in the immediate neighborhood

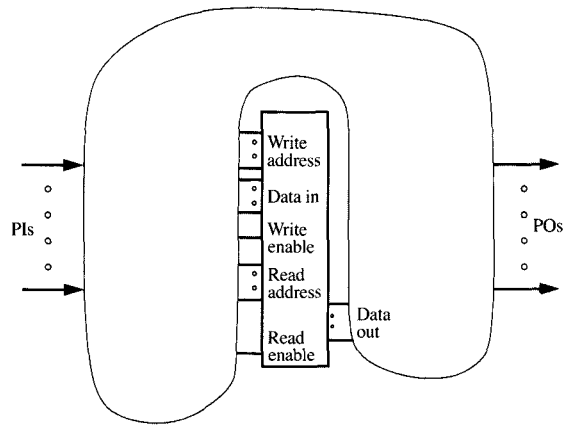

Figure 1

Two-port memory configuration.

of the fault location. Most of the time, the logical values on the internal nets of a faulty design are the same as those in the fault-free design, and there is therefore no reason to simulate the faulty design explicitly. When all faults are simulated simultaneously, however, each fault that *may* affect the write-port of some internal memory in general *will*, if sufficiently many patterns are applied, but we do not know how and when a particular fault will corrupt the memory. Therefore, to properly simulate all faulty designs, multiple copies of each embedded memory may have to be maintained; each memory whose write port(s) can be affected by N different faults requires $N + 1$ copies: N for the faulty designs and 1 for the fault-free design.

The storage required for copies of large memories can easily exceed the available storage in the host machine. This can be and often is solved by partitioning the fault list and simulating only one (small) portion at a time. This reduces the memory requirements but also greatly increases the run time. Far less storage would be required by storing only the addresses containing incorrect data and the incorrect data themselves. On a read from a memory in a faulty design, one then checks first to see whether there is a record for the read address in that memory, indicating that that address contains corrupted data. If there is no such record, the data are simply read from the good machine's copy of that memory.

The required storage may be considerably less than when complete memories are stored. One still needs a storage pool to accommodate the incorrect data in any of the memories. Allocation of the incorrect data to the storage pool can be done by hashing [3, 4]. The key to be hashed contains fields, the first of which is the address in the memory. The second is the name of the memory, or some index, since there may be more than one memory in the design. The third and last field in the key is the name of the fault for which the memory simulation is done. Hashing has the advantage that locating a record can be done in constant time; other techniques for allocating records to the storage pool, such as binary trees, slow the simulation unacceptably.

However, with hashing there is still the problem that the location in the hash table where (key + data) should be stored is already occupied. This is called a collision. Standard strategies to deal with such collisions are rehashing and linked lists [3]. Resolving the collisions tends to slow down the simulation process and may still result in an overflow of the available memory during run time. To solve the problem of hash-table overflow, and also to avoid the performance degradation associated with the collision handling, a new technique, called defective hashing, is proposed. This technique makes it possible to simulate all fault effects at the inputs to embedded memories using only the available storage.

The next section briefly describes how to simulate a memory when a fault effect arrives at a write port or a read port. In the final section, defective hashing and various implementation details are discussed.

Embedded memories

This section reviews how faults can affect the inputs to and the contents of a memory, and how these fault effects are simulated, using hash tables to store records for faulty addresses. Summary, and sketchy, C programs are presented in Figures 2 and 3 for the write operation and the read operation, respectively. Completeness is not claimed for these programs. Thorny problems, such as how the memory is initialized and what logical values appear in the read buffer when the read enable is off, are not discussed. The main goal of this section is to show that all fault effects can be handled systematically and to focus attention on the few primitive operations that work directly with the hash table.

- *Embedding of the memory*

The memories considered in this paper are assumed to have two ports (see **Figure 1**), although the ideas expressed here are applicable to general multiport memories. There are two separate ports, one for writing and one for reading, each with its own address lines and enable line. Fault effects can arrive at the memory at the data lines of the write port and at any of the addresses or enables. We assume no restrictions on the combinational

logic in front of the memory. For example, on the same input pattern a single fault can give rise to fault effects on the data lines, address lines, and enable line of the write port.

In the concurrent simulation of many faulty designs, a memory is simulated explicitly and serially (i.e., one faulty design at a time) for all faulty designs that have a fault effect on any of the inputs to the memory or in the contents of the memory. The simulator handles the memory by two separate simulation operations: a read and a write. There must be an order of precedence for the two operations if both read and write ports are enabled and both operations refer to the same address: Either the write occurs first and the read produces the newly written data, or the read occurs first and produces stale data. The choice of priority has no effect on the simulation of the reads and writes, however, as long as they are done in the appropriate order.

• *Fault effects*

Faults can affect the memory in many different ways. When a fault affects the read operation, the memory is simulated as if the design were fault-free, but with perhaps a different enable or address than in the actual fault-free design. The write operation, however, is affected by faults in a very complex manner.

Table 1 summarizes the ways in which a single fault can affect the write port of a memory. To simplify the text somewhat, the simulated fault-free design is indicated from now on by *g.machine* and the simulated faulty design for some fault by *f.machine*. The columns in Table 1 are labeled according to the four different g.machine/f.machine combinations of the write enables. In the column labeled *0/0*, the write enables of both the fault-free design and the faulty design are off. Clearly, in that case, no write occurs in either the g.machine or the f.machine, and therefore no fault effect (F.E.) is written into the memory.

In the *0/1* column, the write enable of the f.machine is incorrectly on. Data are written to some address that may or may not be the same as the g.machine's address, though data should not have been written at all. This will corrupt the contents of the memory, unless by coincidence the f.machine's data happen to be the same as the data in the g.machine's memory at the f.machine's address. Then the faulty write to the f.machine's memory will in fact overwrite and cancel a previously stored fault effect. We ignore this exception at the cost of a small potential increase in required storage and assume that the memory contents will always be corrupted.

The case in which the write enable is incorrectly off (the *1/0* column) has a similar effect on the contents of the memory. In this case, no write occurs to the f.machine's memory, though a write should have occurred (to the g.machine's address in the faulty design's memory). If the g.machine's address of the faulty design's memory already contains incorrect data, these data will not be affected, and the corresponding record in the hash table will stay. However, if there are no incorrect data at that address, there is no corresponding entry in the hash table, and an explicit write *to the hash table* (not to the f.machine's memory) is required: The f.machine's memory will differ from that of the g.machine, because it will contain the old data from the g.machine, while the g.machine itself contains new data. Again, ignoring the unlikely case in which the data at the affected address are the same as the write data in the g.machine, this lack of a write will always corrupt the f.machine's memory.

Table 1 Fault effects on the write port of a memory

Address/Data	*Enable (g/f)*			
	0/0	*0/1*	*1/0*	*1/1*
Good/Good	No F.E.	F.E. in f.a.	F.E. in g.a.	No F.E.
Good/Faulty	No F.E.	F.E. in f.a.	F.E. in g.a.	F.E. in f.a.
Faulty/Good	No F.E.	F.E. in f.a.	F.E. in g.a.	F.E. in g/f.a.
Faulty/Faulty	No F.E.	F.E. in f.a.	F.E. in g.a.	F.E. in g/f.a.

g/f = g.machine's/f.machine's
F.E. = fault effect
g.a. = g.machine's address in the f.machine's memory
f.a. = f.machine's address
g/f.a. = both the g.machine's and the f.machine's addresses in the f.machine's memory

The *1/1* column treats the most complex case, in which both write enables are on. The rows are labeled with the various possibilities (good or faulty) of the data and the address. When both addresses and data are correct, the f.machine will end up with correct data, i.e., no entry in the hash table, at the g.machine's address. If a record had existed for the f.machine at that address, it would be removed from the hash table. When the fault causes the data at the write port to be incorrect while the address remains valid (row 2), incorrect data are stored in the f.machine's memory at the same address as in the g.machine's memory. Finally, when the write address is faulty (rows 3 and 4), a double write must be performed into the f.machine's memory [1]. First, data are written to an address to which they should not have been written. Second, a write occurs in the g.machine that is not paralleled in the faulty design. Consequently, at the completion of the actual write operation, the f.machine's memory will also contain incorrect data at the g.machine's address, namely the data that were there before the write occurred rather than the data that appeared at the write port.

A complete program written in pseudo-C is shown in **Figure 2**. In this and later program fragments, we assume that the memory simulation is done by hashing (defective

```
int rcf, rcfg;
unsigned fault_descriptor;

switch (fault_descriptor)  {

   case 1: case 5: case 9: case 13:          /*                    Column 2 */
   case 7:                                   /*                 Column 4, row 2 */
      rcf = locate_record(key(f. machine))
      write_record(key(f.machine),data);
      break;

   case 2: case 6: case 10: case 14:         /*                    Column 3 */
      rcfg = locate_record(key(f. machine at g.machine's address))
      if (rcfg ! = RECORD_FOUND)
         copy(g.machine data to g.machine's address in f.machine)
      break;

   case 3:                                   /*                 Column 4, row 1 */
      rcf = locate_record(key(f.machine))
      IF (rcf == RECORD_FOUND) remove_record(key(f.machine));
      break;

   case 11: case 15:                         /*           Column 4, rows 3 and 4 */
      rcf = locate_record(key(f.machine))
      write_record(key(f.machine),data);
      rcfg = locate_record(key(f.machine at g.machine's address))
      if (rcfg ! = RECORD_FOUND)
         copy(g.machine data to g.machine's address in f.machine)
      break;

   default:                                  /*                    Column 1 */
      break;
   }
```

Figure 2

Write operation to a memory.

or otherwise). The fault_descriptor variable encodes in some form the sixteen various possibilities of Table 1. In the variation shown here, the elements of the matrix in Table 1 are numbered row-wise from 0 to 15. The function locate_record(key) locates a record in the hash table and returns a variable that indicates whether a record that matches the key was found (RECORD_FOUND) or not (NO_RECORD). Hidden in this function is the algorithm used to handle collisions; key is a structure that contains the information required to store an item in the hash table: address, memory name, and fault name. This structure upon return from locate_record() also contains a record address in the hash table, either of the existing record or of a blank record where the new data will be written. The function write_record(key,data) performs a single write of data into the hash table at the location indicated by key. Consequently, the case corresponding to column *1/1*, rows 3 and 4, needs two invocations of write_record(). The function remove_record(key) removes a record if one is there.

As mentioned above, a read operation is considerably simpler than a write operation. The pseudo-C program for reading from a memory is shown in **Figure 3**.

Defective hashing

The programs shown above assume a complete hash-table storage technique for storing the corrupted addresses. The locate_record, write_record, and read_record functions introduced above are functions that work on the hash table. However, as explained in the Introduction, adding records may eventually lead to local memory overflow and in any case introduces the problem, and performance degradation, of collision handling. In this section, a technique is introduced that obviates the need for collision handling, making it possible to do the simulation with almost any amount of local memory without fear of overflow.

The key idea is that in many applications fault simulation is done to establish that faults are exposed by a test *sequence*, i.e., by at least one of the patterns in the sequence. In such applications it is clearly not necessary that all exposures be identified, but only that at least one be identified. In fact, it is often not even necessary to determine all faults that are exposed, only a sufficiently large fraction. It is therefore possible to relax the fault-simulation requirements and require only that when the fault simulator identifies a pattern as one exposing a

```
int rcf, rcfg;

if (read_enable is off)
   rcf = ENABLE_OFF
else
   rcf = locate_record(key(f.machine))

switch (rcf) {

   case RECORD_FOUND:                /*              read from the hash table */
      copy (f.machine data to read buffer)
      break;

   case NO_RECORD:                   /*              read from good machine */
      rcfg = locate_record(key(g.machine at f.machine's address))
      copy(g.machine data at f.machine's address to read buffer)
      break;

   default:
      break;
   }
```

Figure 3

Read operation from a memory.

certain fault, (a) this fault must be exposed by that pattern when the real test sequence is applied to the real hardware, and (b) this fault must be present in the real hardware. In other words, in many applications a fault simulator is allowed to be pessimistic as long as it is never optimistic. A similar reasoning lies behind the acceptability of critical path tracing [5], which is an approximate but strictly pessimistic fault simulator. Of course, once we allow pessimism, the degree of acceptable pessimism becomes a matter of concern, but we do not address that here.

If we allow a fault simulator to be pessimistic, all the problems with collisions and hash-table overflow can be made to disappear. The general strategy is simply to overwrite on collision, i.e., to use a new item to replace an item that is already in a location. This has the disadvantage that possible exposures are sometimes missed when incorrect data that were stored in a memory are overwritten by other incorrect data that corrupted some other address (perhaps even belonging to an entirely different memory) but were stored in the same location in the hash-table. Such an overwrite may therefore reduce the calculated exposure probability of the fault. This pessimism can be minimized by using as large a storage pool as is feasible and by using a hashing technique that will assign the keys as evenly as possible to the locations in the storage pool [6].

Of course, the g.machine's memory now cannot be simulated using the same hash table, because its data are always needed and we cannot run the risk of overwriting those data. The machine's memories must therefore be simulated using separate storage areas and no defective hashing mechanisms.

This simple idea of overwriting upon collision must be refined, because memories can be embedded in many different ways in combinational logic, and not all embeddings are amenable to simple overwrites. Starting from the fault location, fault effects may fan out, pass through the memory via several different paths, or pass around the memory in addition to passing through it, and reconverge behind that memory. It might then happen that during the actual test a fault effect would follow a path around the memory, or through the memory, and reconverge with another fault effect that originated from the same fault and that was stored previously in the memory. If this reconvergence should lead to a mutual cancellation of the fault effects, no exposure would be observed on the POs for that pattern. However, if we had performed fault simulation with unprotected overwrites, the fault effect stored in the memory during the simulation might have been overwritten by another fault effect with a different key. As a result, the fault effect coming around the memory would reconverge in the simulation with a g.machine value rather than with another fault effect, and the simulator might indicate, incorrectly, that the fault was exposed at the POs.

We discuss a number of strategies of increasing complexity. These strategies differ in the presence or absence of some indication that an overwrite occurred.

Such an indication may be required to ensure that the simulation does not indicate an exposure when there is none, as explained above. Since the presence of such an indication increases the storage required for the memory simulation without increasing the calculated exposure probability, it should be avoided whenever possible.

• *Defective hashing with reconvergence*

In the simplest case, no indication will show that an overwrite occurred. This technique can be used when there is no danger of reconvergence from the fault site before the memory to somewhere behind the memory. Clearly, the presence or absence of such a reconvergence depends on the fault location with respect to the memory and on how the memory is embedded. The same memory therefore may be simulated without an overwrite indication for some faults, but with one for other faults.

The specifics in this case are simple; in fact, the programs shown in Figures 2 and 3 can be used. The locate_record(key) function hashes the key only once (i.e., without attempting to resolve a collision if one occurs) and returns 1) the resulting location in the hash table and 2) an indication whether or not a record with a matching key was found. If no record with a matching key was found, the location is assumed to be empty (rcf = NO_RECORD), even if a record is there. On a write to an f.machine's memory, (key + data) are written at the location that was found by locate_record(), regardless of what was there already. On a read, the data from the hash table are used if there is a match between the key for the read operation and the stored key in the hash table. Otherwise, the f.machine's memory is considered to contain no corrupted data at the read address, and the data are taken from the g.machine's memory instead.

• *Degree of pessimism*

To get some feeling for how pessimistic fault simulation can become when the defective hashing technique is used, we consider a simple example. In this example, a random pattern sequence is applied to a logic design with one embedded memory and no reconvergence around the memory. This memory has I addresses, and the addresses are randomly selected with equal probability (equal to $1/I$) during the test. The logic design is such that only faults on the data lines can affect the memory. There are F such faults, with exposure probabilities $p_1, p_2, \cdots, P_F$. A fault effect is written into the memory when it appears at the write port and the write enable (indicated by w) happens to be on. Similarly, a fault effect is read from the memory when the read enable happens to be on and a fault effect was previously written into the address from which the read occurs.

While a fault effect is present in some address, it may be overwritten by various mechanisms, though not all of them are caused by defective hashing. Random testing is a case in which defective hashing can lead to very pessimistic results, because a fault effect may have to remain a long time in the memory before it is read. Clearly, defective hashing reduces the average stay of a fault effect in the memory because of the additional overwriting mechanisms. One would like to estimate how severe this reduction is, and whether or not the stay will be reduced to 0.

The hash table contains H bins. Because there is only one memory, there are $F*I$ items [(fault,address) pairs] to be hashed, and on average each bin contains $F*I/H$ of them. The case in which $F*I$ is not large compared to H is not interesting, as it is the case in which the hash table is (almost) large enough to hold all items without collision. We therefore assume that both F and I are large compared to 1 and that their product is large compared to H. In fact, we assume that I is itself large compared to H, because we would like to illustrate the effects of defective hashing in its most serious form. For every f.machine, the addresses will be spread more or less evenly over the bins and, as a result, on average I/H addresses will be assigned to each bin. As the total number of (fault,address) pairs mapped in any bin is on average $F*I/H$, all faults will be represented in each bin.

Consider now a particular fault f with exposure probability p_f, and assume that a fault effect caused by this fault has been written in address i in the memory. The pair (f,i) is mapped to a certain bin, called h, in the hash table. The fault effect can be overwritten on a later pattern in three different ways. First, the f.machine can write fault-free data to address i. This overwrites the fault effect even without the defective hashing. The probability of such an overwrite occurring is

$$\frac{w}{I}(1 - p_f),$$

where the factor w/I is the probability of writing to address i. Second, a fault effect can be written by the same f.machine to a different address that is mapped to the same bin h. On average, I/H different addresses from that f.machine will be mapped to h. The probability of writing a fault effect to any of those addresses except i is

$$\frac{w}{I} p_f\left(\frac{I}{H} - 1\right) \simeq \frac{w}{H} p_f.$$

These first two overwrite mechanisms are mutually exclusive. Therefore, the probability that neither one of them caused an overwrite on one pattern can be approximated by

$$1 - \frac{w}{I} - \frac{w}{H} p_f .$$

Finally, the fault effect can also be overwritten by different f.machines that write to the addresses for which the corresponding (fault,address) pairs are mapped to bin h. For every such f.machine, I/H different addresses will use bin h on average. The probability of not writing a fault effect to any of the (fault,address) pairs in bin h is then (assuming I/H to be large)

$$\prod_{\substack{j=1 \\ j \neq f}}^{F} \left(1 - \frac{w}{I} p_j \frac{I}{H}\right) = \prod_{\substack{j=1 \\ j \neq f}}^{F} \left(1 - \frac{w}{H} p_j\right) \simeq \frac{e^{-\frac{w}{H}\langle p \rangle F}}{1 - \frac{w}{H} p_f},$$

where $\langle p \rangle$ is the average fault-exposure probability.

The conclusion is that once a fault effect in some f.machine has been written to some address, the probability that it will not be overwritten is

$$\frac{\left(1 - \frac{w}{I} - \frac{w}{H} p_f\right)}{\left(1 - \frac{w}{H} p_f\right)} e^{-\frac{w}{H}\langle p \rangle F} \simeq \left(1 - \frac{w}{I}\right) e^{-\frac{w}{H}\langle p \rangle F}.$$

Fault effects can also be overwritten as soon as they are written, i.e., on the same clock cycle. The probability of such an overwrite occurring depends on when the corresponding f.machine is handled by the fault simulator; it is largest when the f.machine is first in the list. The first two overwrite mechanisms do not apply, because they are in contradiction with a fault effect being written. The third one does occur. When the f.machine in question is the first one to be considered, the probability of the other f.machines not overwriting the fault effect is the same as above, i.e., a simple exponential factor. Therefore, the probability of a fault effect being written without being overwritten is

$$\frac{w}{I} p_f \frac{e^{-\frac{w}{H}\langle p \rangle F}}{1 - \frac{w}{H} p_f},$$

$$\frac{w}{I} p_f e^{-\frac{w}{H}\langle p \rangle F}.$$

The most important difference from the exact fault simulation, i.e., no defective hashing, is the exponential factor. Interestingly, the exponential factor does not depend on the size of the memory if this size is large. It depends mainly on the ratio of F and H, showing that for defective hashing not to be too pessimistic, the number of bins in the hash table should not be small compared to the number of faults that can affect the memory. A more accurate estimate of the effects of defective hashing is obtained by comparing the exponential factor to its coefficient. This coefficient is present even without defective hashing. Strictly speaking, there is an additional term proportional to p_f in this coefficient, but that term can be ignored when p_f is small. The comparison shows that the effects of defective hashing can be ignored when H/FI is large compared to $\langle p \rangle$. Since we assume that FI/H is large, this condition may not be fulfilled, and defective hashing may have a nonnegligible effect.

Fault exposures will not be eliminated by the defective hashing scheme, however; they will only be diminished. Moreover, if faults are dropped from consideration when they are first exposed, $\langle p \rangle$ will decrease and H/FI will eventually become large compared to $\langle p \rangle$. In the limit that only the hard-to-detect faults remain, $\langle p \rangle$ will be very small, and the exponential factor will have almost no effect.

Defective hashing with reconvergence

When there is reconvergence from the fault site before the memory to behind the memory, some record must be kept that the location in which (key + data) was stored became polluted, i.e., that an exposing vector was overwritten. The easiest way to indicate that an overwrite occurred is to add a so-called pollution bit to the record for the location in the hash table. This bit is set as soon as the overwrite occurs. Once the bit has been set, however, it cannot be reset, because all information on the data and the address whose record was overwritten has been lost. Therefore, it is impossible to determine whether a later write to that address has canceled the fault effect, or even that a fault effect existed in that address in the first place.

The pseudo-C code for the write operation is shown in **Figure 4**. It is basically the same as the code in Figure 2, but with some additions to handle the pollution bit. In particular, locate_record() now also indicates whether a record with a nonmatching key was found at the location where the write data will be stored (OTHER_RECORD_FOUND), because the pollution bit only has to be set when an actual existing record is overwritten. set_pollution_bit(key) is the function that sets the pollution bit of the location in the hash table indicated by key.

When on a read it turns out that (a) the location in the hash table returned by locate_record() is polluted and (b) there is no match between the keys (rcf = = OTHER_RECORD_FOUND), any exposure of a fault during this pattern is ignored. The reason is that the present data in this location might have overwritten a fault effect that was stored previously in the memory at the address being read, and that this overwritten fault effect might have

```
int rcf, rcfg;
unsigned fault_descriptor;

switch (fault_descriptor) {

   case 1: case 5: case 9: case 13:          /*                          Column 2 */
   case 7:                                   /*                  Column 4, row 2 */
      rcf = locate_record(key(f.machine));
      write_record(key(f.machine),data);
      IF (rcf == OTHER_RECORD_FOUND) set_pollution_bit(key(f.machine));
      break;

   case 2: case 6: case 10: case 14:         /*                          Column 3 */
      rcfg = locate_record(key(f.machine at g.machine's address));
      if (rcfg ! = RECORD_FOUND)
         copy(g.machine data to g.machine's address in f.machine);
      IF (rcfg == OTHER_RECORD_FOUND)
         set_pollution_bit(key(f.machine at g.machine's address));
      break;

   case 3:                                   /*                  Column 4, row 1 */
      rcf = locate_record(key(f.machine));
      IF (rcf == RECORD_FOUND) remove_record(key(f.machine));
      break;

   case 11: case 15:                         /*          Column 4, rows 3 and 4 */
      rcf = locate_record(key(f.machine))
      write_record(key(f.machine),data);
      IF (rcf == OTHER_RECORD_FOUND) set_pollution_bit(key(f.machine));
      rcfg = locate_record(key(f.machine at g.machine's address));
      if (rcfg ! = RECORD_FOUND)
         copy(g.machine data to g.machine's address in f.machine);
      IF (rcfg == OTHER_RECORD_FOUND)
         set_pollution_bit(key(f.machine at g.machine's address));
      break;

   default:                                  /*                          Column 1 */
      break;
}
```

Figure 4

Write operation to a memory with pollution bit.

interfered destructively with other fault effects coming around the memory or through the memory via some other path. In all other cases, the read operation works as if there were no pollution bit. The extension of the code shown in Figure 3 for the read operation is straightforward and is not shown separately.

With a simple pollution bit, the exposure predictions are always pessimistic. They become more pessimistic over time because pollution bits cannot be reset once they are set. Such increasing pessimism is not a problem when few patterns are simulated. However, in long test sequences all exposures must eventually be ignored because of the set pollution bits.

This problem of increasing pessimism can be solved, though at some storage cost, by having not one pollution bit per location in the hash table, but instead one pollution bit per address for every memory in every f.machine where reconvergence necessitates some indication that an overwrite occurred. The idea is to replace each memory with a pollution array which has the same number of addresses but is only one bit wide. If a record in the hash table is overwritten, the information about the *data* in the memory is lost, but not the address in the memory and not the fact that there used to be corrupted data at that address.

This technique is generally applicable, but it clearly does not save storage space when the memories themselves are very narrow. In addition, the pollution arrays may not fit in storage if storage is sufficiently small, and their use should therefore be restricted as much as possible. It might be noted, however, that it is not always necessary to set aside in storage one bit for every address in the memory. If few addresses are affected during a simulation, for example because the number of patterns that must be simulated is small or because the fault is hard to expose, the pollution-array representation in storage can be considerably compressed because the *set* bits in the array are sparse.

A bit in the pollution array is set if the corresponding address in the memory contains a fault effect and reset

```
int rcf, rcfg;
unsigned fault_descriptor;

switch (fault_descriptor) {

   case 1: case 5: case 9: case 13:          /*                    Column 2 */
   case 7:                                   /*                 Column 4, row 2 */
      rcf = locate_record(key(f.machine));
      write_record(key(f.machine),data);
      change_pollution_array(key(f.machine),1);
      break;

   case 2: case 6: case 10: case 14:         /*                    Column 3 */
      rcfg = locate_record(key(f.machine at g.machine's address));
      if (rcfg ! = RECORD_FOUND)
         copy(g.machine data to g.machine's address in f.machine);
      change_pollution_array(key(f.machine at g.machine's address),1);
      break;

   case 3:                                   /*                 Column 4, row 1 */
      rcf = locate_record(key(f.machine));
      IF (rcf = = RECORD_FOUND) remove_record(key(f.machine));
      change_pollution_array(key(f.machine at g.machine's address),0);
      break;

   case 11: case 15:                         /*           Column 4, rows 3 and 4 */
      rcf = locate_record(key(f.machine))
      write_record(key(f.machine),data);
      change_pollution_array(key(f.machine),1);
      rcfg = locate_record(key(f.machine at g.machine's address));
      if (rcfg ! = RECORD_FOUND)
         copy(g.machine data to g.machine's address in f.machine);
      change_pollution_array(key(f.machine at g.machine's address),1);
      break;

   default:                                  /*                    Column 1 */
      break;
   }
```

Figure 5

Write operation to a memory with pollution arrays.

when the exposing vector is overwritten by a nonexposing one. It is this possibility of resetting the pollution bit that prevents the simulation from becoming increasingly pessimistic. In the hash table itself, there can be overwriting as above. The C program for the write operation is shown in **Figure 5**. The setting and resetting of pollution bits is done by change_pollution_bit(key,value), where value is either 1, corresponding to a set, or 0, corresponding to a reset.

The program for a read is shown in **Figure 6**. The hash table must be consulted on a read when the pollution bit in the pollution array is set. The pollution array is checked by the function read_pollution_bit(key), which returns the value of the pollution bit. When the pollution bit is set to 1, the f.machine's memory contains corrupted data at the f.machine's address. These data may of course have been overwritten, but a mismatch or match between the keys indicates whether an overwrite did or did not occur. In the former case, locate_record() returns the value OTHER_RECORD_FOUND; in the latter case it returns RECORD_FOUND. As before, on a mismatch, any exposure of the fault should be ignored for this pattern.

Conclusion

Defective hashing makes it possible to handle many memories in a fault simulator, usually within the constraint of existing storage. The cost is a possible decrease in the calculated test coverage. This decrease is under the control of the user, who still has the option of obtaining more storage or partitioning the fault list. Only when the test sequences are long and the memories are embedded in such a complex fashion that many fault effects may follow reconvergent paths through or around the memory will the simple versions of this technique fail. Even then, however, the required storage can be reduced by using pollution arrays, which are only one bit wide. In addition, the representation of these arrays may be compressed if the array elements that have the set value are sparse.

```
int rcf, rcfg;

if (read_enable is off)
   rcf = ENABLE_OFF
else {
   p_bit = read_pollution_array(key(f.machine))
   if (p_bit == 0)
      rcf = NO_RECORD
   else
      rcf = locate_record(key(f.machine))
   }                                        /*                    read enable is on */

switch (rcf) {

   case RECORD_FOUND:                       /*                read from the hash table */
      copy (f.machine data to read buffer)
      break;

   case NO_RECORD:                          /*                read from good machine */
      rcfg = locate_record(key(g.machine at f.machine's address))
      copy(g.machine data at f.machine's address to read buffer)
      break;

   case OTHER_RECORD_FOUND:                 /*                      ignore this pattern */
      /* return and "IGNORE THIS PATTERN INDICATION"
      break;

   default:
      break;

   }
```

Figure 6

Read operation from a memory with pollution arrays.

Acknowledgment

The author wishes to thank Barry Rosen and Larry Carter of the IBM Thomas J. Watson Research Center for their critical reading of the original manuscript.

References

1. Paul H. Bardell, William H. McAnney, and Jacob Savir, *Built-In Test for VLSI: Pseudorandom Techniques*, John Wiley & Sons, Inc., New York, 1987, Ch. 7.11.12.
2. Alexander Miczo, *Digital Logic Testing and Simulation*, Harper & Row Publishers, New York, 1986, Ch. 4.8.
3. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *Data Structures and Algorithms*, Addison-Wesley Publishing Co., Inc., Reading, MA, 1983, Ch. 4.7.
4. Ted G. Lewis and Curtis R. Cook, "Hashing for Dynamic and Static Internal Tables," *Computer*, pp. 45–56 (October 1988).
5. Miron Abramovici, P. R. Menon, and David T. Miller, "Critical Path Tracing: An Alternative to Fault Simulation," *IEEE Design & Test of Computers* **1,** 83–92 (February 1984).
6. J. Lawrence Carter and Mark N. Wegman, "Universal Classes of Hash Functions," *J. Computer & Syst. Sci.* **18,** 143–154 (April 1979).

Received June 29, 1989; accepted for publication August 25, 1989

Leendert M. Huisman *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Huisman received the Ir. degree in technical physics from the Technische Hoge School, Delft, the Netherlands, in 1973. He received a Ph.D. degree in physics from Harvard University in 1979. From 1978 to 1980 Dr. Huisman was a Postdoctoral Research Fellow at Brandeis University, Waltham, Massachusetts. Between 1980 and 1982, he was employed by the Stichting voor Fundamenteel Onderzoek der Materie in the Netherlands. Since joining IBM in 1982, he has worked on VLSI testing and verification. Dr. Huisman is a member of the American Physical Society and the IEEE Computer Society.