May 1995

UILU-ENG-95-2216
CRHC-95-12

Center for Reliable and High Performance Computing

Implementation Consideration for List Splitting, Sequential, and Compact Fault Dictionary Compression Techniques

Ching Tai

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) (CRHC-95-12)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (*If applicable*) N/A	7a. NAME OF MONITORING ORGANIZATION N/A
6c. ADDRESS (*City, State, and ZIP Code*) 1308 W. Main St. Urbana, IL 61801	7b. ADDRESS (*City, State, and ZIP Code*)
8a. NAME OF FUNDING / SPONSORING ORGANIZATION N/A 8b. OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (*City, State, and ZIP Code*)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (*Include Security Classification*)
Implementation Considerations for List Splitting, Sequential, and Compact Fault Dictionary Compression Techniques

12. PERSONAL AUTHOR(S)
Ching Tai

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
Technical	FROM _____ TO _____	May 1995	41

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	fault dictionary, compression, list splitting, sequential, compact, implementation

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

A full fault dictionary enumerates all points of observation and their corresponding fault lists in either a matrix or a list-of-faults format. A point of observation is a test vector and primary output combination. Such dictionaries are generally impractical and large for a large sequential circuit. The large size of the full dictionary creates storage and usage problems. Fault dictionary compression algorithms attempt to solve the size problem of the full fault dictionary by using heuristics to eliminate from inclusion into the compressed dictionary certain points of observation information. Each point of observation has the possibility of detecting, not detecting, or potentially detecting a fault. A fault is consider "detected" by a point of observation if a circuit containing the fault gives a different logic value than the good (fault-free) circuit at that point of observation. A fault is "not detected" (or "undetected") if the faulty circuit gives the same value as the good circuit. If a faulty sequential circuit may potentially give a different value than a good circuit, then it is "potentially detected".

(OVER)

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (*Include Area Code*) / 22c. OFFICE SYMBOL

Abstract Cont.

A full fault dictionary lists all the points of observation of a circuit. For the matrix format, the full dictionary includes a column for each point of observation and a row for each fault. The intersections of the rows and columns of the matrix are marked by a "D", "U", or "P" representing "detected", "undetected", or "potentially detected", respectively, to denote the detection characteristics of the particular fault and point of observation combination. The list-of-faults formatted dictionary includes a row for each point of observation and a list of the detected faults for that point of observation. Potentially detected faults are listed in separately.

A dictionary compression technique attempts to reduce the full dictionary's size by using heuristics to eliminate the inclusion of certain points of observation. Each elimination of a point of observation is manifested as a deletion of a column in a matrix-formatted full fault dictionary or a deletion of a row in a list-of-faults formatted full fault dictionary.

Compression techniques may make tradeoffs in the compressed dictionary's diagnostic resolution to gain better compression, execution time, etc.. The diagnostic resolution of a fault dictionary is the measure of that dictionary's capability to distinguish its faults among each other, and the full fault dictionary has the maximum diagnostic resolution. The diagnostic resolution is based on the set of test vectors and fault model used by a dictionary; however, since the compressed dictionaries are generated from only information contained in the full dictionary (or are generated with access to the same test vectors and fault model as the full dictionary), the full dictionary's resolution sets an upper-bound on the corresponding compressed dictionary.

The test vectors and fault model used have a direct impact on the size of the full dictionary and the heuristic performance of the compression techniques. All test vectors used for the ISCAS89 circuits in this paper are vectors generated by the HITEC program [3] and simulated using the PROOFS sequential circuit fault simulation program [4]. The fault model used is the single stuck-at model with equivalence fault collapsing. Only faults that are definitely detected at least once are included in the initial fault list [9], [10].

Three dictionary compression techniques are implemented and examined. The examination focuses mainly on the resulting programs' asymptotic efficiencies in terms of run time requirements and memory usage; however attention is also given to the multiplicative constants and lower-order terms when applicable. The execution time and memory requirements of each compression technique depends on that technique's predefined heuristic and on the implementation's chosen internal data representation.

This thesis also provides a collection of empirical data on memory usage and run time requirements for some ISCAS89 circuits tested. Similarities among the three techniques, observed through their implementations, are also mentioned during each compression technique's implementation discussion.

Abstract Cont.

A full fault dictionary lists all the points of observation of a circuit. For the matrix format, the full dictionary includes a column for each point of observation and a row for each fault. The intersections of the rows and columns of the matrix are marked by a "D", "U", or "P" representing "detected", "undetected", or "potentially detected", respectively, to denote the detection characteristics of the particular fault and point of observation combination. The list-of-faults formatted dictionary includes a row for each point of observation and a list of the detected faults for that point of observation. Potentially detected faults are listed in separately.

A dictionary compression technique attempts to reduce the full dictionary's size by using heuristics to eliminate the inclusion of certain points of observation. Each elimination of a point of observation is manifested as a deletion of a column in a matrix-formatted full fault dictionary or a deletion of a row in a list-of-faults formatted full fault dictionary.

Compression techniques may make tradeoffs in the compressed dictionary's diagnostic resolution to gain better compression, execution time, etc.. The diagnostic resolution of a fault dictionary is the measure of that dictionary's capability to distinguish its faults among each other, and the full fault dictionary has the maximum diagnostic resolution. The diagnostic resolution is based on the set of test vectors and fault model used by a dictionary; however, since the compressed dictionaries are generated from only information contained in the full dictionary (or are generated with access to the same test vectors and fault model as the full dictionary), the full dictionary's resolution sets an upper-bound on the corresponding compressed dictionary.

The test vectors and fault model used have a direct impact on the size of the full dictionary and the heuristic performance of the compression techniques. All test vectors used for the ISCAS89 circuits in this paper are vectors generated by the HITEC program [3] and simulated using the PROOFS sequential circuit fault simulation program [4]. The fault model used is the single stuck-at model with equivalence fault collapsing. Only faults that are definitely detected at least once are included in the initial fault list [9], [10].

Three dictionary compression techniques are implemented and examined. The examination focuses mainly on the resulting programs' asymptotic efficiencies in terms of run time requirements and memory usage; however attention is also given to the multiplicative constants and lower-order terms when applicable. The execution time and memory requirements of each compression technique depends on that technique's predefined heuristic and on the implementation's chosen internal data representation.

This thesis also provides a collection of empirical data on memory usage and run time requirements for some ISCAS89 circuits tested. Similarities among the three techniques, observed through their implementations, are also mentioned during each compression technique's implementation discussion.

IMPLEMENTATION CONSIDERATIONS FOR LIST SPLITTING, SEQUENTIAL, AND COMPACT FAULT DICTIONARY COMPRESSION TECHNIQUES

BY

CHING TAI

1. Introduction

A full fault dictionary enumerates all points of observation and their corresponding fault lists in either a matrix or a list-of-faults format. A point of observation is a test vector and primary output combination. Such dictionaries are generally impractical and large for a large sequential circuit. The large size of the full dictionary creates storage and usage problems. Fault dictionary compression algorithms attempt to solve the size problem of the full fault dictionary by using heuristics to eliminate from inclusion into the compressed dictionary certain points of observation information. Each point of observation has the possibility of detecting, not detecting, or potentially detecting a fault. A fault is consider "detected" by a point of observation if a circuit containing the fault gives a different logic value than the good (fault-free) circuit at that point of observation. A fault is "not detected" (or "undetected") if the faulty circuit gives the same value as the good circuit. If a faulty sequential circuit may potentially give a different value than a good circuit, then it is "potentially detected".

A full fault dictionary lists all the points of observation of a circuit. For the matrix format, the full dictionary includes a column for each point of observation and a row for each fault. The intersections of the rows and columns of the matrix are marked by a "D", "U", or "P" representing "detected", "undetected", or "potentially detected", respectively, to denote the detection characteristics of the particular fault and point of observation combination. The list-of-faults formatted dictionary includes a row for each point of observation and a list of the detected faults for that point of observation. Potentially detected faults are listed in separately.

A dictionary compression technique attempts to reduce the full dictionary's size by using heuristics to eliminate the inclusion of certain points of observation. Each elimination of a point of observation is manifested as a deletion of a column in a matrix-formatted full fault dictionary or a deletion of a row in a list-of-faults formatted full fault dictionary.

Compression techniques may make tradeoffs in the compressed dictionary's diagnostic resolution to gain better compression, execution time, etc.. The diagnostic resolution of a fault dictionary is the measure of that dictionary's capability to distinguish its faults among each other, and the full fault dictionary has the maximum diagnostic resolution. The diagnostic resolution is based on the set of test vectors and fault model used by a dictionary; however, since the compressed dictionaries are generated from only information contained in the full dictionary (or are generated with access to the same test vectors and fault model as the full dictionary), the full dictionary's resolution sets an upper-bound on the corresponding compressed dictionary.

The test vectors and fault model used have a direct impact on the size of the full dictionary and the heuristic performance of the compression techniques. All test vectors used for the ISCAS89 circuits in this paper are vectors generated by the HITEC program [3] and simulated using the PROOFS sequential circuit fault simulation program [4]. The fault model used is the single stuck-at model with equivalence fault collapsing. Only faults that are definitely detected at least once are included in the initial fault list [9], [10].

Three dictionary compression techniques are implemented and examined. The examination focuses mainly on the resulting programs' asymptotic efficiencies in terms of run time requirements and memory usage; however attention is also given to the multiplicative constants and lower-order terms when applicable. The execution time and

memory requirements of each compression technique depends on that technique's pre-defined heuristic and on the implementation's chosen internal data representation.

This thesis also provides a collection of empirical data on memory usage and run time requirements for some ISCAS89 circuits tested. Similarities among the three techniques, observed through their implementations, are also mentioned during each compression technique's implementation discussion.

2. Overview of Dictionary Compression Techniques Considered

Three dictionary compression techniques are implemented and discussed. They are the List Splitting, Sequential, and Compact techniques [10]. An overview of each technique's definition is provided in the three subsections below. The overview is given in the form of a sequence of instructions dictated by the heuristics; therefore, the overviews provide insights into a straightforward implementation of the compression techniques. Each compression technique's diagnostic resolution and simulation cost (number of fault simulations required) is also discussed in the corresponding overview subsections below.

The dictionary sizes and time required to generate each dictionary are given in Table 1 and 2. Table 1 and 2 also lists previous results for comparison [10]. The minor differences between the previous results and the results of this thesis are explained in the following sections 2.1, 2.2, and 2.3.

Table 1: Summary dictionary size and running times for List Splitting.

Circuit	Previous Results [10]	My Results	
	List Size	List Size	Time
s386	161	162	(0)
s641	233	234	(1)
s713	231	232	(1)
s820	342	342	(3)
s832	327	327	(4)
s1196	610	610	(4)
s1238	612	612	(5)
s1488	792	793	(17)
s1494	785	786	(16)
s35932	10892	10919	(2.54h+0.12h)

All sizes are given in matrix columns for a matrix representation of the dictionary.

All times are in seconds except where noted by "h" suffix, which is in hours. When the time a program spends in system calls is above 1 second, that time is listed after the "+" sign. Executions done on a 4 processor SPARC 20 with 55MB memory.

Table 2: Summary dictionary size and running times for Sequential and Compact.

Circuit	Previous Results [10]		My Results			
	Sequ Size	Comp Size	Sequ Size	Time	Comp Size	Time
s386	297	295	298	(0)	296	(1)
s641	212	212	212	(1)	212	(3)
s713	176	176	176	(1)	176	(5)
s820	472	450	471	(2)	448	(25)
s832	498	483	501	(3)	486	(25)
s1196	508	499	508	(3)	499	(14)
s1238	533	529	533	(3)	529	(20+1)
s1488	1.33K	1.29K	1.33K	(13+1)	1.29K	(214+6)
s1494	1.23K	1.19K	1.23K	(13+1)	1.19K	(186+4)
s35932[1]	N/A	N/A	N/A		646	(10.08h+1.55h)
s35932[2]	N/A	N/A	1871	(0.66h+0.11h)	656	(13.84h+1.25h)
s35932	6768	N/A	6788	(1.99h+0.11h)	N/A	
s35932[3]			6788	(2.37h+0.12h)	131	(118.45h+2.74h)

All sizes are given in matrix columns for a matrix representation of the dictionary.

All times are in seconds except where noted by "h" suffix, which is in hours. When the time a program spends in system calls is above 1 second, that time is listed after the "+" sign. executions done on a 4 processor SPARC 20 with 55MB memory.

[1] This s35932 data is derived from using only 5% of the faults of s35932. The 5% is from a random, uniform distribution.

[2] This s35932 data is derived from using only 10% of the faults of s35932.

[3] This simluation is done on a single processor Sparc 20 with 25MB RAM.

2.1 The List Splitting Compression Technique Overview

List Splitting works with indistinguishable fault classes. An indistinguishable fault class is defined as a group of faults that cannot be distinguished from each other based on the point of observation information of a dictionary.

The List Splitting technique initially groups all faults into one indistinguishable class, and updates the initial indistinguishable fault class with the information gathered from each point of observation considered. Hence, in the beginning, no points of observation have

yet been considered, and as a result, there is one indistinguishable class containing all faults (i.e., no faults can be distinguished).

List Splitting then considers each point of observation with no consideration ordering constraints and updates the indistinguishable classes according to the detections made on each point of observation. Points of observation are only included in the compressed dictionary only if they enhance the resolution of the dictionary.

The updating of an indistinguishable class involves creating two new indistinguishable classes in place of the original if the current point of observation has made both detection(s) and undetection(s) on the faults within the original class. One of the new classes will contain all the detected faults, while the other contains the undetected faults. The potentially detected faults are added to the larger of the two new classes. If any indistinguishable class is broken into two, then List Splitting considers that point of observation to have enhanced the resolution of the dictionary's current state from the previous state and includes the point of observation in its compressed dictionary. The point of observation consideration and indistinguishable fault class updating process continues until all points of observation have been checked against a state of the indistinguishable fault class.

The characteristic grouping of potentially detected faults with only the larger of the newly created indistinguishable classes possibly compromises List Splitting's diagnostic resolution for sequential circuits. List Splitting's diagnostic resolution for combinational circuits is not compromised because combinational circuits do not have potentially detectable faults. By only grouping the potentially detectable faults with the larger class, List Splitting denies the consideration of the potentially detected faults in one of their

other possibilities, which may allow List Splitting to pass up later points of observation that distinguishes those faults in the smaller class as not contributing towards enhancing the resolution. If potentially detected faults are included with the undetected class, then their possible detection is not considered, and vice versa.

List Splitting specifies no particular ordering of consideration for the points of observation. Changing the ordering of consideration can lead to List Splitting dictionaries of different sizes for a combinational or sequential circuit and to different diagnostic resolutions for a sequential circuit.

The main advantage and disadvantage of the List Splitting technique is few simulation runs when compared with the other two techniques, number of simulation runs is same as for a full dictionary, and possibly lower diagnostic resolution for sequential circuits. The possible lower resolution characteristic makes List Splitting technique more applicable to combinational circuits than sequential circuits.

2.2 The Sequential Compression Technique Overview

Sequential works with indistinguishable fault pairs. An indistinguishable fault pair is a pair of faults where its two members cannot be distinguished from each other based on the points of observation information in a dictionary. An indistinguishable fault pair is definitely distinguished by a point of observation if and only if one of the faults in the pair is detected while the other is undetected.

The Sequential technique generates its initial indistinguishable fault pairs by creating all possible pairs of combinations of all its faults. "n" faults would generate [n(n-1)]/2 pairs.

In the beginning, no fault is distinguishable from any other fault since no point of observation information has yet been considered. Later, as points of observation are considered, the number of indistinguishable fault pairs will decrease as they become distinguished. Once an indistinguishable fault pair has been distinguished, it is no longer considered by Sequential and should be deleted.

After the generation of the initial indistinguishable fault pairs, the Sequential technique goes through two steps in the creation of its compressed dictionary. In its first step, it creates a pass/fail dictionary. Then, on the next step, it enhances the resolution of the pass/fail dictionary to create the final compressed dictionary.

A pass/fail dictionary is created by considering, as well as adding to the dictionary, one test vector (which includes all primary outputs) at a time and updates its internal fault pairs by eliminating all pairs that are definitely distinguished by that vector. Although much smaller in size than the full fault dictionary, a pass/fail dictionary's diagnostic resolution is generally much lower than that of the full dictionary's resolution [10]. The sequential technique next enhances the pass/fail dictionary's resolution to equal that of the full dictionary.

After completion of the first step in the creation of the pass/fail dictionary, the second step of the Sequential technique enhances the previously created pass/fail dictionary's resolution by reconsidering each test vector and output pin combination (the previous definition of point of observation) and adds that point of observation into its compressed dictionary if and only if it definitely distinguishes additional fault pair(s). Step 2 stops when either all points of observation have been considered or all undistinguished fault pairs have been distinguished.

Potentially detected faults are handled conservatively since the indistinguishable fault pairs are only deleted when they have been definitely distinguished based on the point of observation information. As a result, Sequential technique's compressed dictionaries have the same diagnostic resolution as the corresponding full fault dictionary for both combinational and sequential circuits.

Sequential specifies no particular ordering of consideration for the points of observation. Changing the order of consideration of points of observation of a circuit can lead to Sequential dictionaries of different sizes.

The simulation cost of a Sequential dictionary is twice that of the full dictionary. Once to generate the pass/fail dictionary and once more to enhance the pass/fail dictionary.

2.3 The Compact Compression Technique Overview

The Compact technique uses a two step process to construct its compressed dictionary. Compact shares many similarities with the Sequential technique. As with the Sequential technique, Compact initially starts out with all faults paired as indistinguishable fault pairs, and it performs a pass/fail dictionary generation like the Sequential technique, eliminating any definitely distinguished fault pairs. Hence the initial fault pair generation procedure and first step of the Compact compression technique is the same as that for a Sequential.

On the second step, Compact considers each test vector and primary output combination (i.e., point of observation) independently on its current state of indistinguishable fault

pair. From those points of observation, Compact selects the point of observation that distinguished the largest number of its currently indistinguishable fault pairs and adds that point of observation to its compressed dictionary, updating its indistinguishable fault pairs with the information associated with the chosen point of observation. There are no rules for tie breaking.

The updating procedure is the same as that used by Sequential, i.e., remove from further consideration of the distinguished fault pairs. This point of observation selection and indistinguishable fault pair elimination process is continued until zero currently undistinguished fault pairs can distinguished by each of the points of observation, i.e., no more fault pairs could be distinguished.

Compact is greedy in that it only considers each point of observation's local effectiveness by basing its decision on which point of observation to include entirely on how many indistinguishable fault pairs are distinguished in the current context, i.e. Compact does not consider the consequences of influence later on in the indistinguishable fault pairs state. Compact dictionaries are usually smaller than or the same size as their Sequential counterparts [10].

Potentially detected faults are handled conservatively by deleting only fault pairs that are definitely distinguished by the point of observation, as in the Sequential compression technique. As a result, Compact dictionaries have the same diagnostic resolutions as the corresponding full dictionary for both combinational and sequential circuits.

The simulation cost of the Compact technique is significantly more than the full fault dictionary. The pass/fail dictionary generation during its first step requires the simulation

cost of a full fault dictionary. Additional simulations are required during the second step. Each iteration of the selection process used to determine the point of observation distinguishing the most fault pairs requires close to the full dictionary simulation cost. The worst case of simulation runs could be in the order of fault dictionary's run time squared; however, this does not happen frequently in practice since usually all distinguishable fault pairs get distinguished well before then. Compact's number of simulations is still significantly more when compared to Sequential.

3. Implementation of the Three Techniques

A total of five programs were written. They are List Splitting Implementation 1, Sequential Implementation 1, Compact Implementation 1, Sequential Implementation 2, and Compact Implementation 2. The difference between Implementation 1 and 2 lies in the internal data structures used to represent the program's state information.

The sections below describe each of the five program's internal data representation and operations the program performs on its data to correctly perform the necessary dictionary compression.

3.1 Implementation 1

In Implementation 1, each of the three compression techniques is implemented in a straightforward fashion in terms of the data structures used, and hence, the procedures of operations, as describe in the previous overview section. Due to the exact match between Implementation 1 programs' construction to each corresponding technique's overview and definition, detailed implementation description are omitted here.

List Splitting stores its indistinguishable fault classes as lists of faults, one list for each fault class, and splits the lists according to the procedure outlined in List Splitting's technique overview. Sequential and Compact store their indistinguishable fault pairs as a linked list of explicitly enumerated pairs of faults that corresponds directly to an indistinguishable fault pair. The list of fault pairs is maintained according to the procedures outlined in each technique's overview.

3.1.2 Enhancements for Compact

A strategy to improve Compact's efficiency was mentioned in a previous work [10]. The strategy is to select additional points of observations that distinguishes the same number of additional, indistinguishable fault pairs as the chosen point of observation for inclusion into the Compact dictionary. This strategy selects valid points of observation because each successive point of observation sets a new upper-bound on the number of distinguishable fault pairs (by definition of Compact). The enhancement strategy only works well towards the end of a Compact run because in the beginning, very few points of observation distinguishes the same maximum number of fault pairs.

A different strategy is used by the Compact algorithms in this thesis for increasing their efficiencies. While going through the process of selecting the locally optimal point of observation, the program needs to consider all points of observation that has not been included in the dictionary and have better-than-zero fault pair distinguishing counts. During this process, the program would save the top N number of points of observation (the points of observations that distinguished the most fault pairs).

Let the N points of observation be ordered in descending order according to the number of fault pairs distinguished. The first saved point of observation is the locally optimal point of observation and will be included in the Compact dictionary. The last of those N saved points of observation will be a higher bound on the next locally optimal point of observation excluding the saved points of observation occupying positions 2 through N-1.

The hope for a performance enhancement for Compact is that among the saved points of observation lies the next locally optimal point of observation. This is checked by

calculating the number of fault pairs distinguished by the saved points of observation on the indistinguishable fault pairs state after the application of the first saved point of observation (i.e., the current local optimal). If the next local optimal is among the saved point of observation it will have the following two characteristics: (1) have a fault pairs distinguished count $\geq$ the higher bound of the non-saved points of observation, and (2) have a fault pairs distinguished count $\geq$ all other saved points of observation.

The enhancement outlined above (and used in the Compact program of this thesis) is more general than the method mentioned previously [10], i.e., the previous method can arise during a normal Compact program's run as a special case of this strategy.

3.2 Implementation 2 (for Sequential and Compact Only)

In addition to the straightforward fashion of Implementation 1, a second implementation (Implementation 2) is made for the Sequential and Compact techniques to possibly solve the large memory requirement problem of Implementation 1 when running on circuits with a large number of faults. Implementation 2 stores an implication of the current indistinguishable fault pairs by storing lists of faults. The fault pairs are generated as a combination of two faults chosen from all faults within a list. An example of Implementation 2 follows. Assume n lists are stored by a program, and each list has K_i faults, where i is 1, 2, ...n and refers to each of the n lists. Then, using this implementation, the program, by storing the sum of all n number of k_i number of faults in its n lists, can represent (i.e., imply) the sum of all n number of k_i choose 2 number of fault pairs.

3.2.1 Application of Implementation 2 on the Sequential Technique

The application of Implementation 2's method of pair representation can be adapted for the Sequential technique. The following discussion will show how it is used in Implementation 2 of the Sequential compression technique, and some of the reasoning can be readily applied to the Implementation 2 of the Compact compression technique.

Implementation 2 begins with a single list containing all of its faults. This single list represents all possible combinations of the fault pairs to form the initial indistinguishable fault pairs of the Sequential technique. All possible fault pairs can be constructed by performing an "n choose 2" from the n faults, and by definition of Implementation 2, that is what a list of n faults implies.

After creation of the initial indistinguishable fault pairs state, the Sequential technique uses information from points of observation to distinguish all distinguishable pairs. In the first step (refer to technique overview section for details) of the Sequential's operation, it does not perform selections among the vectors when generating the pass/fail dictionary. Each vector's detections are applied to the current indistinguishable fault pairs. In the second step of its operation, the technique selects only points of observations that distinguish at least one pair in its current indistinguishable fault pairs state. Therefore, the Sequential technique can be constructed using repeated calls to two major functions: one that checks for any pair distinction and one that generates the next state of the indistinguishable fault pairs by removing all distinguished pairs from the current state. Once these two functions are implemented, the implementation of the Sequential technique becomes a simple process.

The pair distinction detection function. The criteria for distinction of an indistinguishable fault pair is that one of the faults in the pair be detected and the other be undetected. Hence to check for at least one pair distinction, Implementation 1 simply runs the distinction criteria against its stored list of indistinguishable fault pairs and stops when a pair is found to comply with the criteria. Implementation 2 checks for a distinction by considering each list in its stored lists of faults. In Implementation 2, if at least one list contains at least one detected and at least one undetected fault, then Implementation 2 can conclude that the point of observation has distinguished at least one indistinguishable fault pair; otherwise, the point of observation cannot distinguish any of the indistinguishable fault pairs in the current state.

The next state generation function. The Sequential technique removes the distinguished fault pairs from further consideration in the next state, i.e., change the set of indistinguishable fault pairs to the next state, reflecting the information gained from the point of observation. In Implementation 1, the change from a state of indistinguishable pairs to the next state involves the removal of the distinguished fault pairs from the linked list of pairs of the previous state. In Implementation 2, the elimination of distinguished pairs is achieved by replacing any list that has distinguished pairs by two necessarily smaller lists. The two new lists are created by placing all detected and potentially detected faults onto one list and placing all undetected and potentially detected faults onto the other list.

As an observation, Implementation 2 of Sequential compression technique bears some resemblance to Implementation 1's List Splitting technique. Both work with fault lists and split them according to detected and undetected characteristics of a fault against a point of observation. The difference is that Sequential puts potentially detected faults

onto both lists instead of List Splitting's heuristic of adding them only to the longer of the two lists. Therefore List Splitting can be thought of as a special case of the Sequential compression. The pass/fail dictionary generation of step one of the Sequential technique is mainly superficial in separating the similarities of the two techniques, since it can easily be applied to List Splitting.

3.2.2 Application of Implementation 2 on the Compact Technique

The operations of the Compact technique are very similar to the Sequential technique, and as a result, the implementation of Compact technique is based on the two major functions of the Sequential: the pair distinction function and the next state generation function. Additionally, since the Compact technique selects the locally optimum point of observation (i.e., the point of observation distinguishing the most fault pairs), it needs to derive this extra information that is not previously needed for the Sequential technique.

By using the Sequential technique's next state generation function, a set of lists may contain duplicate pairs which should only be counted once for the determination of the locally optimal point of observation. The multiple counting problem can be solved in many ways; the solution requires a way of finding the number of multiple counts. The solution used in Implementation 2 of Compact is described below.

One solution (used in Implementation 2 of Compact) to correct the multiple counting of duplicate pairs is to keep two lists of faults and two counters. One of the list contains encountered, detected faults; and the other list contains encountered, undetected faults. One of the counters contains encountered, detected counts; and the other contains encountered, undetected counts. Both lists are initialized to clear, i.e., initialized to the

state of no faults encountered, before applying the point of observation on the set of lists. During the next state generation, each list is considered separately, and the encountered, detected and undetected counters are set to zero on each list. For each list, all of that list's faults are run against the encountered, detected list if the fault is detected, and all of the list's faults are run against the encountered, undetected list if the fault is undetected. If any of those faults have not been previously encountered, the corresponding encountered list is set to reflect that the fault has now been encountered. Otherwise, if the corresponding encountered list has already encountered the fault before, then the corresponding counter is incremented by 1. The number of extra counted pairs is equal to the product of the two encounter counters.

The explanations for the finding of the correct number of pairs distinguished on a point of observation is done in three parts (part 1, part 2, and part 3). Part 1 provides a formula to finding the number of distinguished fault pairs assuming there are no duplicate pairs. Part 2 makes an observation that is required for part 3's case 2. Part 3 uses the method of the two list and counter maintenance described above to find the number of distinguished pairs when duplicates are present due to the handling of potentially detected faults in the next state generating function.

Part 1. A list has no duplicated faults, so each of its implied indistinguishable fault pairs is unique. For each detected fault on that list, the number of pairs that will have that fault and an undetected fault in the list is the number of undetected fault in the list. Hence, the total number of distinguished pairs is equal to detected fault count multiplied by the undetected fault count.

Part 2. The group of duplicate faults on all lists that contain them are the same. E.g., if a list has faults f1, f2, ... fn, in duplicate with another list, later if the faults f1, f2, ... fn of one list is split or further duplicated, the same group of fault on the other list will experience the same split or duplication.

Part 3. The number of distinguished pairs for a list is equal to: (number of detected faults * number of undetected faults) - (value of encountered, detected counter * value of encountered, undetected counter). The first term of the equation contains an extra count for each duplicate instance of a distinguished pair because each instance will count once for each list. Therefore, to get the correct number of distinguished pairs, the number of duplicate instances of the pairs are subtracted from the over-counted number. The second term contains the number of over counts in a list. If the encountered, detected counter is set, it means a fault that is counted as detected has been previously encountered in a previously processed list that is part of the current set of lists that determines the current indistinguishable fault pairs. The same applies to the encountered, undetected counter.

Part 3, case 1. Assume that at least one of the "encountered" counters is zero after following through the counter updating procedure described above. That means all of the detected or undetected or both categories of faults have never been encountered before; and any pair formed will be different in at least one fault member from any previously encountered (and counted) distinguished pair. Hence the product of the two counters are zero, which is correct.

Part 3, case 2. Assume the two "encountered" counters are non-zero after following through the counter updating procedure described above. This means that there are a certain number of detected faults that have been encountered previously and a certain

number of undetected fault that have been encountered previously. The pairs formed by those detected and undetected faults are duplicates. They are duplicate pairs because of part 2 above. Note that no other lists, besides the lists containing the group of duplicates, can have that duplicate fault by the definition of initial conditions of the Implementation 2 and the next state generation function. Therefore the product of these two counters provide the correct duplicate count.

The enhancement mentioned in the previous section for Compact Implementation 1 is also applicable to this implementation.

4. Memory Usage of the Implementations

The discussion of memory usage is broken up into three parts: the first part discusses the memory usage of Implementation 1, the second part discusses memory usage of Implementation 2 (applicable to Sequential and Compact techniques only), and the third part compares Implementations 1 and 2 (for Sequential and Compact).

4.1 Memory Requirements of Implementation 1

4.1.1 List Splitting Compression

Memory usage is not a major problem in List Splitting. The memory usage of an algorithm implementing this technique in a straightforward fashion starts out at a level equal to that of the number of faults for a circuit, i.e., linear complexity, to store the initial fault list, and the memory requirement decreases as each fault is totally distinguished from all the rest of the faults and is deleted.

The actual rate of decrease depends on each point of observation's detection results on a list and the ordering of points of observation consideration by the algorithm. The memory requirement does not increase for List Splitting during the program's execution because when a list is split, the maximum sum of the lengths of the two lists can only be as long as the original list's length. The sum can be smaller than the original list's length if any of the two new lists has only one fault, in which case that list can be deleted.

Experimental results on the memory usage for ISCAS89 circuits are plotted in Figures 1 and 2. A summary of the characteristics of each ISCAS89 circuit is listed in Table 3.

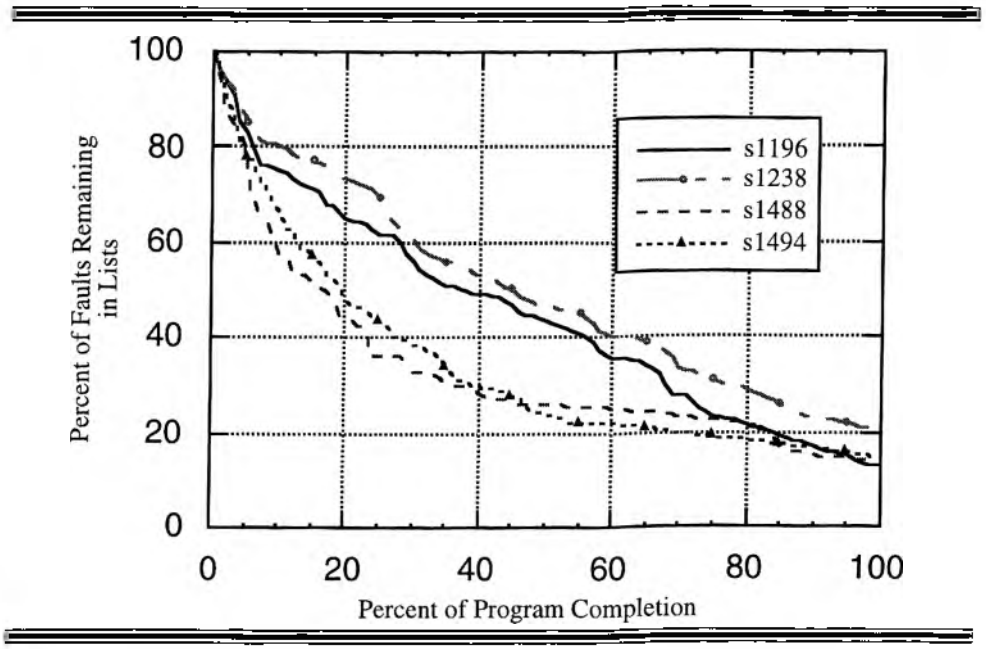

Figure 1: Memory usage (measured by number of total faults in lists) for Implementation 1 of List Splitting for ISCAS89 circuits s1196, s1238, s1488, and s1494.

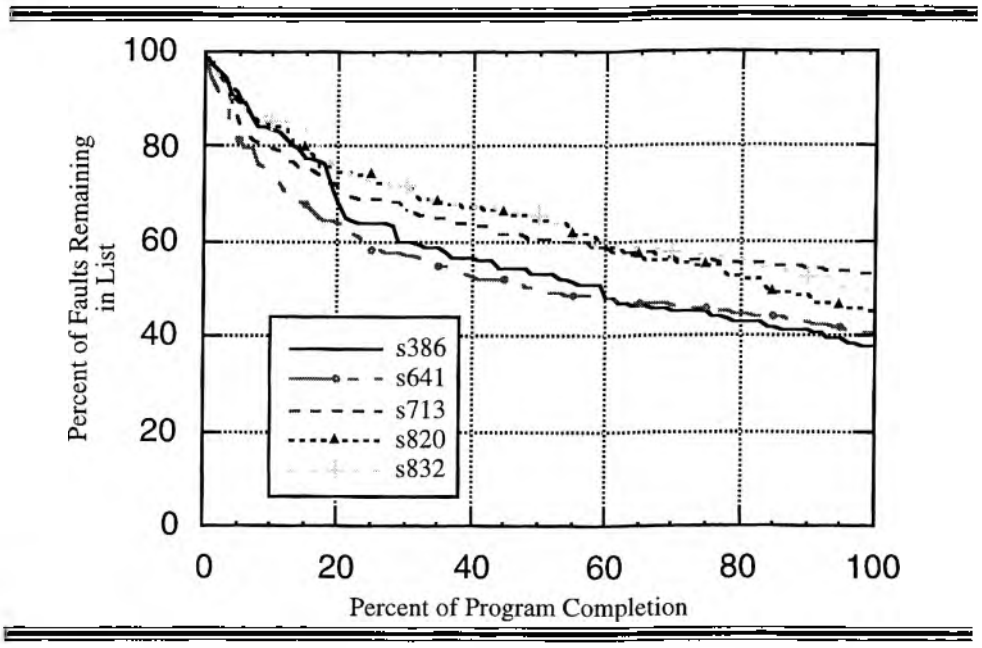

Figure 2: Memory usage (measured by number of total faults in lists) for Implementation 1 of List Splitting for ISCAS89 circuits s386, s641, s713, s820, and s832.

Table 3: Summary for the ISCAS89 circuits.

Circuit Name	Vectors	Outputs	Collapsed Faults
s386	285	7	384
s641	208	24	467
s713	170	23	581
s820	381	19	850
s832	426	19	870
s1196	453	14	1242
s1238	478	14	1355
s1488	1197	19	1486
s1494	1083	19	1506
s35932	630	320	39094

4.1.2 Sequential Compression Technique

Implementation 1 of the Sequential technique simply enumerates each of the indistinguishable fault pairs of the circuit and delete each pair when they are distinguished. E.g. to store the indistinguishable fault pairs: (f1, f4), (f1, f6), (f4, f6), (f2, f5), (f2, f8), and (f5, f8); the program will simply store them as six pairs, representing each pair by a "pairs data structure" containing the two fault numbers.

Using this form of representation, a Sequential compression program can either statistically allocate the required memory space or dynamically allocated it. The advantages of a static allocation is that it may be simpler, and hence, can decrease the memory required to represent each pair. Also, certain storage devices (e.g., disk drives) only allows static allocation for acceptable performance considerations. Dynamic allocation allows the possibility of reducing memory requirement by deleting and freeing the memory associated with each distinguished pair.

In static allocation, the memory usage requirement does not change and remains constant at [n(n-1)/2] fault pairs, where n is the number of collapsed faults for the tested circuit.

In dynamic allocation, the memory usage requirement starts out at the maximum, [n(n-1)/2], and the requirement decreases as fault pairs are distinguished and deleted during the course of program execution. The rates of decrease in memory requirement depends on the detection results of each point of observation on the current state of indistinguishable fault pairs. If the rate of decrease is fast enough and during the beginning of program's life, the asymptotic bound on the memory requirement may be less than the static allocation method's $O(n^2)$.

Experimental data on dynamic allocation's total memory usage are plotted for the tested ISCAS89 circuits in Figures 3-11 (see Imp 1 graph). The graphs plot both Implementation 1 and 2 for comparison. Implementation 1 is labeled "Imp 1" and Implementation 2 is labeled "Imp2." Implementation 1's data structure stores pairs of fault and Implementation 2's data structure stores lists of faults; hence the measure of memory usage, which is the vertical axis of the graphs, counts either the total number of pairs for Implementation 1 or total number of faults in all lists for Implementation 2.

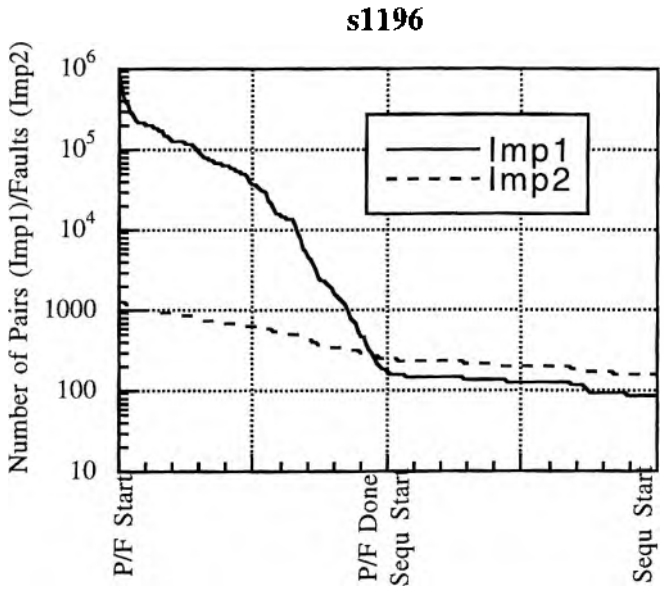

Figure 3: Memory usage (measured by total number of pairs or total number of faults) for Implementation 1 and 2 of Sequential.

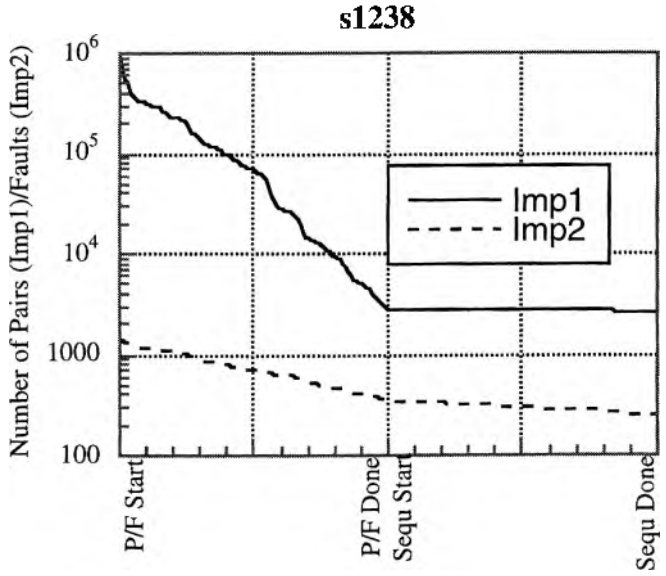

Figure 4: Memory usage (measured by total number of pairs or total number faults) for Implementation 1 and 2 of Sequential.

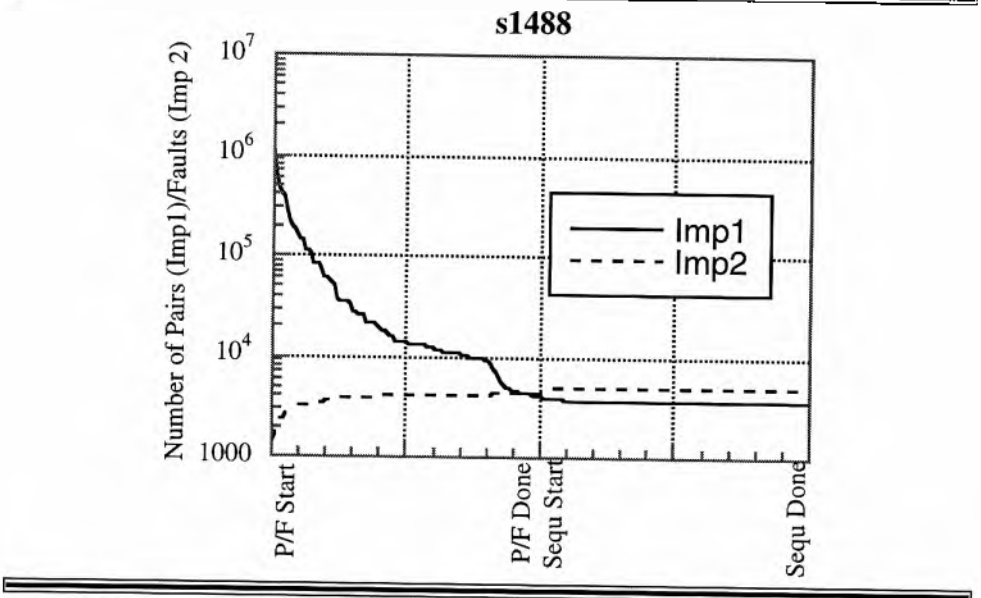

Figure 5: Memory usage (measured by total number of pairs or total number of faults) for Implementation 1 and 2 of Sequential.

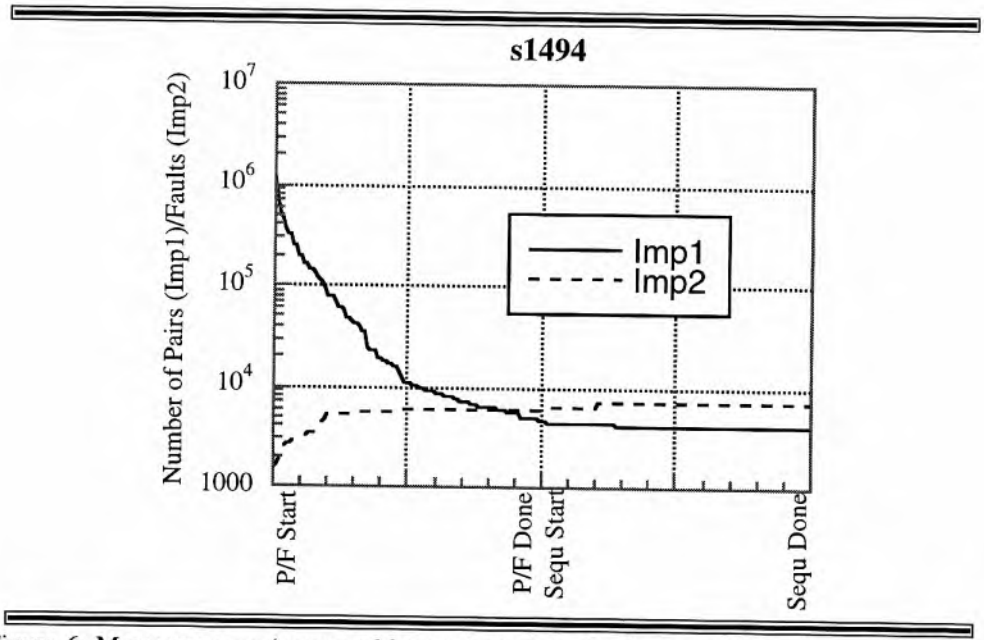

Figure 6: Memory usage (measured by total number of pairs or total number of faults) for Implementation 1 and 2 of Sequential.

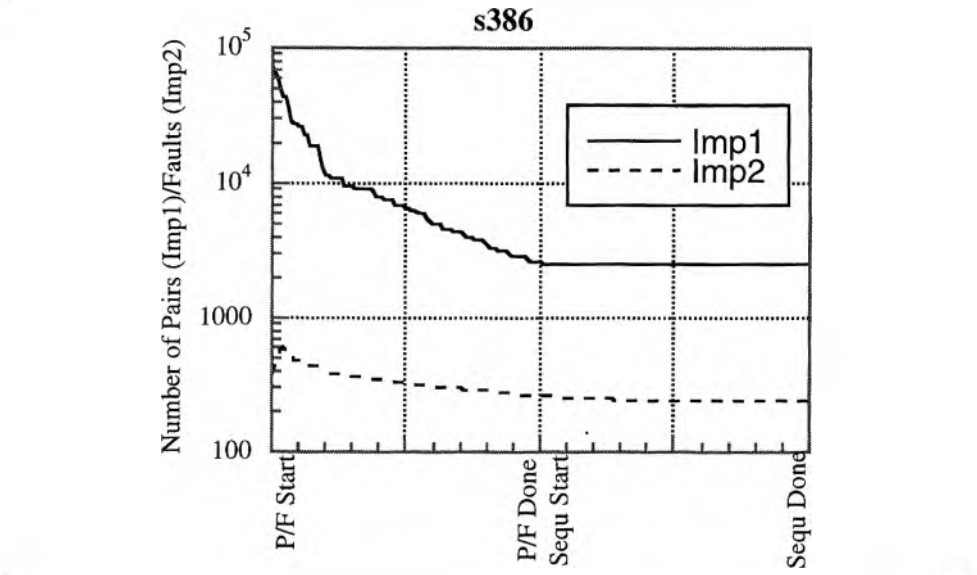

Figure 7: Memory usage (measured by total number of pairs or total number of faults) for Implementation 1 and 2 of Sequential.

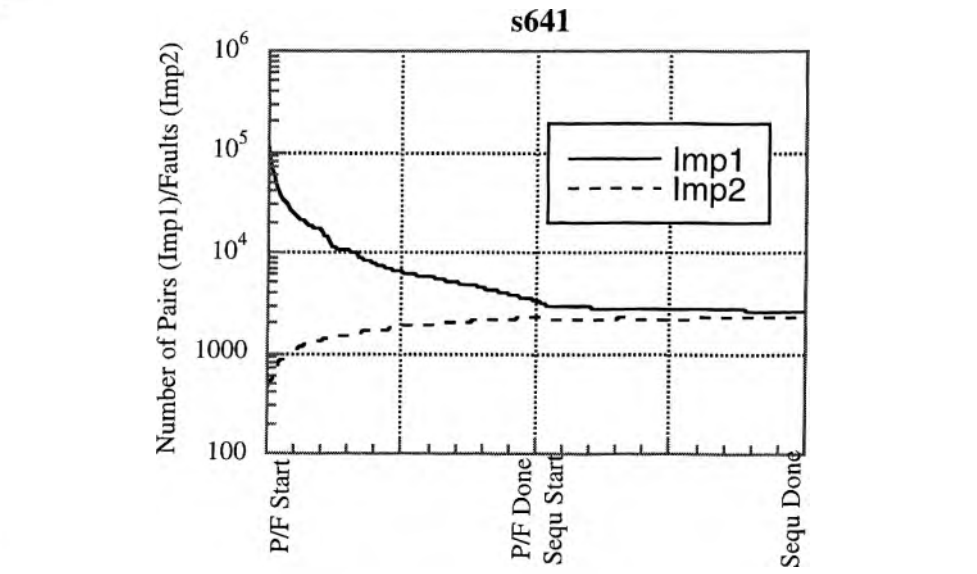

Figure 8: Memory usage (measured by total number of pairs or total number of faults) for Implementation 1 and 2 of Sequential.

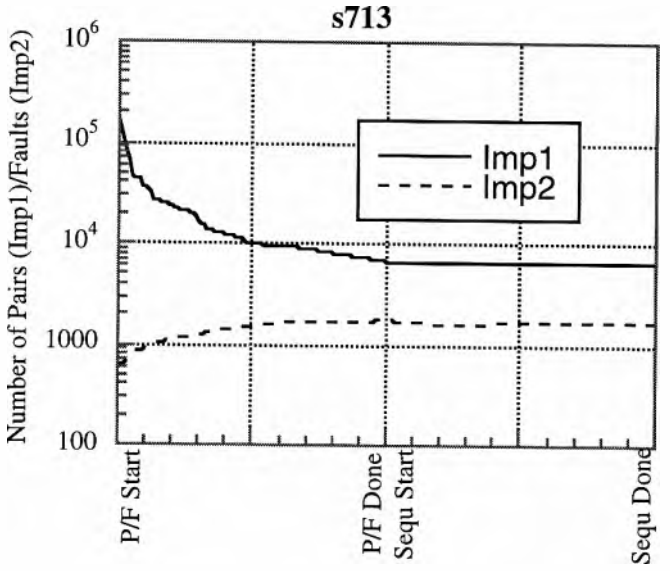

Figure 9: Memory usage (measured by total number of pairs or total number of faults) for Implementation 1 and 2 of Sequential.

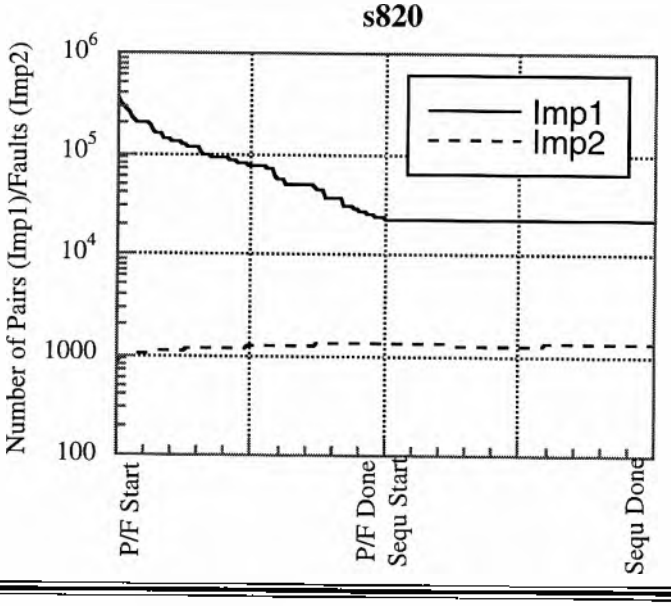

Figure 10: Memory usage (measured by total number of pairs or total number of faults) for Implementation 1 and 2 of Sequential.

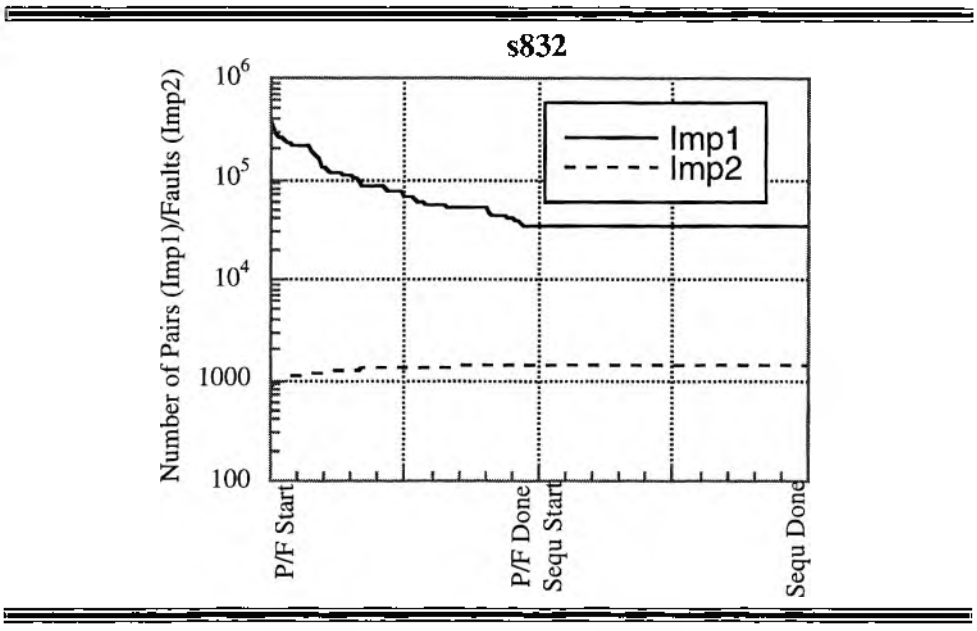

Figure 11: Memory usage (measured by total number of pairs or total number of faults) for Implementation 1 and 2 of Sequential.

4.1.3 Compact Compression

The Compact technique's memory usage is similar to the Sequential technique. For static allocation, there is no difference. The difference in dynamic allocation results from the possible increase in the rates of reduction of memory usage in the beginning of program execution due to Compact's selection of locally optimal points of observation, since each distinguished pairs are deleted.

However, the increased rate of reduction for Compact over Sequential does not mean that Compact has a smaller total memory usage per unit of execution time. The opposite is true because even before Compact experiences its first memory usage decrease, it had done the equivalence of an entire Sequential run, needing the maximum memory requirement for Sequential.

Experimental data on dynamic allocation's total memory usage for Compact are not graphed because they are very similar to the Sequential data except for the mentioned possible differences of rates of decrease of memory usage. However, as the Sequential memory usage figures show, the bulk of the memory usage decrease occurs during step 1 when the pass/fail dictionary is generated; hence, the different rates of decrease between Compact and Sequential is not consequential.

4.2 Memory Requirements of Implementation 2 (Sequential and Compact Only)

4.2.1 Sequential Compression

Implementation 2 of the Sequential technique implies the pairs in its indistinguishable fault pairs state by storing lists containing faults and the pairs are defined as all pair combinations of all fault within a single list. Hence, using this implementation it is possible to store n choose 2 pairs using only n memory space.

Unlike Implementation 1 where as pairs are distinguished, they are removed, Implementation 2 removes distinguished pairs by splitting a list. Potentially detected faults, by being on both of the new lists, increase the memory requirement as a list is split which contains potential detectable faults, i.e., the sum of the length of the two new lists is larger than the original list's length. When a fault is totally distinguished from all other faults, it can be deleted from the new list, and thereby decrease the memory requirement.

Hence the memory requirements of Implementation 2 changes as the lists are split based on the point of observation's detection results. The best-case memory usage occurs when no potentially detected faults occur during the pair distinguishing stage for each point of

observation, and the maximum memory usage remains constant at n. The worst-case memory usage occurs when each list contains all potentially detectable fault and the necessary one detected and one undetected faults on each point of observation, and the maximum memory requirement in this case is $O(2^n)$.

Worst-case memory usage of Implementation 2 is $O(2^n)$. Each new list is one element less than the original list, and there can be a maximum of n-2 indistinguishable fault pairs states generated from the initial state. Therefore at state n-1 (if the first initial state is assigned state value of 1), a possible of $2^{(n-2)}$ lists of faults can be present. See Figure 12 for a graphical representation.

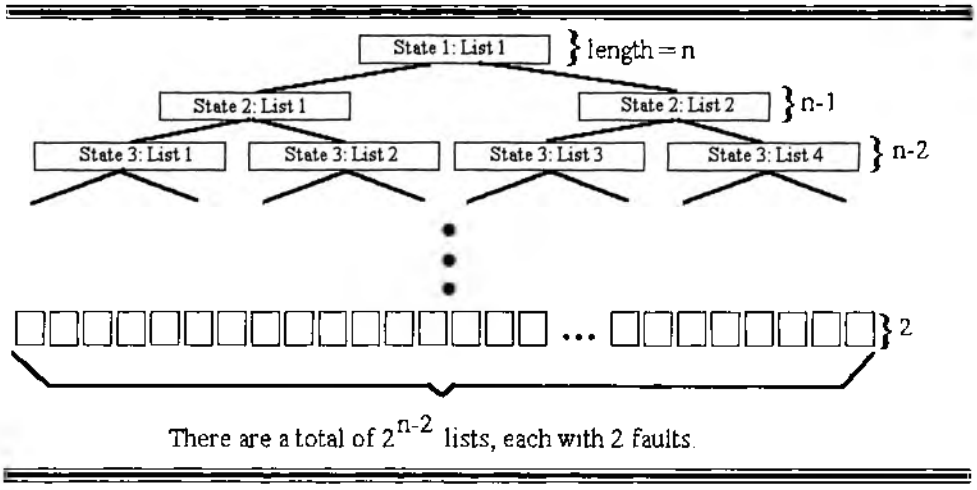

Figure 12: Number of lists and number of faults in a list for worst-case situation of Implementation 2.

The case where a certain constant percentage is assigned to each point of observation's detected results being potentially detected fault, provides a memory usage requirement of O(n). Let the height of the tree be "x", the constant percent be "k", and the initial length of the list be "n". Then the equation $(n)(k^x) = 2$ can be set up. Solving for "x", provides the solution (log 2 - log n)/(log k) for "x". Hence the height of the tree in Figure 12 becomes bounded by log n, and the final number of lists is also bounded by log n.

Experimental data on total memory usage throughout the execution of Implementation 2's Sequential compression technique are plotted for the ISCAS89 circuits in Figures 3 through 11 (see Imp 2-labeled lines).

4.2.2 Compact Compression

The Compact technique's memory usage is similar to the Sequential technique. Although some extra memory is required to check for possible multiple counting of duplicate pairs in the set of all lists, this memory is limited to, as it is implemented by the experimented Implementation 2 of Compact program, linear to the number of faults considered in a tested circuit. The number of faults is also used to derive the memory usage requirement for Sequential (which is now applied to Compact); and hence, the extra memory required for duplicate pair correction does not change the asymptotic memory requirement for Compact from that of Sequential. Experimental data on the total memory usage of Compact are very similar to those for Sequential for the tested ISCAS89 circuits, and they are not plotted.

4.3 Comparison of Implementation 1 and 2 (Sequential and Compact Only)

The possible savings in memory requirement when using Implementation 2 is achieved by allowing the representation of [n(n-1)/2] pairs with n storage requirement. However, actual memory requirement may be more than implementation 1, depending on the number of potentially detectable faults and on the number of distinguished pairs each point of observation finds.

For all the tested circuits, maximum memory usage for Implementation 2 has been lower than that for Implementation 1. The memory requirements for Implementation 2 for all tested circuits also do not approach Implementation 2's worst-case value. Implementation 2 also allows the possible execution of larger circuits that Implementation 1 will have difficulty with due to the unrealistic initial [n(n-1)/2], where n is the number of faults, memory requirement. E.g., the circuit s35932 with 39,094 faults would require ~764 million initial pairs to be stored. If s35932 were to be run on Implementation 1, all those pairs would need to be created and stored, making it difficult to run. However, s35932 has been run successfully on Implementation 2 using approximately 4-6 MB of memory during its highest memory usage period.

5. Running Times of the Implementations

The run time of each compression technique is based on that technique's definition and its implementation. The technique definition determines the number of iterations. Each iteration is defined as a simulation, i.e., generation a point of observation, and applying that point of observation information of the indistinguishable fault class/pair state. A technique's implementation determines each iteration's time requirement.

Each iteration's time bound for a technique is determined by its internal data representation or the memory usage, since the two are directly related. Because the previous memory usage analysis have been based on pair or fault counts, the run time requirement is the time that would be needed to go through each of the pairs once to generate the next state of indistinguishable fault pairs. In Implementation 2 of Compact, this would include the time used to check for duplicate pairs, but this time bound is a constant if the encountered, detected/undetected lists are maintained as arrays. Therefore the analysis for memory usage can be readily applied here to determine the run time of an iteration, i.e., each iteration's run time is linearly proportional to memory requirement during that iteration.

The number of iterations, simulations, needed is intrinsic to a particular technique's definition, and it is not possible to change that asymptotic bound. For example, Sequential and Compact techniques repeatedly simulate the same test vectors; and hence, their simulation times could possibly be decreased by saving the results of the first simulation; this behavior would create a full fault dictionary temporarily and may not be possible or feasible in certain situations. Also, even if the full dictionary were created and a look up is performed rather than a simulation, the time required is still of the same order

since there is now a look-up time instead of the simulation time. Hence, no change in the time bound.

The run time for each technique on each of the tested ISCAS89 circuits can be estimated from the corresponding memory usage graphs and the overview of that technique. Listing the actual run times of each implementation of the techniques is not useful for comparison because the slight variations of optimizations on each program would throw the numbers off by multiple factors due to the large number of repeated iterations that parts of a program receives.

6. Conclusion

The similarities among the three compression techniques have been discussed during each technique's implementation. Compact technique's two implementations have been directly based on their Sequential counterparts, and Implementation 2 of the Sequential technique bears high resemblance to List Splitting.

The memory usages of the three compression techniques depend mainly on internal data representation. Implementation 1 constructs each technique in a straightforward fashion. Memory requirements for Implementation 1 of List Splitting is linear to the number of initial faults which creates no problems. However, Implementation 1's memory requirements for Sequential and Compact techniques both require the square of the number of initial faults. It has been shown that the Implementation 1's straightforward data representation of explicitly enumerating the indistinguishable fault pairs for the Sequential and Compact techniques creates problems due to the large memory requirement for circuits of the size of s35932. Another representation of indistinguishable fault pairs is used for Implementation 2 of Sequential and Compact that have a worst-case memory requirement much larger than the explicit pair representation, but actual usage on the ISCAS89 circuits is much lower. Using Implementation 2, building dictionaries for large circuits may be possible and efficient from the memory requirement point of view. Experimental results of memory usages for tested ISCAS89 circuits are plotted and referenced during the memory requirements discussion to provide some more information on typical memory requirements.

Run time dependencies of the implementations of the three compression techniques are mentioned. Run time depends on the definition of the technique and data representation

(i.e., memory requirements), and its asymptotic bound can be calculated from the analysis on memory usage and compression technique definition.

7. Acknowledgments

I am grateful to Vamsi Boppana, W. Kent Fuchs, Ismed Hartanto, Paul Ryan, and Srikanth Venkataraman for their guidance and help that made possible this thesis.

References

[1] V. Boppana, W. K. Fuchs, "Fault Dictionary Compaction by Output Sequence Removal," Proc. IEEE Int. Conf. on Computer-Aided Design, pp. 576-579, Nov. 1994.

[2] T. Niermann, "Techniques for Sequential Circuit Automatic Test Generation," CHRC Technical Report 91-8, University of Illinois, pp. 28-38, March 1991.

[3] T. Niermann, J. Patel, "HITEC: A Test Generation Package for Sequential Circuits," Proc. European Design Automation Conf., pp. 214-218, Feb. 1991.

[4] T. Niermann, W. Cheng, J. Patel, "PROOFS: A Fast, Memory-Efficient Sequential Circuit Fault Simulator," in IEEE Trans. on Computer Aided Design, Vol. II, No. 2, Feb. 1992.

[5] I. Pomeranz and S. Reddy, "On the Generation of Small Dictionaries for Fault Location," in Proc. IEEE Int. Conf. on Computer-Aided Design, pp. 272-279, Nov. 1992.

[6] E. Rudnick, W. K. Fuchs, and J. Patel, "Diagnostic Fault Simulation of Sequential Circuits," in Proc. IEEE Int. Test Conf., pp. 178-186, Oct. 1992.

[7] P. Ryan, S. Rawat, and W. K. Fuchs, "Two-Stage Fault Location," in Proc. IEEE Int. Test Conf., pp. 963-968, Oct. 1991.

[8] P. Ryan, W. K. Fuchs, "Addressing the Size Problem in Fault Dictionaries," Proc.19th Int. Symp. on Testing and Failure Analysis, pp. 129-133, Nov. 1993.

[9] P. Ryan, W. K. Fuchs, I. Pomeranz, "Fault Dictionary Compression and Equivalence Class Computation for Sequential Circuits," IEEE/ACM Int. Conf. on Computer Aided Design, pp. 508-511, Nov. 1993.

[10] P. Ryan, W. K. Fuchs, I. Pomeranz, "Fault Dictionary Compression," CHRC Technical Report, University of Illinois, Sept. 1994.

[11] P. Ryan and W. K. Fuchs, "Dynamic Fault Dictionaries and Two-Stage Fault Isolation," CHRC Technical Report, University of Illinois, Sept. 1994.