MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE

KHARKOV NATIONAL UNIVERSITY OF RADIOELECTRONICS

ISBN 966-659-088-3

Proceedings of East-West Design & Test Workshop (EWDTW'04)

Yalta, Alushta, Crimea, Ukraine, September 23 – 26, 2004

CONTENTS

SET OPERATION SPEED-UP OF FAULT SIMULATION

S.A. ZAYCHENKO,
A.N. PARFENTIY,
E.A. KAMENUKA,
H. KTIAMAN

Design Automation Department, Kharkiv National University of Radio Electronics, Lenin ave, 14, Kharkiv, 61166 Ukraine. E-mail: hahanov@kture.kharkov.ua

Abstract. In this paper there are presented data structures and algorithms for performing set theory operations upon lists of defects within deductive fault simulation method of digital systems. There are suggested 4 types of data structures and calculation procedures, which provide maximum performance for basic operations required for effective software implementation of the method.

Keywords: set operations, data structures, performance analysis, fault, fault simulation.

1. Introduction

Hardware designers and manufacturers demand significant performance acceleration for fault simulation and automatic test patterns generation tools (ATPG) [1] for large-scale digital systems, being targeted into the application specific integrated circuits (ASIC's). Over 50% of existing ATPG systems [1-4] use deductive method of fault simulation to obtain table of faults, covered by the applied test.

The performance distribution analysis of computation cycle during test-vector processing within deductive method (fig. 1) shows, that about 70% of time is spent on performing set theory's operations upon lists of faults: union, intersection and complement (difference). That's why the software implementation performance of the deductive method strongly depends on implementation efficiency of the set theory operations.

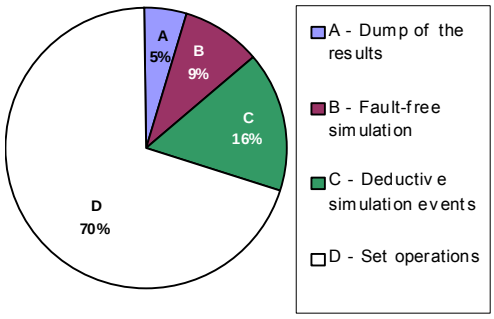

Figure 1. Performance distribution of computation cycle of the deductive fault simulation method

Software implementation of the set operations may use classic storage data structures and algorithms, which efficiency differs for various numbers of elements under processing. Relatively to deductive fault simulation method, the particular computations at the same time are performed upon sets with various range of elements number. That's why, there is no well-known data structure in general programming, which provides acceptable performance of implementation of the set operations for deductive fault simulation method.

The *research goal* is to analyze and select optimal data structures and processing algorithms of set theory operations, that will provide the highest performance and lowest memory usage for software implementation of the deductive fault simulation method.

The *research tasks* include:

– analysis of classic data structures, being used in discrete mathematics [5,6] and general programming [7-9] for implementation of set theory operations;

– development of the computation strategy, which provides high speed and low memory usage for fault simulation of large-scale digital systems;

– efficiency assessment of the developed strategy.

2. Classic data structures for sets. Characteristic vectors

Set theory operations are used for solving many mathematical tasks, including those, which are related to the described fault simulation topics. There are several data structures with different internal organization, which are frequently used in discrete mathematics and programming for set elements storage:

– characteristic vectors;

– linked lists;

– binary trees;

– hash-tables.

Definition 1. Characteristic vector [1,5,6] is a data structure for storing subsets of universe U with finite number of elements n, represented by the n-width binary word, where value 1 of bit I means that element i of U belongs to the set, while 0 means opposite:

$$W = (W_0,..., W_i,..., W_{n-1}),$$
$$W_i = \begin{cases} 1 \leftarrow W_i \in U; \\ 0 \leftarrow W_i \notin U, \end{cases} \quad (1)$$

where W – characteristic vector for stored set, W_i – bit i of the characteristic vector.

For example, the representation of the set with 4 elements, such as {2, 4, 7, 9}, in the universe of 10 elements will contain 4 bits with 1-value and 6 bits with 0-value in the indices, corresponding to element indices minus 1 (fig. 2):

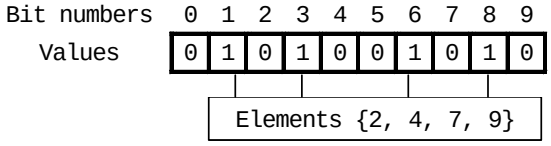

Fig. 2. Set representation by the characteristic vector

Main advantage of the characteristic vectors is the simplicity of implementation of the set operations, which are reduced to standard logic computations:

$$W_{A\cup B} = W_A \vee W_B,\ W_{A\cap B} = W_A \cdot W_B,$$
$$W_{A-B} = W_A \cdot \overline{W_B}, \quad (2)$$

where $W_{A\cup B}$, $W_{A\cap B}$, W_{A-B} – characteristic vectors of union, intersection and complement results, W_A, W_B – characteristic vectors for operand sets.

From (2) it is obvious, that such implementation of set operations has linear complexity relatively to the length of characteristic vector:

$$\theta_{A\cup B}(n) = t_\vee \cdot n, \theta_{A\cap B}(n) = t_\& \cdot n,$$
$$\theta_{A-B}(n) = (t_\& + t_!) \cdot n, \quad (3)$$

where $\theta_{A\cup B}$, $\theta_{A\cap B}$, θ_{A-B} – complexity of union, intersection and complement operations, $t_\vee$, $t_\&$, $t_!$ – time for performing logic operations OR, AND, NOT upon 1-bit arguments.

Another advantage of the characteristic vectors is the fixed amount of storage memory: allocation is performed only once during simulation initialization, and cleanup – once at the end.

Main disadvantage of using characteristic vectors for fault simulation is the square complexity of the storage memory depending on the number of circuit lines:

$$M = n_l \cdot n_w = n \cdot n = n^2, \quad (4)$$

where M – total amount of set storage memory, n_l – number of lines in the circuit, n_w – length of the characteristic vector, $n = n_l = n_w$.

For instance, to simulate relatively small circuit with 10 thousands gates, accordingly to (3), it is required to allocate 100 millions of bits. Such big memory usage makes characteristic vectors almost unacceptable for designs, which have hundreds of thousands and millions of gates. This problem can be partially solved by hierarchical device model decomposition, but this significantly complicates simulation between hierarchy levels.

Another significant disadvantage of the characteristic vectors is that large part of the memory is not being used within computations, as number of elements activated during simulation rarely exceeds 20%. The example shown on the fig. 2 contains 6 bits, holding non-informative zero bits. It shows that large part of the computations during set operations with characteristic vectors are performed on 0 values and are not useful.

3. Linked lists

Definition 2. Linked list [7-9] is a structure of interconnected information blocks, stroring data and addresses of the neighbor blocks:

$$L = (L_1,..., L_i,..., L_n),\ L_i = (D_i, A_{i-1}, A_{i+1}), \quad (5)$$

where L – stored set, L_i – i^{th} set block, D_i – date of the i^{th} set element, A_i – address of the i^{th} block.

The amount of memory required for storing such sets is proportional to the number of elements. The set, described on fig. 2, by applying definition 2, can be represented by the following data structure:

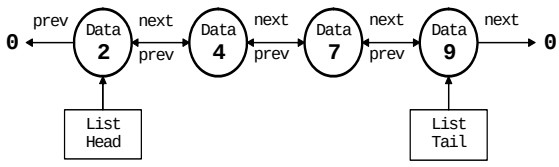

Fig. 3. Representation of the set by the linked list

It is efficient to perform set operations with ordered linked lists. Such structure does not

spend time on searching elements during comparison of two sets.

To perform union operation upon ordered linked lists (see procedure 6) it is required to iteratively compare elements from lists A and B with consequent increment of those index, whose current element is smaller, which should be written in the resulting set C. If elements of both lists are equal, one of them should be recorded to C with increment of both indices. After performing procedure (6) list C will be also ordered. The intersection (7) and complement (8) set operations can be implemented in the similar way.

$$\begin{cases} A_i < B_t \rightarrow C_j = A_i; j = j + 1; i = i + 1; \\ A_i > B_t \rightarrow C_j = B_t; j = j + 1; t = t + 1; \\ A_i = B_t \rightarrow C_j = B_t; j = j + 1; i = i + 1; t = t + 1, \end{cases} \quad (6)$$

$$\begin{cases} A_i > B_t \rightarrow t = t + 1; \\ A_i < B_t \rightarrow i = i + 1; \\ A_i = B_t \rightarrow C_j = B_t; j = j + 1; i = i + 1; t = t + 1; \end{cases} \quad (7)$$

$$\begin{cases} A_i > B_t \rightarrow t = t + 1; \\ A_i = B_t \rightarrow i = i + 1; t = t + 1; \\ A_i < B_t \rightarrow C_j = B_t; j = j + 1; i = i + 1. \end{cases} \quad (8)$$

where A, B – operands; Ñ – result; i, j, t – current element indices for A, B and C sets.

Procedures (6-8) have linear complexity relatively to the sum of number of elements for both sets:

$$\begin{aligned} \theta_{A \cup B} &= 2 \cdot t_C \cdot \min(n_1, n_2) + t_A \cdot (n_1 + n_2) + 2 \cdot t_I \cdot (n_1 + n_2); \\ \theta_{A \cap B} &= (2 \cdot t_C + t_A) \cdot \min(n_1, n_2) + t_I \cdot (n_1 + n_2 + \min(n_1, n_2)); \\ \theta_{A - B} &= 2 \cdot t_C \cdot \min(n_1, n_2) + t_A \cdot n_1 + t_I \cdot (2 \cdot n_1 + n_2), \end{aligned} \quad (9)$$

where n_1, n_2 – number of operands' elements, t_C – comparison time for 2 elements, t_A – assignment time to the element of output list, t_I – increment time.

The worst task for linked lists is an insertion of a single element. In this case it is required to locate the position of the new element taking existing order into a count, which requires sequential list iteration. Insertion of new element to the found position requires to assign references between element and its left and right neighbor.

4. Binary trees

Definition 3. Binary tree [5-6] – is an acyclic graph that represents relations between elements (subsets), where each node has one input and two output edges. The weight of each node is always bigger than the weight of left descendant and smaller than the right one:

$$V = (V_1, ..., V_i, ..., V_n), \; V_i = \left\{ A_i^L < A_i < A_i^R \right\} \quad (10)$$

where V – binary tree; V_i – i^{th} node of the tree; A_i – weight of the i^{th} node, A_i^L, A_i^R – weights of the left and right descendants.

Such set representation requires the amount of memory proportional to the number of elements, which is equivalent to linked lists. The set, shown on fig. 3, by taking into a count definition 3, can be represented by the following data structures:

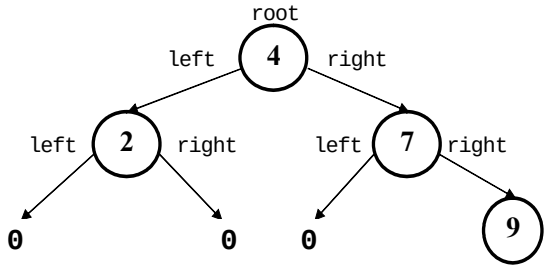

Fig. 4. Representation of the set by the binary tree

The search of the element in the binary tree is implemented by the iterative comparison of the required value and the current node according the following rule:

$$\begin{cases} x < A_i \rightarrow A_i^L; \\ x > A_i \rightarrow A_i^R. \end{cases} \quad (11)$$

Binary tree is called balanced, if it is organized in such way, that the number of left and right sub-trees is approximately equal. Balanced binary tree provides logarithmic computation complexity for the element search operation. In unbalanced tree state search becomes ineffective, because finding the element position requires more comparison operations. To convert unbalanced tree into balanced one it is necessary to select one of the nodes as a new root one in such way that the total amount of left and right sub trees would become approximately equal.

Usage of the binary trees is effective for execution of sets union, intersection and complement

operations, which can be implemented in the following way:

$$A \cap B: A_i \in B \to C_j = A_i; j = j + 1; i = i + 1;$$
$$A \cup B: \begin{cases} C_j = A_i; i = i + 1; j = j + 1; \\ B_t \notin A \to C_j = B_t; j = j + 1; t = t + 1; \end{cases} \quad (12)$$
$$A - B: A_i \notin B \to C_j = A_i; j = j + 1; i = i + 1,$$

The complexity of procedures (12) is relative to the size of the operands and defined as follows:

$$\theta_{A\cap B}(n_1, n_2) = \theta_{A-B}(n_1, n_2) = n_1 \cdot \theta_S(n_2) = n_1 \cdot \log_2 n_2$$
$$\theta_{A\cup B}(n_1, n_2) = n_1 + n_2 \cdot \theta_S(n_1) = n_1 + n_2 \cdot \log_2 n_1 \quad (13)$$

where θ_S – complexity of the element search in the tree.

5. Hash-tables

Definition 4. Hash-table [7-9] is a data structure for storage of set of elements, which position is defined by the hash-function.

Definition 5. Hash-function – special selected function returning low-informative decimal numbers (hash-codes) for the element values with small probability of duplication for different applied arguments.

Hash-codes are interpreted as cell indices of the table, which allows determining location of the element by fast hash-code computation. In general case, the dependency between element values, generated hash-codes and cell indices is very hard to predict. If table contains m cells, to define index of the element it is required to calculate modulo of the hash-code division on the m:

$$P_i = H(D_i) - \left\lfloor \frac{H(D_i)}{m} \right\rfloor, \quad (14)$$

where H – used hash-function; D_i – value of the i^{th} set element; P_i – position of the i^{th} in the hash-table;] [– notation of the integer division.

To store the set shown on fig. 2 in the hash-table it is required to create model, similar to the one shown on fig. 2. Let's assume, table contains 8 cells and hash-function for the elements generates the hash-codes, presented in (15). By computing modulo of the integer division of hash-codes on 8, the cell indices can be obtained:

$$H(2) = 1160294;\ P(2) = 1160294 \bmod 8 = 6;$$
$$H(4) = 1191544;\ P(4) = 1191544 \bmod 8 = 0;$$
$$H(7) = 1238419;\ P(7) = 1238419 \bmod 8 = 3; \quad (15)$$
$$H(9) = 1269669;\ P(9) = 1269669 \bmod 8 = 5.$$

Cell indices	0	1	2	3	4	5	6	7
Cell values	**4**			**7**		**9**	**2**	

Fig. 5. Set representation by hash-tables

Element search and insertion operations in the hash-table have amortized constant complexity [7-9]. This means, that in special situations, like collision and rehashing, operations can be more expensive.

Collision – situation during the work of hash-table, when calculated cell indices for two different elements are equal. In this situation element is placed in the closest free cell with bigger index. Frequent collisions make hash-tables inefficient.

To reduce collision probability the number of hash-table cells should be increased, which involves low-performance rehashing operations. This includes recalculation of all cell indices taking new size of the table into a count, and also elements replacing.

Implementation of the intersection, union and complement upon hash-tables is done according to (12), but unlike binary trees has amortized linear complexity:

$$\theta_{A\cap B}(n_1, n_2) = \theta_{A-B}(n_1, n_2) = n_1 \cdot \theta_S(n_2) = \\ = n_1 \cdot \theta^+(\text{const}), \\ \theta_{A\cup B}(n_1, n_2) = n_1 + n_2 \cdot \theta_S(n_1) = \\ = n_1 + n_2 \cdot \theta^+(\text{const}), \quad (16)$$

where $\theta^+(\text{const})$ – widely used notation of the amortized constant complexity.

Hash-tables are much more effective than binary trees for sets with large number of elements, but for very small sets (less than 10 elements) hash-tables are slower than all other described data structures.

The amount of storage memory for the hash-tables depends on the number of cells, which should be at least two times larger than the number of stored elements. This is required for effective hashing organization without frequent collisions. It is bigger, than for binary trees, but significantly smaller than for characteristic vectors.

6. Performance comparison of classic data structures

Performance of the described classic data structures can be experimentally compared by measuring computation time of big number (104-105 times) of intersection and union operations on sets of various sizes, filled by pseudo-random elements.

Experiment 1: let's measure the performance for operations upon sets containing less than 1000 elements (fig. 6 and fig. 7). Results allow to make the following conclusions:

1) Characteristic vectors should be used for smallest sets only (less than 50 elements); for larger sets this data structure uses unacceptable amount of memory.

2) For sets from 50 to 1000 elements usage of linked lists, binary trees and hash-tables is almost equivalent.

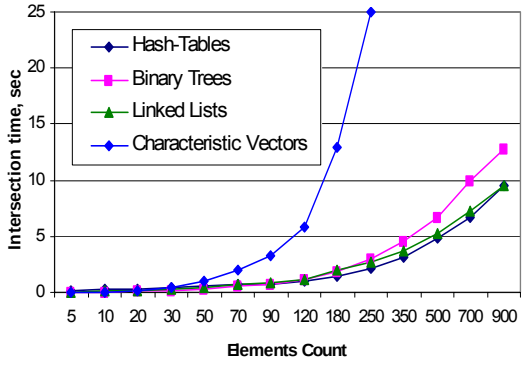

Fig. 6. Performance of intersection for small sets

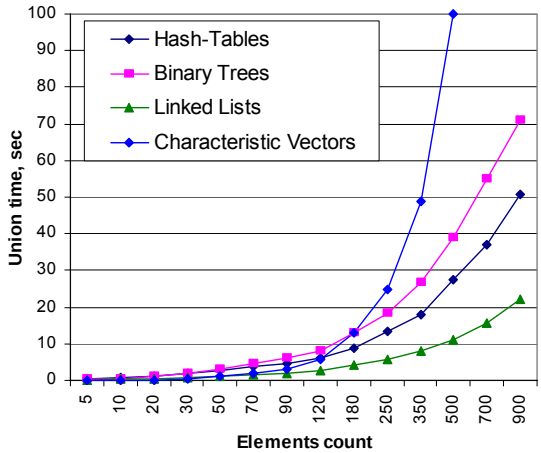

Fig. 7. Performance of union for small sets

Experiment 2: let's measure the performance for operations upon sets, containing from 1 to 50 thousand elements (fig. 8 and fig. 9).

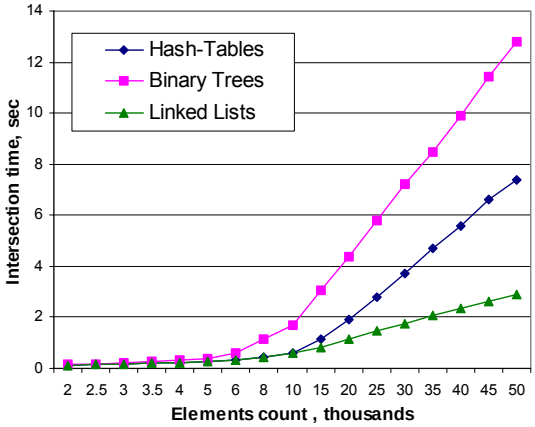

Fig. 8. Performance of intersection for large sets

Results allow to make the following conclusions:

1) Linked list is the most effective data structure for operations with large sets, esp. for union.

2) Performance of the set operations when using the hash tables is much better than using binary trees upon sets with 2-25 thousands of elements, but because of rehashing effects during table growth their efficiency for larger sets becomes closer to binary trees.

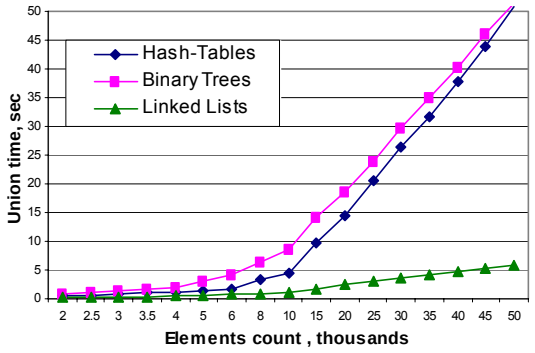

Fig. 9. Performance of union for large sets

Experiment 3: let's measure the performance of operation of single element insertion (fig. 10).

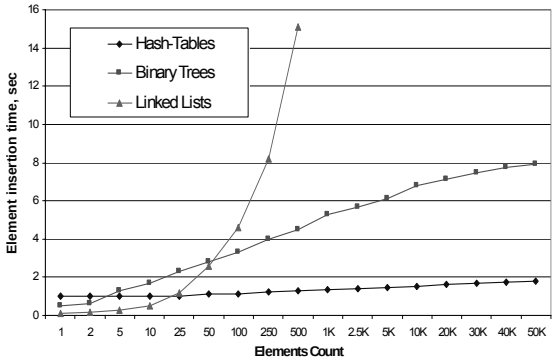

Fig. 10. Performance of single element insertion

Results show, that low-performance of the element insertion reduces advantages of the linked lists, found in experiment 2, almost to zero.

So, the researched internal structure and performance of the classic data structures for sets allow to make the following conclusions:

1) Definite choice of the most effective universal data structure for representing fault lists is not possible: performance properties of data structures are different for various operation types and set sizes.

2) Data structure should be effective both for set operations and single element insertion.

3) Any data structure for sets is not effective when number of stored elements strongly changes.

4) Exact estimation of the memory amount for elements, which are expected to appear in the future, is the way to improve performance of the basic operations execution.

7. Development of effective processing strategy for fault sets

Usage of only one of described data structures for representing defect lists in deductive fault simulation method will not provide performance acceptable for processing of large-scale digital systems. To create fast simulation program the FLP computation strategy is suggested, which is based on the *fault lists pool*.

The key of this strategy consists in centralized management of the data objects, implementing representation of the fault lists. Pool automatically manages creation, storage, allocation and cleanup of the list-objects, and also selection of the most effective list object for particular set operation. FLP works in the following way:

1) Pool manages created, used and freed list objects, dividing them according to the set size ranges.

2) Deductive simulator requests the most effective list object in the pool before set operation.

3) Depending on the operand sizes and the selected operation, pool estimates the maximum possible number of elements in the resulting set:

$$n_{A\cap B} \le \min(n_A, n_B);\ n_{A\cup B} \le n_A + n_B;$$

$$n_{A-B} \le n_A; \qquad (17)$$

where n_A, n_B – sizes of operand sets, $n_{A\cap B}$, $n_{A\cup B}, n_{A-B}$ – sizes of intersection, union and complement results.

4) Taking the previous estimation into a count, pool searches the suitable list object using the algorithm, which is shown on fig. 11:

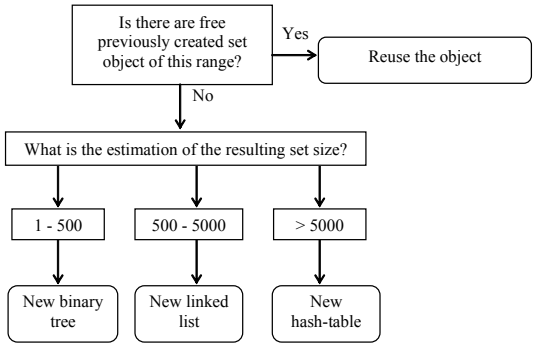

Fig. 11. Optimal list-object selection algorithm

5) Selected list object is passed to the deductive simulator for performing set operation.

6) If deductive simulator no longer needs the list object, it returns it to the pool, so it can be reused in further operations later.

Implementation of the set theory operations using the FLP resolves described problems of the classic data structures:

1. Accordingly to size of the set, pool creates the most effective object for the list. This allows using data structures at the most suitable range of sets size, providing maximum performance relatively to other data structures.

2. Preliminary estimation of the operation result size reduces time being spent on internal data structure reorganization, which increases the speed of the operations computation and tends to rational memory usage.

The disadvantage of the suggested computation strategy is that unordered and ordered data structures can not be used at the same time for computation. For instance, hash-tables hold their elements in the unpredictable order, and other data structures – in the sorted. If operand sets are ordered and the result isn't, computations do not make difficulties. But if operand set is unordered, the resulting set must be unordered too.

As usage of the pool reduces the strong oscillations of the number of stored elements, the possibility of using the simplest data structure – fixed-size arrays – appears. Without pool its usage does not have a sense, as the sizes of sets might be significantly changed during the simulation. For implementation of set theory operations on the base of arrays the same algorithms as for linked lists may be used.

Unlike linked lists, fixed-size arrays provide fast element insertion possibility, if performed in parallel with set theory operations:

1) It is required to locate expected insertion position of the single element in each array using the binary search algorithm [6-8].

2) Operand arrays are separated on two parts – before located insertion position and after it.

3) The set operation upon first parts of the operands is performed.

4) The single element should be added at the end of resulting array.

5) The set operation upon second parts of the operands should be performed.

The disadvantage of using arrays in pool is that they cannot be effectively used together with other data structures. But the speed of the set operations using the arrays is equivalent to the fastest linked lists, while insertion time is even better than for hash-tables. Also arrays require much smaller amount of the storage memory.

Experiment 4: efficiency assessment for the developed fault lists processing strategy. Fig. 12 shows the performance of deductive fault simulation using different set processing strategy.

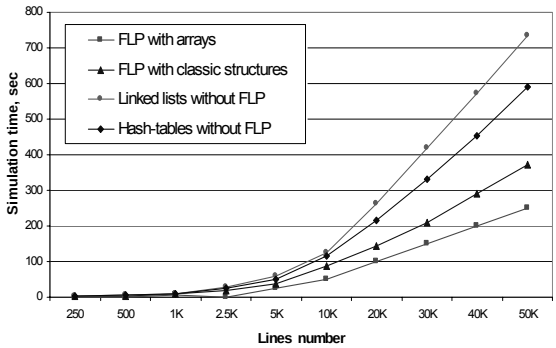

Fig. 12. Performance of the developed FLP-strategy

Experimental results confirm significant advantage (up to 3-4 times) of the developed FLP-strategy for processing fault lists in comparison with the best widely used approaches.

8. Conclusions

Suggested FLP-strategy for implementation of the set theory operations allowed to develop a high-performance deductive fault simulation system for fault coverage estimation of applied tests.

Main scientific and practical results include:

– performance properties assessment of the classic data structures for implementation of the set theory operations;

– selection of the optimal data structures for implementation of set theory operations and development of the FLP-strategy for processing fault lists, providing the maximum speed among with small memory usage;

– joining efficiency of the set operations upon fault lists with the speed of single fault insertion;

– high-performance software implementation of the deductive fault simulation method, acceptable for processing modern large-scale digital system-on-chips, containing millions of logical gates.

References: 1. *Abramovici M., Breuer M.A. and Friedman A.D.* Digital systems testing and testable design". *Computer Science Press*, 1998. **2.** *Armstrong D.B.* Deductive Method for Simulating Faults in Logic Circuits". *IEEE Transactions on computers,* 1972, vol. C-21, 464-471. **3.** *Levendel Y.H., Menon P.R.* Comparison of fault simulation methods – Treatment of unknown signal values // Journal of digital systems, 1980, vol. 4, 443-459. **4.** *Hahanov V., Krivoulya G., Hahanova I., Melnikova O., Obrizan V.* High Performance Fault Simulation for Digital Systems // Proceedings of the Second IEEE International Workshop on Intelligent Data Acquisition and advanced Computing Systems: Technology and Application, 2003, Lviv, Ukraine, 390-395. **5.** *Rossen K.*. Discrete Mathematics and its Applications // McGraw Hill, 2003. **6.** *Gorbatov V. A.* Basic of the discrete mathematics // Moscow, Wyshaja shkola, 1986. **7.** *Stroustrup B.* The C++ programming Language, 3rd edition // Addison-Wesley, 2000. **8.** *Eckel B.* Thinking in C++, 2nd edition, vol.2: standard libraries and advanced topics // Prentice Hall, 2000. **9.** *Meyers S.* Effective and More Effective C++. *Addison-Wesley*, 2001.

ELECTRICAL TEST IS NOT ENOUGH FOR QUALITY

BENGT MAGNHAGEN

JONKOPING UNIVERSITY, SWEDEN

bengt.magnhagen@ing.hj.se

Electrical test means Functional Test (FT), In Circuit Test (ICT) or Boundary Scan Test (BST) or even a combination of these technologies. However, with modern technology, like SMD (Surface Mounted Devices) technology, BGA (Ball Grid Array) components and extremely small component dimensions, electrical test alone does not meet the quality requierments.

Electrical test can not identify bad soldering and bad alignment of components, as examples. Missing decoupling capacitors and so on can not be detected because of it is hard to get physical access for testprobes. Do not forget that digital designs contains a lot of analogue devices!

The tutorial will discuss today test technology with equipment for ICT and BST as well as its pros and cons. And as the addition of this, Inspection. Inspection has traditonally been performed manually but this is not realistic today with board crowded by components. Today Inspection is performed by machine vision. Optical technique named Automated Optical Inspection (AOI) and more advanced X-ray inspection (AXI). AOI and AXI is not the future, it is here today.

EMC /EMI is also a growing challenge and some new ideas will be discussed how to test for these phenomenas.

Підписано до дру*у 9.09.т004. Формат 60*84$^{1}/_{8}$.

Умов. дру*. ар*. ууи4. Облі*.-вид. ар*. у0и0. Зам. № т-875. Тираж №0 прим.

Віддру*овано в навчальнофнау*овому видавничофполіграфічному центрі ХНУРЕ.

6№№6иХар*івипросп. ЛенінаиМ4.