Optimal Hardware Pattern Generation for Functional BIST

Silvia CATALDO, Silvia CHIUSANO, Paolo PRINETTO
Politecnico di Torino, Dipartimento di Automatica e Informatica, Italy
{cataldo, chiusano, prinetto}@polito.it
http://www.testgroup.polito.it

Hans-Joachim WUNDERLICH
Computer Architecture Lab, University of Stuttgart, Germany
wu@informatik.uni-stuttgart.de
http://www.ra.informatik.uni-stuttgart.de

Abstract*

Functional BIST is a promising solution for self-testing complex digital systems at reduced costs in terms of area and performance degradation. The present paper addresses the computation of optimal seeds for an arbitrary sequential module to be used as hardware test pattern generator. Up to now, only linear feedback shift registers and accumulator based structures have been used for deterministic test pattern generation by reseeding.

In this paper, a method is proposed which can be applied to general finite state machines. Nevertheless the method is absolutely general, for sake of comparison with previous approaches, in this paper an accumulator based unit is assumed as pattern generator module. Experiments prove the effectiveness of the approach which outperforms previous results for accumulators, in terms of test size and test time, without sacrifying the fault detection capability.

1. Introduction

To increase functionality, achieve higher performance, and decrease cost, IC manufacters are actually moving quickly towards very deep sub-micron technologies. From one hand, the vast availability of gates permits the integration of a variety of IPs, memories, processors, and analog units on a single chip (System-on-Chip). On the other hand, traditional testing approaches based on an external ATE become more and more unfeasible. In fact, the bandwidth gap between the I/O frequency and the very high internal clock rate often prevents ATEs from testing SoCs *at speed*. Moreover, the number of externally accessible I/O pins, although counting up to several hundreds, strongly limits the controllability and observability of embedded cores.

* This work was supported in part by the Deutsche Forschungsgemeinschaft (DFG) under grant Wu 245/1-3, and in part by the CNR (Italian National Research Council) Short Term Mobility Program.

At the same time, the highly large scaled technology make systems more susceptible to transient and intermittent faults. Often, the target reliability levels require a continuous on-the-fly monitoring, in-field, of the system behavior.

In this scenario, *Built-In Self Test* (BIST) and, more in general, *Embedded Test*, have been widely recognized as effective approaches to SoC testing, moving on board the main functionalities previously carried out by ATEs. In traditional BIST architectures, test pattern generation is mostly performed by ad-hoc circuitry, typically Linear Feedback Shift Registers (LFSRs) [1] [2], cellular automata [3], multifunctional registers, like BILBO [4].

The *functional BIST* test strategy, instead, exploits functionalities and modules embedded into the system itself for test pattern generation. In Figure 1, both module M_i and module M_j are part of the system logic, and during testing M_i is controlled in such a way that its outputs serve as test patterns for module M_j. Typically, M_i is a Sequential circuit used as Test Pattern Generator (STPG) for a given Unit Under Test (UUT), in our case M_j.

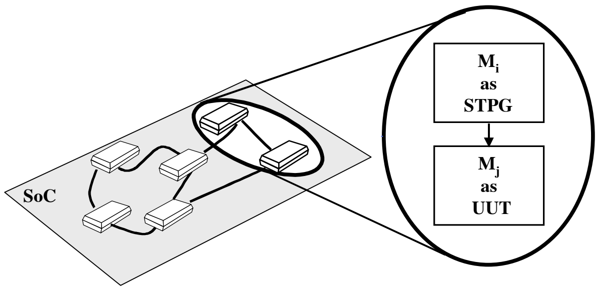

Figure 1: ***Functional BIST***

In previous works, mainly pseudo-random and pseudo-exhaustive test pattern generation has been investigated, since accumulator based circuits for addition, subtraction, multiplication and even more complex modules are able to produce patterns with rather good random properties [5]

[6] [7] [8]. This technique was named as Arithmetic BIST (ABIST) and is comprehensively described in [9].

Most of the approaches presented so far use an accumulator based structure as STPG and assume the UUT being a combinational or full-scan network. [7] presents two computation methods for the initial values (a simulation-based and an analytic one) using a simple adder as arithmetic unit. [10] and [8] include a theoretical analysis of the use of various functional units mentioned above as STPG. Very often, the UUT has random pattern resistant faults, and applying random patterns does not provide sufficient fault coverage. In these cases, a different type of patterns should be applied, and in [5] an accumulator-based method for generating pseudo-exhaustive patterns has been proposed. As only a few circuits are testable by pseudo-exhaustive patterns, a method has been developed to control adder-based structures so that they generate pre-computed deterministic test patterns in an autonomous mode [11].

The present paper proposes a universal method to control and initialize sequential structures so that they work as an STPG for a given unit under test. Suitable test sets are obtained by properly driving the STPG evolution, thus minimizing hardware overhead.

The sequence of values appearing on the STPG outputs is a function of the triplet (σ, δ, τ), as well as of the arithmetic (or logic) function embedded into the block (Figure 2). First the *state register* of the STPG is initialized with an *Initial state* value (δ) and its Primary Inputs (PIs) are fixed at an *Input value* (σ); then the STPG is let evolve for a certain number of clock cycles (τ). The *Initial state* and the *Input value* are often collectively referred to as *Seed* of the STPG.

For sake of efficiency and flexibility, the STPG can be periodically *reseeded*, stopping its evolution and restarting it with a new triplet $(\sigma, \delta, \tau)_i$ until the target fault coverage is reached. In such a case, the global test length is a function of the number of reseedings:

$$\Sigma_{0 \le i \le n} \tau_i .$$

When adopting a functional BIST approach, the designer can trade-off between:

- Maximizing the *fault coverage*;
- Minimizing the overall number of reseedings, in order to reduce the *extra area* needed to store the triplets (e.g., in a ROM);
- Minimizing the *test time*, i.e., the global test length;
- Minimizing the complexity of the *BIST controller*, e.g., by selecting a proper common τ.

In the present paper, we propose a test synthesis algorithm capable of achieving maximal fault coverage with a minimal number of reseeding steps. A prototype tool called *GATSBY* (*G*enetic *A*lgorithm based *T*est *S*ynthesis tool for *B*IST applications) has been implemented to experimentally validate the effectiveness of the approach. The work has been developed under the following assumptions:

- The function performed by the STPG is known and both its primary inputs and its state register are fully accessible, e.g., via full scan;
- The netlist of the UUT is available;
- Single stuck-at faults are the target fault population;

Although the proposed method is absolutely general, for sake of comparison with previous approaches, experimental results have been carried out under the conditions that:

- The UUT is either combinational of full-scan;
- The STPG is an accumulator based unit, including a simple adder as arithmetic logic function.

Experimental results proved that GATBSY either significantly reduces the global test length or minimizes the number of reseedings. The overall structure of GATBSY is presented in Section 2, while Section 3 better details the Genetic Algorithms for computing the reseeding set. Experimental results are discussed in Section 4, and Section 5 eventually draws some conclusions.

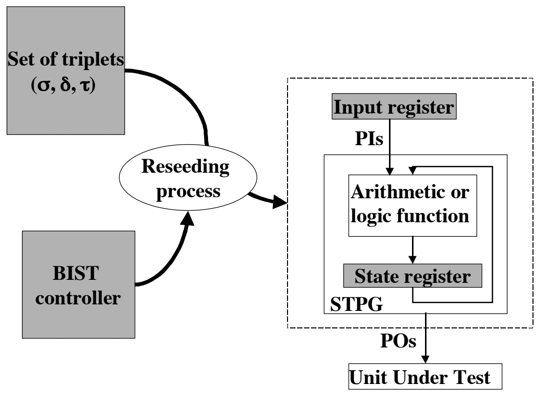

Figure 2: ***The STPG***

2. The Test-Synthesis Tool

Figure 3 sketches the overall architecture of the implemented test synthesis tool, GATSBY; the computation process runs through three different phases:

- We assume to run a gate-level ATPG in a preliminary phase, in order to generate a *target fault list* (single stuck-at fault model) and the relative *deterministic test set*. In Phase 1 an *ATPG Post Processor* elaborates the outputs of the ATPG, generating an *instrumented test set* and a *sorted fault list*. By fault simulation, we compute the detection capability of each pattern with respect to the whole target fault list. The faults are sorted according to their hardness to be tested, i.e., according to the number of patterns each one is covered by. Both the *instrumented test set* and notion about *faults complexity* support the Phase 2 of the computation process.

- Phase 2 is the kernel of the test-synthesis tool. The *Triplets generator* aims at computing a minimum set of triplets $(\sigma, \delta, \tau)_i$ needed to cover the whole *sorted fault list*. The computation is performed resorting to a procedure based on Genetic Algorithms, where the goodness of each triplet is evaluated by fault simulating on the UUT the outputs produced by the STPG, when it is initialized by $(\sigma, \delta)_i$ and let run for τ_i clock cycles. The *instrumented test set* computed in the previous phase, instead, supports the computation opportunely driving the GA evolution. The STPG evolves adding, at each clock cycle, the value σ to the content of its state register. The state register value is a test pattern for the UUT. For a complete analysis about properties of patterns generated by simple adder refer to [8]. The implemented Genetic Algorithm will be detailed in Section 3.
- In Phase 3 a triplet optimizer eventually post-processes the set of triplets generated in the previous phase, aiming at improving it according to a set of predefine *optimization parameters* (e.g., fault coverage, global test length, number of triplets). During the optimization process, first the triplets are *fault-simulated* for a large number of clock cycles, trying to minimize the size of the set. Then, the triplets are fault-simulated in *reverse order*, having as a goal the reduction of the global test length.

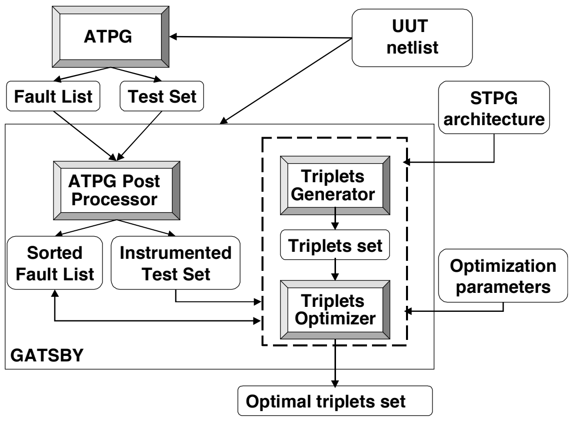

Figure 3: ***The test-synthesis tool architecture***

3. The Genetic Algorithm

Genetic Algorithms (GAs) [12] aim at evolving a *population* of *individuals* in order to increase their goodness (*fitness*). The population evolves through *generations*, based on a mechanism that mimics Nature. In each generation, the reproduction is performed by the exchange of genetic material (*crossover*, *mutation*), and the new individuals must compete with their parents for survival: only individuals having higher fitness values will appear in the next generation. The population improvement is based on the capability of generating individuals able of inheriting and merging the best characteristics of their parents.

To exploit the GAs in our specific application, we encode an *individual* as a single triplet; the *population* is thus a set of candidate triplets, and the evolution process aims at improving the goodness of all of them with respect to the *evaluation parameters*. At the end of the computation process, the optimal solution is extracted from the last population as a minimal subset of triplets guaranteeing the coverage of all the target faults.

The developed GA is structured in two nested main loops (Figure 4). The inner one traces a typical GA structure: at each generation a set of new individuals (triplets) is created, starting from existing ones, through the *evolution rules* (line 3). Then the quality of the individuals is assessed (line 4) and the fitness functions values are used to rank the population (line 5).

As pointed out before, the GA-based tool works toward reaching the 100% target fault list coverage. To further guide the GA evolution, we embed the quite standard inner loop into an ad-hoc defined outside loop. Based on our approach, the GA focuses first on a small subset of very complex faults. Then, whenever one execution of the inner loop ends, an incremental percentage of easier faults is targeted by the GA (line 2), and eventually the whole target fault list is take into account. The “incremental” fault list strategy allows a sort of tuning process, in which the population is first customized on hard faults, and it is progressively slightly modified to detect easier ones, as well.

The GA evolution takes advantages from the pre-processing elaboration of Phase 1. To set up an individual, a pattern is extracted from the *instrumented test set* and it is used as *Initial state* value (δ) of the triplet, where the *Input value* (σ) is chosen randomly. This technique is used to initialize the first population (line 1), and later to generate, during the reproduction process, a subset of new individuals (line 3). Rather than starting from a random population, the GA can therefore immediately starts dealing with a set of already good individuals, and work to improve them.

```
1:Population initialization
  do{/*outside loop*/
2: Update the set of addressed faults
   {/* inner loop*/
3:  Compute new Individuals (σ, δ, τ)i
4:  Compute Fitness
5:  Select the next Population
6:  generation ++;
   }until generation < MAX
7:}until not addressed all the faults
```

Figure 4: ***Genetic Algorithm skeleton***

3.1. Evaluation Parameters and Fitness Function

The fitness function is expressed as goodness of the test set obtained processing the triplet by means of the STPG. When computed, the test set is fault simulated according to a *target fault list*, and using a standard gate-level event-driven fault simulator. During the fault simulation process three different evaluation parameters are measured:

- The *fault coverage* value, expressed as percentage of covered faults on the target fault list.
- The *circuit sensitization* parameter, to estimate how near the test set is in increment its actual fault coverage. The circuit sensitization is the number of logic differences between the good and faulty machine injected by a pattern, in the presence of a target fault. The test set circuit sensitization is obtained summing up the maximum circuit sensitization values of its test patterns.
- The number of patterns not allowing the detection of any new fault (*dummy patterns*). Large numbers of dummy patterns decrease the test set efficiency in terms of test time, since only few applied patterns contribute in testing the UUT. Our intent is thus to reward individuals with fewest dummy patterns.

Our fitness function consists in a multiple reordering procedure. Individuals are sorted mainly based on how they contribute to the fault coverage of the whole population. The triplet t_i with highest fault coverage is stored into the next population. Among the remaining individuals, a second triplet t_j, covering the largest subset of faults not detected by t_i, is selected. Then, a third triplet t_k is chosen, guaranteeing maximum fault coverage on the faults undetected by t_i and t_j. The circuit sensitization and the dummy patterns number, instead, are used to distinguish among individuals with same fault coverage. The test length τ is not directly taken into account since it is kept constant for all the triplets of the population.

3.2. Evolution Rules

The generation of new individuals is based on the following genetic operators:

- The *horizontal*, *two-cut crossover* operator: the seed of the new individual is obtained by combining portion of the seeds of either parents, according to the position of two randomly generated cut-points x_1 and x_2.
- *Mutation operators*: to further introduce variability in the evolution process, according to a mutation probability, the bits in the seed of a given triplet are flipped. The operator is applied on the outputs of the crossover operation, and on few individuals of the actual population.

Finally, based on the same strategy exploited to set up the first population, new individuals are obtained setting the seeds with a test pattern extracted from the instrumented test set: the pattern is used as *Initial state* value (δ) of the triplet, where the *Input value* (σ) is randomly chosen. The selected pattern must allow the detection of faults currently addressed by the GA, but not covered by the best individual of the population.

4. Experimental Results

The proposed algorithm has been implemented in ANSI C as a prototype tool named GATBSY. To assess the efficiency of the tool, experiments were performed on the ISCAS'85 [13] and the ISCAS'89 [14] (full scan version) benchmarks, which are circuits not randomly testable by 10K patterns. The STPG is assumed having a state register size customized on the UUT input-bits parallelism. In the present paper, the gate-level ATPG Sunrise [15] is used to compute the fault list and the deterministic test exploited in the GATSBY computation process. Moreover, the Sunrise fault coverage represents a reference value to assess the quality of the GATSBY solutions.

Table 1 reports the genetic parameters used to set up the experiments with GATSBY. The Table shows the number of generations, and the number of new individuals created by crossover, mutation and using the instrumented test set.

Table 2 presents the results obtained running GATSBY on a SparcStation 5/110 with 64Megabytes of RAM, and limiting the computation time up to 30 hours. For each benchmark, we report both the reseeding solution expressed as the cardinality of the triplets set (*triplets*), and the number clock cycles (τ) for which each triplet must be let evolve. Regardless the low number of triplets, we experimentally verified that our solution guarantees, on all the circuits (except on the s9234), the same detection capability of the Sunrise test set. In particular, for the c7552 and the s1494 circuits, our tool outperforms the Sunrise test set fault coverage. Table 1 also reports the test length reduction obtained simulating the triplets in reverse order: the improvements are significant and range from 2% to 54%.

Two additional figures allows to deeply analyze the results of Table 2. Taking as an example the s838 circuit, Figure 5 shows how the triplets contribute to the final fault coverage. Few triplets are needed to approximate the maximum fault coverage value, detecting a large number of easy faults. Most of the triplets, instead, are included into the set to cover the remaining hard faults; such triplets contribute in a limited, but crucial, percentage to the final fault coverage.

Figure 6 focuses on the trade-off between number of triplets and global test length. On the s5378 circuit, the maximum fault coverage can be guaranteed by a small set of triplets, where each triplet is let evolve for a large number of cycles τ, as well as a larger set of triplets with lower τ.

Experiments therefore show that the number of reseedings, required to guarantee the maximum fault coverage, can be reduced incrementing the value of τ, while decrementing τ the same fault coverage is obtained using a larger set of triplets.

GATSBY allows the designer to exploit such a trade-to, identifying a suitable solution according to the given constraints. Opportunely setting the optimization parameters, in fact, GATSBY can be driven to generate a solution characterized by either a larger test time or more reseedings steps.

Eventually, a comparison between GATSBY and some previously published results is presented.

Table 3 compares the amount of test data storage required by GATSBY and [11]. [11] assumes the UUT be a hard core and, resorting to symbolic techniques, computes the minimum triplets to reproduce a deterministic test set. Experiments show that in 11 cases out of 14 GATSBY is able to either outperform or provide the same results of [11].

Table 4 compares GATSBY and [7] in terms of test length, when just a single triplet is considered. GATSBY always outperforms [7], nevertheless GATSBY mainly addresses the minimization of the triplets set, and [7] presented a method ad-hoc developed for test length minimization. Although using a single triplet the maximum fault coverage is guaranteed, where on the circuits not dealt by [7] GATSBY approciates, quite closely, the target Fault Coverage.

5. Conclusions

This paper presented an approach to compute an optimal reseeding for a sequential test pattern generator.

A prototypical tool, named GATSBY, has been implemented to sperimentally validate the goodness of the methodology.

Experiments proved that GATSBY allows to outperforms results presented so far in literature, reducing the number of reseedings or the global test length, guaranteeing at the same time the whole detection of non random testable circuits. The proposed methodology has been applied to the case of an accumulator based STPG, but can be extended to deal with a more general functional unit.

6. Acknowledgments

The authors wish to thank A. Bergadano for implementig GASTBY, and A. Benso and R. Dorsch for the fruitfull discussions concerning the development of the tool.

7. References

[1] S. Venkataraman, J. Rajski, S. Hellebrand, S. Tarnick, *An Efficient BIST Scheme Based on Reseeding of Multiple Polynomial Linear Feedback Shift Register*, IEEE International Conference on Computer-Aided Design, 1993, pp. 572-577

[2] H.J Wunderlich, G. Kiefer, *Bit-flipping BIST*, IEEE International Conference on Computer-Aided Design, 1996, pp. 337-343

[3] S. Chiusano, F. Corno, P. Prinetto, M. Sonza Reorda, *Cellular Automata for Deterministic Sequential Test Pattern Generation*, IEEE VLSI Test Symposium, 1997, pp. 60-65

[4] M. Abramovici, M.A. Breuer, A.D. Friedman, *Digital System Testing and Testable Design*, Computer Science Press, 1990

[5] S. Gupta, J. Rajski, J. Tyszer, *Arithmetic Adaptive Generators of Pseudo-Exhaustive Test Patterns*, IEEE Transactions on Computers, vol. 8, num. 45, pp. 939-949, August, 1996

[6] J.Rajski, J.Tyszer, *Multiplicative Window Generators of Pseudo Random Test Vectors*, IEEE European Design & Test Conference, 1996, pp. 42-48

[7] A. P. Stroele, F. Mayer, *Methods to reduce Test Application Time for Accumulator-Based Self –Test*, IEEE VLSI Test Symposium, 1997, pp. 48-53

[8] A. P. Stroele, *Arithmetic Pattern Generation for Built-In Self Test*, IEEE International Conference Computer Design, 1996, pp. 131-134

[9] J. Rajski, J. Tyszer, *Arithmetic Built-In Self-Test for Embedded Systems,* Prentice Hall, 1998

[10] S. Gupta, J.Rajski, J. Tyszer, *Test Pattern Generation Based on Arithmetic Operations*, IEEE International Conference on Computer-Aided Design, 1994, pp. 117-122

[11] R. Dorsch, H.-J. Wunderlich, *Accumulator Based Deterministic BIST*, IEEE International Test Conference, 1998, pp. 412-421

[12] D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989

[13] F. Brglez and H. Fujiwara, *A Neutral Netlist of 10 Combinatorial Benchmark Circuits*, IEEE International Symposium on Circuits and Systems, 1985

[14] F. Brglez, D. Bryan, K. Kozminski, *Combinatorial Profiles of Sequential Benchmark Circuits*, IEEE International Symposium on Circuits and Systems, 1989, pp. 1129-1234

[15] Sunrise Reference Manual, *Sunrise Test Systems*, 1995

Parameter	Value
Generations	100
Population size	16
New individuals at each generation	20
New individuals by crossover	10
New individuals by mutation	4
New individuals by instrumented test set	6

Table 1: ***Genetic parameters***

	GA based tool and Post Processing Simulation				Rev. Order Sim.
Circuit	# Triplets	τ	FC_{GATSBY}%	$FC_{Sunrise}$% - FC_{GATSBY}%	Test length reduction
c432	1	243	99.12	0	0%
c499	1	398	98.84	0	-7.54%
c880	1	2,175	100	0	-3.26%
c1355	1	1,354	99.47	0	-14.99%
c1908	1	4,500	99.61	0	-15.98%
c2670	33	2,000	95.64	0	-35.95%
c3540	1	3,548	96.03	0	-2.28%
c5315	1	1,324	98.84	0	-2.49%
c6288	1	58	99.56	0	-3.45%
c7552	64	4,000	98.19	-0.11	-12.41%
s420	6	3,000	100	0	-31.55%
s641	5	3,000	100	0	-54.61%
s713	5	5,000	93.46	0	-34.55%
s820	3	2,000	100	0	-11.48%
s832	2	4,000	98.31	0	-15.95%
s838	11	2,000	100	0	-38.92%
s953	3	4,000	100	0	-15.39%
s1196	4	4,000	100	0	-37.50%
s1238	4	2,000	94.74	0	-8.05%
s1423	3	5,000	98.99	0	-34.38%
s1494	1	2,000	99.19	-0.07	-6.60%
s1512	24	2,000	100	0	-11.07%
s5378	7	5,000	99.05	0	-25.17%
s9234	50	2,000	89.41	3.75	-13.73%

Table 2: ***GATSBY results***

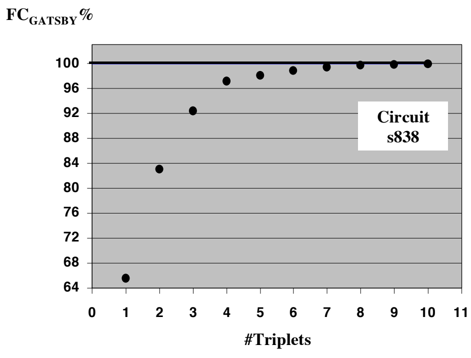

Figure 5: ***Fault coverage vs. Number of Triplets***

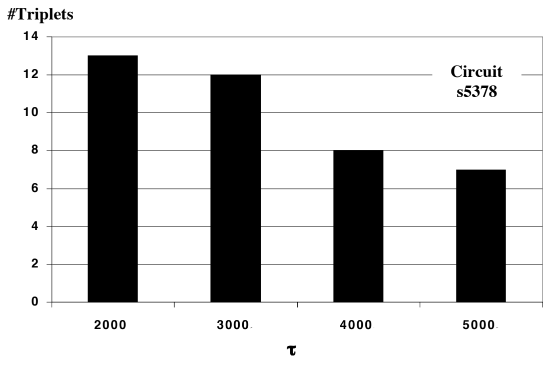

Figure 6: ***Number of Triplets vs. Test Length***

Circuit	GATSBY	[11]
c2670	33	18
c7552	64	23
s641	5	6
s713	5	5
s420	6	7
s820	3	8
s832	2	8
s838	11	26
s953	3	3
s1196	4	5
s1238	4	5
s1423	3	3
s5378	7	8
s9234	50	15

Table 3: ***Number of triplets w.r.t.* [11]**

	GATSBY		[7]		
Circuit	$FC_{Sunrise}$%-FC_{GATSBY}%	τ	$FC_{Sunrise}$%-$FC_{[8]}$%	τ	τ reduction
C432	0	243	0	256	-5.07%
C499	0	368	0	544	-32.35%
C880	0	2,104	0	2,272	-7.4%
C1355	0	1,237	0	1,280	-3.4%
C1908	0	3,781	0	3,872	-2.35%
C2670	7.63	2,000	not dealt in [7]		-
C3540	0	3,467	0	7,872	-56%
C5315	0	1,291	not dealt in [7]		-
C6288	0	56	0	96	-41.66%
C7552	4.24	2,000	not dealt in [7]		-

Table 4: ***Test length reduction by using a single triplet w.r.t.* [7]**