University of Baghdad
College of Engineering
JOURNAL OF ENGINEERING

Journal of Engineering

journal homepage: www.jcoeng.edu.iq

Volume 32 Number 5 May 2026

Accelerating Sequence Covering Array Generation and Optimization via Differential Evaluation and Process-Level Parallelism

Zainab A. Najm, **Mohammed Issam Younis***

Department of Computer Engineering, College of Engineering, University of Baghdad, Baghdad, Iraq

ABSTRACT

Sequence Covering Arrays (SCAs) are an extension of combinatorial testing configurations designed to catch ordering faults. An SCA guarantees that all ordered t-way interactions exist as subsequences within the configuration. Generating and optimizing SCAs is computationally expensive, which has hindered their use in practice. This work presents a novel acceleration framework tailored for SCA construction routines. The key idea is to apply algorithm restructuring to existing routines to achieve gains without modifying their underlying semantics. The acceleration framework utilizes differential evaluation to prevent redundant global computations, leverages fixed-size tensors to streamline coverage management, and employs process-level parallelism to allocate independent evaluations over multiple CPU cores. Across all test scenarios, the performance improvements were clearly evident. Phase 1 shows marginal gains from purely algorithmic acceleration but demonstrates a speedup of 2.58× to 3.80× end-to-end when implemented in its accelerated and parallel form. However, phase 2 shows significant improvement from algorithmic restructuring yielding 2.21× – 10.41× speedup. When accelerated and parallelized, phase 2 shows 80.65× – 259.70× speedup end-to-end across all configurations. Best case speedup observed was 259.70×. These speedups demonstrate that scalable SCA construction is possible through restructuring for parallel evaluation while maintaining coverage correctness.

Keywords: Sequence Covering Arrays (SCAs), Combinatorial testing, Algorithmic acceleration, Differential evaluation, Tensor-based representation, Process-level parallelism

1. INTRODUCTION

Failures within systems are often dependent not only on parameter values but on event ordering. Existing combinatorial testing techniques based on unordered interactions are therefore inadequate for finding these faults, even though these techniques are pervasive in combinatorial testing tools **(Nie and Leung, 2011; Petke et al., 2015)**. Sequence-aware testing approaches were first defined by **(Kuhn et al., 2012)**, which led to Sequence

*Corresponding author
Peer review under the responsibility of University of Baghdad.
https://doi.org/10.31026/j.eng.2026.05.09

Article received: 07/02/2026
Article revised: 22/04/2026
Article accepted: 23/04/2026
Article published: 01/05/2026

Covering Arrays (SCA) being introduced by **(Chee et al., 2013)** as an extension of Covering Arrays guaranteeing each ordered t-way interaction is covered by at least one sequence. Order aware testing techniques have been shown useful at modeling failures that depend on event order; however, they also come with significant computational cost during both generation and optimization. Prior efforts have focused on heuristics and metaheuristics for improving the construction quality, such as Dynamic Event Ordering **(Younis, 2020)**, fish swarm optimization **(Rahman et al., 2020)**, or the Barnacles Mating Optimizer **(Zamli and Kader, 2021)**. Few works have focused on reducing repeated evaluation or have improved indexing/sampling time complexity during interaction growth. Works within both software testing and combinatorial testing have demonstrated improvements in performance through incremental evaluation **(Li et al., 2022; Torbunova et al., 2024)**, structured representation of test cases/datapoints **(Fan et al., 2023; Salehi et al., 2023)**, and parallelization of underlying methods/processes **(Bombarda et al., 2023; Castro et al., 2024)**.

The existing approaches have mostly existed in isolation, lacking the unified framework needed to speed up coverage of ordered interactions in SCAs. In fact, current methods for SCAs continue to perform repeated full recomputation of coverage during both construct and optimization operations. Motivated by these needs, this work introduces an acceleration-oriented framework for Sequence Covering Array generation and optimization to improve computational performance without modifying the underlying construction heuristic. This approach draws from three complementary techniques: (i) differential evaluation to only recompute ordered interactions affected by change, (ii) fixed-size indexed coverage representation allowing efficient access to coverage information, and (iii) process level parallelization of independent candidates.

This work makes the following contributions:

- A unified acceleration framework for Sequence Covering Array generation and optimization phases that does not modify the underlying construction heuristic.
- Differential evaluation strategy limiting recomputation to locally modified ordered t-interactions.
- Fixed-size indexed coverage representation allowing for efficient access.
- Process-level parallel candidate evaluation model.
- Experimental evaluation showing improved scalability in both phases.

2. METHODOLOGY

This paper presents how Sequence Covering Array (SCA) generation was parallelized. Rather than implementing new generation or optimization heuristics, the technique described here further breaks down the original problem's definition. The existing sequential execution pipeline is modified to reduce redundant work and leverage inherent parallelism found in multi-core architectures.

The serial baseline pipeline consists of two parts generation and optimization that have fundamentally different serial characteristics and dominating costs associated. As such, each part's execution is profiled separately in the serial model described below. Acceleration strategies are then described with respect to each baseline computation. Correctness and coverage guarantees are maintained. **Fig. 1** provides an overview of the proposed approach, including the baseline SCA pipeline, evaluation bottleneck of primary interest, accelerations introduced in this work, and execution modes analyzed for performance.

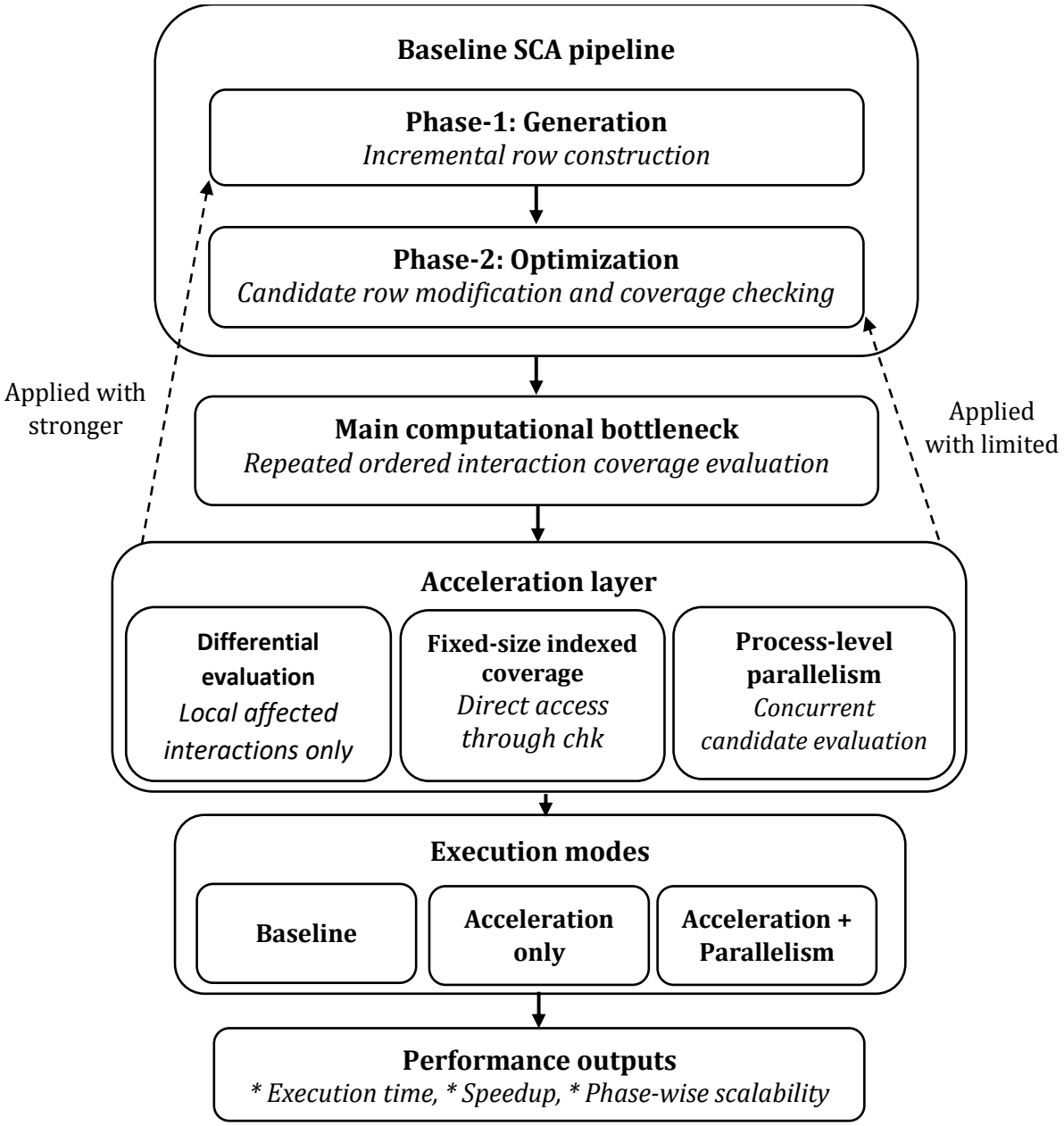

Figure 1. Acceleration workflow overview for SCA generation and optimization

2.1 Pipeline Description

Within the SCA construction pipeline, the two crucial computational phases are generation and optimization. This pipeline pattern is prevalent in combinatorial test generation/improvement processes **(Nayeri et al., 2013; Lara-Alvarez and Avila-George, 2015; Demiroz and Yilmaz, 2016)**. Phase 1 incrementally builds sequences in an attempt to generate an initial covering array, achieving full t-way ordered interaction coverage. Sequence generation follows ordering restrictions and is also governed by a feasibility condition tied to the present construction phase. Phase 2 incrementally improves the test suite generated during construction in order to reduce redundancy, subject to coverage. The optimization phase employs algorithms to explore potential test-suite alterations, guided by a specific neighborhood definition. Determining whether coverage is

preserved requires reevaluation against the current ordered interaction coverage state for each modification. While both pipeline phases necessitate the manipulation and querying of coverage information, they differ drastically in execution structure and dominant computation. For this reason, we analyze each phase independently before presenting acceleration and parallelization strategies.

2.2 Baseline Sequential Execution

Sequential pipeline execution can be broken down into two distinct phases: Phase 1 and Phase 2. Generation creates an initial covering array by greedily constructing feasible test sequences satisfying a number of ordering constraints.
Optimization tries to eliminate the redundancy in the set of test sequences created in Phase1. It is usually solved by applying metaheuristic and search-based methods **(Torres-Jimenez and Rodriguez-Tello, 2012; Zeng, 2016; Sabharwal et al., 2016; Li et al., 2022).** While both phases operate on coverage information, generation decisions are locally dependent on ordering constraints and greedy heuristics. Optimization procedures search through many possible modifications to find reduced test suites.
Decisions made during optimization have less dependence on previous decisions and are considered with respect to the current coverage state.

2.2.1 Phase 1: Generation

Sequence construction is incremental and relies on greedy or one-test-at-a-time methods similar to those commonly employed in covering array generation **(Kuhn et al., 2012; Younis, 2020)** as well as general combinatorial testing strategies **(Izquierdo-Marquez et al., 2018; Nasser et al., 2018; Zabil et al., 2018; Fadhil et al., 2023)**. Combinatorial and constraint-based approaches, including U-CIT and its offshoots, have also served as a basis for exploring comparable construction heuristics **(Mercan, 2021)**. During each iteration of sequence construction, the algorithm picks the next allowable event/parameter value to extend the partial sequence, given the current state of construction and uncovered ordered interactions remaining. When a value is selected for inclusion, it instantly limits the subsequent options that can be considered. Because the next step builds upon the expanded sequence, prior decisions shape each selection. Execution time during generation is spent sequentially constructing candidate sequences while meeting strict ordering requirements and is bottlenecked by feasibility checks that must access many uncovered interactions.

2.2.2 Phase 2: Optimization

The goal of optimization is to reduce any duplication found within the Phase 1 test sequence results. The most frequent solutions to this specific optimization task come from metaheuristic and hybrid techniques **(Torres-Jimenez and Rodriguez-Tello, 2012; Sabharwal et al., 2016; Li et al., 2022; Zeng et al., 2016; Ahmed et al., 2017)**. Unlike the generation phase, optimization examines a broad range of potential changes at each stage. Although generating modification candidates is inexpensive, calculating their effect on ordered interaction coverage necessitates substantial repeated calculations. Every candidate must be evaluated against the current coverage state to determine if it can be accepted, and only a fraction of evaluated candidates is accepted into the suite. Similar strategies have been employed extensively for post-processing as well as test suite minimization techniques **(Sheng et al., 2018; Guo et al., 2023)**. Runtime is therefore dominated not by generation, but by repeated impact evaluation and coverage verification following each accepted

candidate. For the sequential execution model, this implies repeated scans and recomputation of the full set of ordered interactions. Thus, the cost considerations differ significantly between these two phases. Because generation incrementally constructs rows by appending to initially empty sequences, each row must wait for the previous to complete before beginning. Runtime is therefore dominated by strict sequential dependence imposed by row construction. Optimization, in contrast, is spent evaluating many candidate modifications against the contents of the suite.

2.3 Algorithmic Acceleration

Repeated coverage evaluation is the main performance issue within the sequential SCA pipeline. Similar issues with performance, arising from repeated assessments, have also surfaced in combinatorial interaction testing and sequence-based optimization **(Islam et al., 2023; Witharana et al., 2024)**. Each candidate is evaluated during generation and optimization steps that may cause recomputation of the entire coverage state even if the candidate only modifies coverage for a small number of interactions. Recomputation can be avoided by transforming coverage evaluation to operate only on affected portions of the coverage state. Not only does this decrease the runtime of each evaluation, but the evaluation itself can be transformed to improve performance. Thus, this method, or algorithmic acceleration, proceeds in two distinct phases. The first phase, differential evaluation, limits the amount of computation that must be done. The second phase replaces the naive coverage representation with a fixed-size representation that is faster to access and update.

2.3.1 Differential Evaluation

The baseline evaluation strategy involves full-scale calculation of coverage. The effect of candidate modification is determined by re-computing coverage over all interactions. Therefore, most interactions are computed redundantly since local modification usually only affects coverage of nearby interactions. Differential evaluation solves this problem by limiting computation to only interactions whose coverage state could have been altered by the modification. Equivalent ideas have been investigated for incremental and delta-based techniques to minimize recomputation in testing and combinatorial systems as well **(Li et al., 2022; Bohm et al., 2024; Witharana et al., 2024; Torbunova et al., 2024)**.

If $\boldsymbol{chk}$: the coverage state of ordered interactions.
$\boldsymbol{chk}_{\{old\}}$: current coverage state before modification
$\boldsymbol{chk}_{\{new\}}$: updated coverage state after modification
$\boldsymbol{S}$: current test suite
$\boldsymbol{r}$: modified row (addition, removal or updated sequence)

$$\boldsymbol{chk}_{\{new\}} = \boldsymbol{f}(\boldsymbol{all\ S})$$

which re-computes the entire coverage state using all existing sequences.
Differential evaluation seeks only to compute the local impact of the modification:
$\boldsymbol{\Delta}$: set of coverage changes induced by r

$$\boldsymbol{chk}_{\{new\}} = \boldsymbol{chk}_{\{old\}} \oplus \boldsymbol{\Delta}$$

Using $\oplus$ the system updates solely those entries whose coverage might be impacted. **Fig. 2** shows how differential evaluation differs from full recomputation.

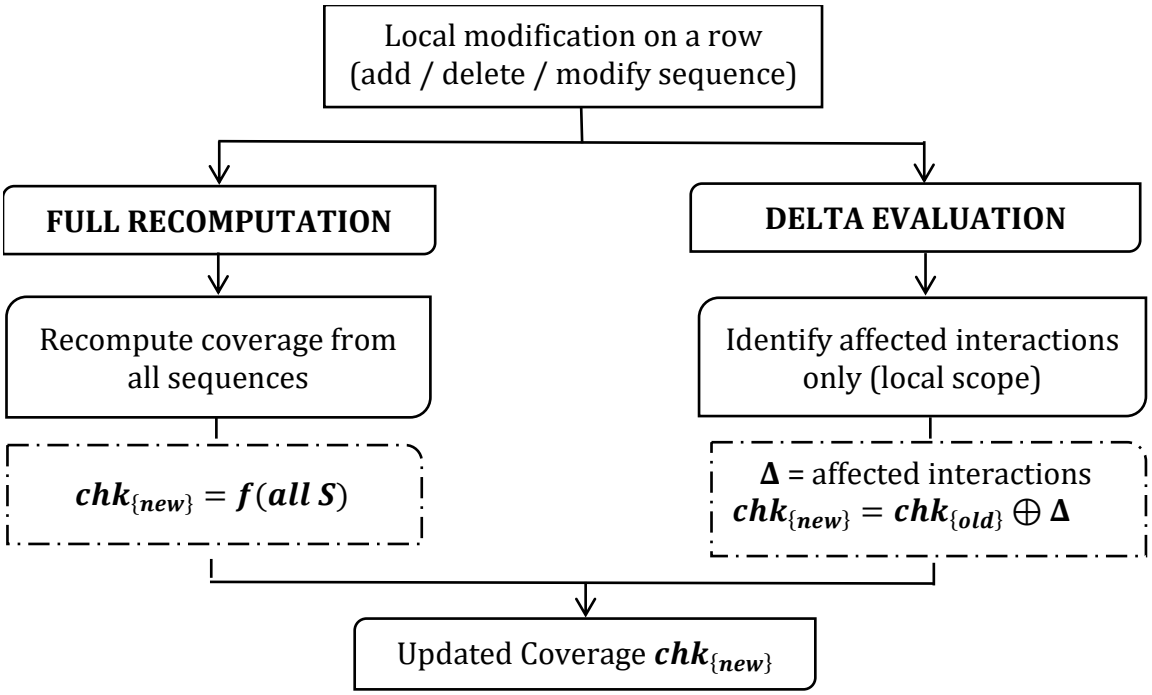

Figure 2. Differential evaluation for updating coverage locally

The distinction here is that full recomputation considers every interaction space while differential evaluation only considers the subset of interactions that changed. There is no unnecessary computation and, as such, evaluation becomes much cheaper. This becomes incredibly valuable during the optimization phase, where frequent alterations are the norm. Differential evaluation decreases the number of interactions we need to evaluate, but efficient implementation still relies on coverage state representation in memory.

2.3.2 Fixed-size Coverage Representation

Coverage information in baseline implementation is represented using Python lists, sets and dictionaries. Structured representations have been demonstrated to enable efficient scaling to high-dimensional interactions as well as reducing redundancy in state representations **(Fan et al., 2023; Salehi et al., 2023; Yang et al., 2023)**. These data structures have inherent overhead associated with dynamic memory allocation, object lookup, hashing and irregular memory access patterns. This overhead is costly when coverage information is being queried and updated frequently, as occurs during optimization.

Fig. 3 Clarifies the shifting from a dynamic coverage setup to fixed-size index lookups. Switching to indexed access within a fixed-size structure provides a robust computational base, enabling efficient execution and paving the way for parallel evaluation.
A quick example helps to further highlight this change. As an example, let it be examined how coverage is determined for the ordered interaction (i, j, k). In the baseline representation, this interaction must be accessed in a dynamic data structure to determine coverage. By employing our tensor representation, this interaction's fixed location is known, making indexed access immediate. On repeated evaluations, we no longer need to look up each affected interaction; we simply access the interaction via its position.

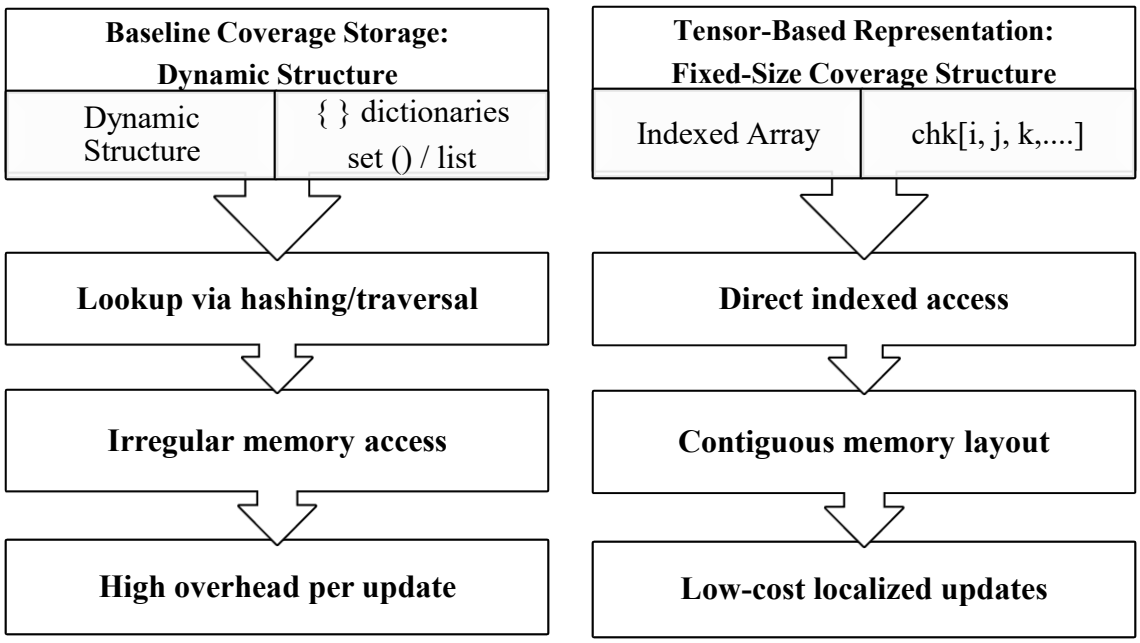

Figure 3. Mapping from dynamic coverage representation to fixed size index lookup

2.4 Parallel Execution Model Interpretation

Since candidate modifications are independent, parallel evaluation is only employed at the evaluation stage. Process-level parallelism has been utilized extensively to speed up computationally expensive workflows in testing and optimization **(Bombarda et al., 2023; Castro et al., 2024; Lee and Kim, 2020; Python Software Foundation, 2023)**. Rather than considering candidates one at a time, the approach sends candidates to worker processes so that many evaluations can occur simultaneously.

The simplified parallel workflow above can be described:

A. Generate many candidate modifications from the current suite of tests

B. Broadcast candidates to workers for parallelization

C. Each worker, given a candidate:

- Locates affected interactions,
- Look up affected interactions in chk's index
- Calculate the local coverage impact via differential calculation.

D. The workers then transmit their individual findings

E. Wait for all results to return. Race them to acceptance.

F. Commit accepted candidate to global coverage state atomically

Scalability follows directly from this structure. Because Phase 1 relies heavily on incremental construction and ordered commits, there are naturally serial bottlenecks limiting parallel speedup. During Phase 2, evaluation dominates runtime. Localized candidate evaluation and indexed lookup create high volumes of independent, inexpensive tasks.

This form of behavior is shown schematically in **Fig. 4**. Amdahl's law is formulated to express the theoretical speedup in latency of the entire system of interest, when only part of the system is improved **(Amdahl, 1967).** It is defined by Eq. (1):

$$S(P) = \left(\frac{1}{f+\frac{1-f}{P}}\right) \quad (1)$$

Where '*f*' is the serial fraction of execution and P is the number of processors.
Similarly, Gustafson's law is used to describe scalability when enlarging the problem size linearly with the number of processors **(Gustafson, 1988)**. It is defined by Eq. (2):

$$S(P) = P - f(P - 1) \quad (2)$$

As shown in **Fig. 4**, the gap between these models is representative of the breakdown from serial limited to purely parallel scaling. This disparity is addressed in our framework by leveraging differential updates and tensor-based representations to split the evaluation, assigning each independent candidate evaluation to a separate process, and synchronizing the accepted updates as a whole.

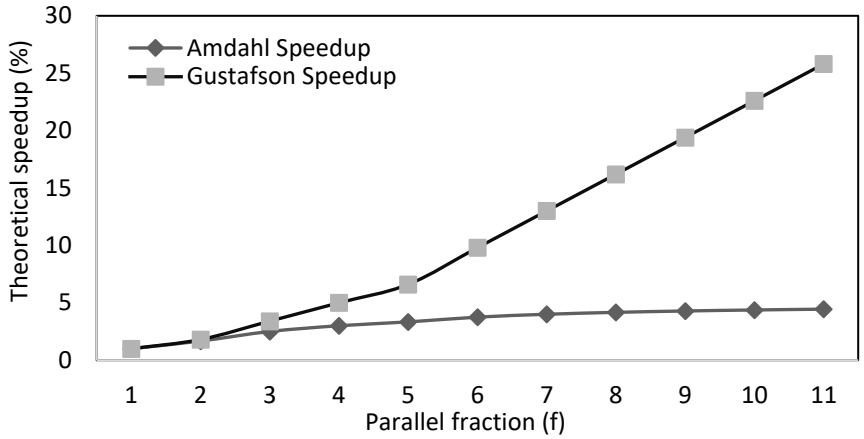

Figure 4. Amdahl and Gustafson Scaling Models

2.5 Running Example

Let it be illustrated with this trace set.

- e_0: Initialize device
- e_1: Authenticate user
- e_2: Load configuration
- e_3: Send data packet

Suppose t = 3. The current test suite is: S = {r_1, r_2} Where: r_1 = (e_0, e_1, e_2, e_3), r_2 = (e_0, e_2, e_1, e_3)
That is,

- r_1: Initialize device → Authenticate user → Load configuration → Send data packet
- r_2: Initialize device → Load configuration → Authenticate user → Send data packet

Each row would contribute ordered t-way interactions to the suite-level coverage matrix chk. For t = 3, some examples of what this looks like are:

r_1 contributes: (e_0, e_1, e_2), (e_0, e_1, e_3), (e_0, e_2, e_3), (e_1, e_2, e_3) and
r_2 contributes: (e_0, e_2, e_1), (e_0, e_2, e_3), (e_0, e_1, e_3), (e_2, e_1, e_3)
Since coverage for the entire test suite is maintained in chk, only a portion of that coverage is calculated per-row.

2.5.1 Phase 1: Generation

Now, suppose we generate a candidate row during generation:
r_3 = (e_1, e_0, e_2, e_3) or Authenticate user → Initialize device → Load configuration → Send data packet

Baseline implementation checks its contribution against suite coverage via repeated global evaluation.
Accelerated implementation looks only at the interactions contributed by r3 and updates only their indexed entries in chk.

2.5.2 Phase 2: Optimization

Select one existing row r_1 from the suite, and attempt a local modification to it:

$r_1 = (e_0, e_1, e_2, e_3) \rightarrow r_1' = (e_0, e_2, e_1, e_3)$

i.e. Initialize device → Authenticate user → Load configuration → Send data packet
becomes
Initialize device → Load configuration → Authenticate user → Send data packet
Note that this only modifies a portion of r_1's contribution.
(E.g. affected interactions are: (e_0, e_1, e_2), (e_1, e_2, e_3) … but other interactions already covered by other rows in the suite may remain covered.)
In the baseline implementation, the effect of this change is validated against the suite-level coverage state by broader recomputation.
In the optimized implementation, only the interactions locally affected by the modification to r_1 are recomputed, and only those indexed entries in chk are updated.

2.5.3 CPU Parallelism using the Same Example:

During the optimization phase, it is possible that there are several candidate row modifications to evaluate as part of the same work loop. For example, if both r_1 and r_2 each have candidate modifications $r_1 \rightarrow r_1'$, $r_2 \rightarrow r_2'$, then these candidate rows need not be processed sequentially. Instead:

- Worker 1 processes affected interactions of $r_1 \rightarrow r_1'$
- Worker 2 processes affected interactions of $r_2 \rightarrow r_2'$

Where each worker only checks the local interaction differences caused by its candidate row and only accesses the corresponding indexed entries in chk. Then return those local results, and once the winner is selected, that modification is applied globally to coverage state chk in a single update. What this example shows:

- The entire test suite is the set of rows S
- The coverage state is stored globally in chk
- Phase 1 incrementally adds contributions from new rows to S
- Phase 2 modifies contributions from existing rows in S
- Differential evaluation limits comparison to added/affected interactions
- Fixed-size indexed representation maps directly to interactions in chk
- CPU parallelism can process multiple candidate rows in parallel

3. EXPERIMENTAL SETUP

3.1 Hardware and Software Environment

Experiments were performed on a personal desktop/workstation with the following characteristics:

- **Operating system**: Windows 11
- **Processor:** Intel(R) Core (TM) Ultra 7 258V
- **Physical cores:** 8

- **Logical processors:** 8
- **RAM:** 16 GB

Runtime was determined by wall-clock time and presented in seconds. All numbers presented are the average of several runs.

3.2 Experimental Setup

Two groups of experiments were performed, each serving as a complement of the other:

3.2.1 Problem Size Scaling

The data in **Tables 1–4** detail the performance of the various implementations when problem size is scaled up. The interaction strength (t) remained constant at a value of 4 throughout these experiments, while we modified k.

- **Table 1** presents the sequential baseline alongside its parallel counterpart, both operating on the unaccelerated baseline.
- **Table 2** details the performance of the accelerated version when parallelism is not employed.
- **Table 3** shows the accelerated implementation with parallelism.
- **Table 4** shows speedup values corresponding to the other tables, using the sequential baseline as a reference.

3.2.2 Speedup Scaling with Worker Count

Tables 5 to 7 studied how speedup scales as the number of parallel workers varies. Here we set N workers $\in \{2,4,6,8\}$ and fixed t, k to the following values for each table:

- **Table 5:** t = 3, k = 30
- **Table 6:** t = 4, k = 12
- **Table 7:** t = 5, k = 10

The two phases' runtimes are compared, contrasting the unaccelerated baseline with the accelerated and parallel implementation. This allows us to observe how the worker count impacts speedup at different strengths of interaction. Optimization parameters remained consistent across all experiments.

4. RESULTS AND DISCUSSION

Results are organized according to the three optimization stages presented in Section 3: baseline execution, algorithmic acceleration, and acceleration coupled with parallelism. Since Phase 1 and Phase 2 have different underlying structures, their runtime characteristics are discussed independently throughout.

4.1 Baseline and Parallel Execution (No Acceleration)

The baseline formulation of the algorithm is executed sequentially as well as in parallel without utilizing differential evaluation, tensor-based coverage representation, or any other acceleration techniques. Runtime results for this stage can be found in **Table 1.** The first thing to note is that Phase 2 incurs a higher cost than Phase 1 for all values of "k" tested. This tells us that redundant coverage evaluation is the bottleneck of the unaccelerated pipeline. Running in parallel provides a speedup for both phases, though the improvement is greater for Phase 2.

4.2 Algorithmic Acceleration (No Parallelism)

Runtime reduction via algorithmic restructuring alone is investigated during stage 2. Execution times are summarized in **Table 2.** As expected, acceleration does little to reduce Phase 1 runtime, which is dominated by incremental construction and ordered commit. There is, however, a significant reduction observed for Phase 2 in all tested instances of 'k', highlighting repeated coverage evaluation as the stage most impacted by differential evaluation and fixed-size indexed coverage representation.

4.3 Acceleration + Parallelism

Parallelization (in addition to algorithmic acceleration) is also taken into account at stage3. Runtime results are displayed in **Table 3.** Phase 1 does not scale significantly with parallelism in addition to acceleration because of the previously discussed sequential dependencies. Phase 2, when parallelized, exhibits further decreased runtime. Larger speedups can be seen with increased values of k because the candidate evaluation can be further parallelized. For ease of visual comparison between configurations, **Fig. 5** displays the runtime profile of Phase 2 with baseline overlaid with accelerated–parallel runtime. Observe how the difference between these two modes of execution grows with increasing values of k. Phase 2 was selected to be highlighted because it sees the most advantageous and practical improvement from using the proposed acceleration methodology. When compared to Phase 1, Phase 2 has a much higher baseline runtime. So the effects of differential candidate evaluation and parallelization of candidate processing are more apparent.

Table 1. Execution Time for Baseline Sequential and Parallel Execution Without Acceleration

K	Phase 1 Baseline	Phase 1 Parallel	Phase 2 Baseline	Phase 2 Parallel
6	8.445	2.743	84.320	35.110
7	11.537	3.028	124.521	53.603
8	15.458	3.956	183.713	68.348
9	19.889	5.130	271.311	94.489
10	23.604	6.163	337.596	104.686
11	30.14	7.871	571.158	178.238
12	34.463	10.376	1160.601	285.84
13	41.054	13.544	1284.711	422.866
14	54.729	18.886	2082.412	720.156
15	52.685	27.729	2569.438	970.297
16	59.868	32.686	4094.981	1648.495
17	68.226	42.346	5817.820	2544.260
18	76.821	46.233	7820.657	3340.231
19	85.536	50.376	10150.43	4410.221
20	94.773	55.631	13100	5800.467

Table 2. Execution Time for Accelerated Execution

K	Phase 1 Accelerated	Phase 2 Accelerated
6	8.099	21.551
7	10.945	66.391
8	15.162	96.478
9	20.806	120.158
10	25.436	125.837
11	32.603	141.369
12	36.487	146.616
13	50.336	630.618
14	50.216	697.944
15	55.664	794.130
16	62.529	643.221
17	68.270	626.587
18	75.031	823.281
19	82.404	1010.546
20	89.238	1180.114

Table 3. Execution Time for Acceleration + Parallelism

K	Phase 1 Acc + Par	Phase 2 Acc + Par
6	2.943	0.768
7	3.028	1.544
8	4.849	4.889
9	12.130	6.034
10	14.803	7.864
11	16.478	9.922
12	23.852	11.853
13	21.624	44.571
14	25.445	61.264
15	29.540	83.344
16	32.262	89.547
17	19.947	97.750
18	21.444	108.101
19	23.186	129.400
20	25.121	154.801

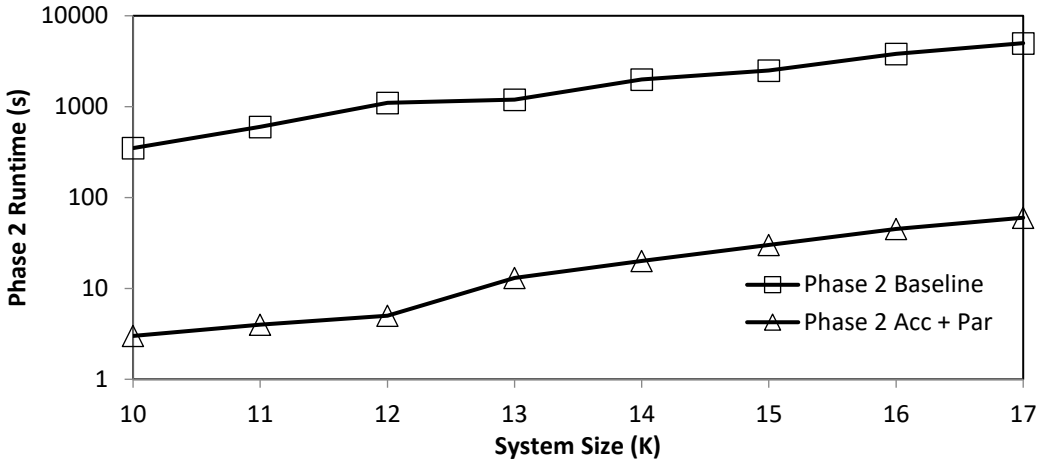

Figure 5. Baseline vs accelerated-parallel phase 2 runtime comparison (log scale)

4.4 Quantitative Comparison

Table 4 summarizes relative speedup achieved at each stage. Speedup was calculated by the formula below:

Speedup = Time_slow / Time_fast

As can be seen from the table Phase 1 experiences very little speedup since it is largely sequential. There is much greater speedup seen in Phase 2.
Runtime improvements are even better when parallelism is taken into account, since Phase 2, in particular benefits, seeing speedups greater than 100× for many instances. Phase 2 runtime is dominated by repeatedly evaluating candidates. Framework overhead is reduced by localizing coverage assessment and pruning redundant recomputation. Indexed representation and parallel execution eliminate it. The takeaway from these results is that performance is dominated by the fact that Phase 1 and Phase 2 have very different structures.

Table 4. Summary of Speedups Across Baseline, Accelerated, and Accelerated-Parallel Execution Runs

K	Acc vs Baseline (×)		Acc + Par vs Acc (×)		Acc +Par vs Baseline (×)	
	P1	P2	P1	P2	P1	P2
6	1.043	3.91	2.752	28.061	2.869	109.79
7	1.054	1.88	3.615	42.999	3.810	80.64
8	1.02	1.90	3.127	19.734	3.188	37.58
9	0.956	2.26	1.715	19.913	1.640	44.96
10	0.928	2.68	1.718	16.002	1.595	42.94
11	0.925	4.04	1.979	14.248	1.829	57.57
12	0.945	7.92	1.530	12.370	1.445	97.93
13	0.816	2.04	2.328	14.149	1.899	28.82
14	1.09	2.98	1.974	11.392	2.152	34.00
15	0.947	3.24	1.884	9.528	1.783	30.83
16	0.957	6.37	1.938	7.183	1.855	45.73
17	1.00	9.28	3.423	6.410	3.420	59.51
18	1.024	9.50	3.499	7.616	3.582	72.35
19	1.038	10.04	3.554	7.809	3.689	78.45
20	1.062	11.10	3.552	7.623	3.773	84.63

4.5 Scaling Characteristics for Acceleration, Considering 't' and Worker Count

Tables 5 to 7 show the effectiveness of parallel execution scales with strength of interactions and the number of workers. For these tests, workers count scales while t and k are fixed per table.

- When t=3 (**Table 5**), execution times for both phases are quite small. Speedup between baseline and accelerated-parallel execution does not differ by much. Here there are simply not enough calculations to amortize the overhead required for parallel execution.
- When t=4 (**Table 6**), we enter a transition regime. There are now more interactions evaluated per row, and Phase 2 starts to see massive batches of independent row-wise evaluations. For these reasons, we begin to see more benefits to parallel execution, though Phase 1 is only partially scalable due to its inherent sequential nature.
- When t=5 (**Table 7),** there are enough floating-point operations to more effectively leverage parallel execution. In this regime, the expense of performing evaluation far outweighs parallel overhead.

In summary, these experiments highlight that accelerator usage is application dependent. For trivially small amounts of work, overhead will dominate. However, as interaction strength rises, parallel execution shows better scaling. This observation shows us that scalability doesn't just depend on adding processors. Parallel execution has limited advantage when the interaction strength is low, but as the amount of work increases the advantages of parallel execution grow. Phase 1 is limited by sequential dependencies but phase 2 scales much better because each evaluation is independent. Parallel execution only plays a factor when there is enough work to take advantage of it.

Table 5. Execution Time and Speedup for Baseline and Accelerated-Parallel Execution runs at t = 3, k = 25

Run	N JOBS	P1 Seq (s)	P1 Par (s)	P1 Speedup	P2 Seq (s)	P2 Par (s)	P2 Speedup
R1	**2**	0.05	0.4	0.12	2.8	2.4	1.16
R2	**4**	0.02	0.02	1	2.6	2.5	1.04
R3	**6**	0.01	0.014	0.71	2.5	2.4	1.04
R4	**8**	0.07	0.04	1.75	2.8	1.7	1.65

Table 6. Execution Time and Speedup for Baseline and Accelerated-Parallel Execution Runs at t =4, k = 12

Run	N JOBS	P1 Seq (s)	P1 Par (s)	P1 Speedup	P2 Seq (s)	P2 Par (s)	P2 Speedup
R1	**2**	35.9	22.3	1.6	38.3	1.7	22.3
R2	**4**	35.9	16.5	2.1	38.3	2.3	16.3
R3	**6**	35.9	13.4	2.6	38.3	2.8	13.3
R4	**8**	35.9	12.1	2.9	38.3	3.3	11.4

Table 7. Execution Time and Speedup for Baseline and Accelerated-Parallel Execution runs at t = 5, k = 10

Run	N JOBS	P1 Seq (s)	P1 Par (s)	P1 Speedup	P2 Seq (s)	P2 Par (s)	P2 Speedup
R1	**2**	274.2	163.6	1.67	33.3	1.4	23.6
R2	**4**	274.2	110.7	2.47	33.3	1.7	19.4
R3	**6**	274.2	88.5	3.10	33.3	1.9	16.8
R4	**8**	274.2	76.7	3.57	33.3	2.2	14.7

For reference, the average speedup across all reported worker configurations was 0.89× and 1.22× for Phase 1 and 2, respectively, at t = 3, k = 25. At t = 4, k = 12. These numbers became 2.30× and 15.83×. Finally, at t = 5, k = 10, We saw average speedups of 2.70× and 18.63× for Phase 1 and 2, respectively.

5. CONCLUSIONS

The research detailed how Sequence Covering Array generation and optimization were accelerated through differential evaluation, indexed coverage representation, and process-level parallelism. Instead of devising a new construction heuristic, computation was rearranged to increase throughput with the same guarantee of ordered interaction coverage. Performance was found empirically to scale almost exclusively with removal of redundant coverage evaluation and the ability to efficiently process independent evaluations concurrently. Significant speedups were observed from restructuring, confirming its vital role in preparing for parallel execution. Indexed coverage representation also allowed Sequence Covering Arrays to gain utility by decreasing generation and optimization expense, making them viable for larger systems. Still, this advancement is not without its drawbacks. Indexed coverage representation required additional memory, which scales with the size of the interaction space. Indexed coverage representation motivated future work investigating alternative coverage representations and implementations.

NOMENCLATURE

Symbol	Description	Symbol	Description
f	Sequential fraction in Amdahl's law	N_JOBS	Number of parallel workers
p	Number of processors	chk	Coverage matrix (indexed representation)
S(P)	Parallel speedup	Tests	Generated test suite
t	Interaction strength	Time_fast	Execution time of faster configuration(s)
K	Number of events (system size)	Time_slow	Execution time of slower configuration(s)

Credit Authorship Contribution Statement

Zainab A. Najm: Conceptualization, writing of original draft, Implementation, data analysis, Mohammed Issam Younis: Editing and supervision.

Declaration of Competing Interest

The authors declare no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

REFERENCES

Ahmed, B.S., Gambardella, L.M., Afzal, W. and Zamli, K.Z., 2017. Handling constraints in combinatorial interaction testing in the presence of multi-objective particle swarm optimization and multithreading. *Information and Software Technology*, 86, pp. 20–36. https://doi.org/10.1016/j.infsof.2017.02.004

Amdahl, G.M., 1967. Validity of the single processor approach to achieving large-scale computing capabilities. *In: Proceedings of the April 18–20, 1967 Spring Joint Computer Conference* (AFIPS '67 Spring), Atlantic City, NJ, USA, pp. 483–485. https://doi.org/10.1145/1465482.1465560

Bohm, S., Krieter, S., Heb, T., Thum, T. and Lochau, M., 2024. Incremental identification of t-wise feature interactions. In*: Proceedings of the 18th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS '24)*, pp. 27–36. https://doi.org/10.1145/3634713.3634715

Bombarda, A., Gargantini, A. and Calvagna, A., 2023. Multi-thread combinatorial test generation with SMT solvers. In: *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing (SAC '23)*, Tallinn, Estonia, pp. 1698–1705. https://doi.org/10.1145/3555776.3577703

Castro, O., Bruneau, P., Sottet, J.-S. and Torregrossa, D., 2024. Landscape of high-performance Python to develop data science and machine learning applications. *ACM Computing Surveys*, 56(3), Article 65, pp. 1–30. https://doi.org/10.1145/3617588

Chee, Y.M., Colbourn, C.J., Horsley, D. and Zhou, J., 2013. Sequence covering arrays. *SIAM Journal on Discrete Mathematics*, 27(4), pp. 1844–1861. https://doi.org/10.1137/120894099

Demiroz, M. and Yılmaz, C., 2016. Using simulated annealing for computing cost-aware covering arrays. *Applied Soft Computing*, 49, pp. 1129–1144. https://doi.org/10.1016/j.asoc.2016.08.022

Fadhil, H.M., Abdullah, M.N. and Younis, M.I., 2023. Innovations in t-way test creation based on a hybrid hill climbing-greedy algorithm. *IAES International Journal of Artificial Intelligence*, 12(2), pp. 794–805. https://doi.org/10.11591/ijai.v12.i2.pp794-805

Fan, Y., Wan, C., Fu, C., Han, L. and Xu, H., 2023. VDoTR: Vulnerability detection based on tensor representation of comprehensive code graphs. *Computers & Security*, 130, P. 103247. https://doi.org/10.1016/j.cose.2023.103247

Guo, X., Song, X., Zhou, J., Wang, F., Tang, K. and Wang, Z., 2023. A memetic algorithm for high-strength covering array generation. *IET Software*, 17(4), pp. 538–553. https://doi.org/10.1049/sfw2.12138

Gustafson, J.L., 1988. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5), pp. 532–533. https://doi.org/10.1145/42411.42415

Islam, M., Khan, F., Alam, S. and Hasan, M., 2023. Artificial intelligence in software testing: A systematic review. In: *Proceedings of the IEEE Region 10 Conference (TENCON 2023).* https://doi.org/10.1109/TENCON58879.2023.10322349

Izquierdo-Marquez, I., Torres-Jimenez, J., Acevedo, B. and Avila-George, H., 2018. A greedy-metaheuristic 3-stage approach to construct covering arrays. *Information Sciences*, 460–461, pp. 172–189. https://doi.org/10.1016/j.ins.2018.05.047

Kuhn, D.R., Higdon, J.M., Lawrence, J.F., Kacker, R.N. and Lei, Y., 2012. Combinatorial methods for event sequence testing. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST),* pp. 601–609. https://doi.org/10.1109/ICST.2012.147

Lara-Alvarez, M. and Avila-George, H., 2015. A new algorithm for post-processing covering arrays. *International Journal of Advanced Computer Science and Applications*, 6(12), pp. 250–254. https://doi.org/10.14569/IJACSA.2015.061234

Lee, S. and Kim, S., 2020. Parallel simulated annealing with a greedy algorithm for Bayesian network structure learning. *IEEE Transactions on Knowledge and Data Engineering*, 32(6), pp. 1157–1166. https://doi.org/10.1109/TKDE.2019.2899096.

Li, F., Zhou, J., Li, Y., Hao, D. and Zhang, L., 2022. AGA: An accelerated greedy additional algorithm for test case prioritization. *IEEE Transactions on Software Engineering*, 48(12), pp. 5102–5119. https://doi.org/10.1109/TSE.2021.3137929

Mercan, H., 2021. Unified Combinatorial Interaction Testing (U-CIT). PhD thesis, Sabancı University, Istanbul, Turkey.

Nasser, A., Zamli, K.Z., Alsewari, A.R. and Ahmed, B., 2018. An elitist-flower pollination-based strategy for constructing sequence and sequence-less t-way test suite, *International Journal of Bio-Inspired Computation*, 12(2), pp. 115–127. https://doi.org/10.1504/IJBIC.2018.094223

Nayeri, M., Colbourn, C.J. and Konjevod, G., 2013. Randomized post-optimization of covering arrays. *European Journal of Combinatorics*, 34(1), pp. 91–103. https://doi.org/10.1016/j.ejc.2012.07.017

Nie, C. and Leung, H., 2011. A survey of combinatorial testing. *ACM Computing Surveys*, 43(2), pp. 11:1–11:29. https://doi.org/10.1145/1883612.1883618

Petke, J., Cohen, M., Harman, M. and Yoo, S., 2015. Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection. *IEEE Transactions on Software Engineering*, 41(9), pp. 901–924. https://doi.org/10.1109/TSE.2015.2421279

Rahman, M.M., Sultana, D., Khatun, S. and Mat Jusof, M.F., 2020. T-way strategy for sequence input interaction test case generation adopting fish swarm algorithm. In: Nasir, A.N.K. et al. (eds.), *InECCE2019*. Lecture Notes in Electrical Engineering, vol. 632. Singapore: Springer, pp. 87–99. https://doi.org/10.1007/978-981-15-2317-5_9

Sabharwal, S., Bansal, P. and Mittal, N., 2016. Construction of t-way covering arrays using genetic algorithm. *International Journal of System Assurance Engineering and Management*, 8, pp. 264–274. https://doi.org/10.1007/s13198-016-0430-6

Salehi, H., Gorodetsky, A., Solhmirzaei, R. and Jiao, P., 2023. High-dimensional data analytics in civil engineering: A review on matrix and tensor decomposition. *Engineering Applications of Artificial Intelligence*, 125, 106659. https://doi.org/10.1016/j.engappai.2023.106659

Sheng, Y., Sun, C., Jiang, S. and Wei, C., 2018. Extended covering arrays for sequence coverage, *Symmetry*, 10(5), P. 146. https://doi.org/10.3390/sym10050146

Torbunova, A., Strandberg, P.E. and Porres, I., 2024. Dynamic test case prioritization in industrial test result datasets. In: *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*, Lisbon, Portugal, pp. 154–158. https://doi.org/10.1145/3644032.3644452

Torres-Jimenez, J. and Rodriguez-Tello, E., 2012. New bounds for binary covering arrays using simulated annealing. *Information Sciences*, 185(1), pp. 137–152. https://doi.org/10.1016/j.ins.2011.09.020

Witharana, H., Jayasena, A. and Mishra, P., 2024. Incremental concolic testing of register-transfer level designs. *ACM Transactions on Design Automation of Electronic Systems,* 29(3), pp.1–23. https://doi.org/10.1145/3655621

Yang, J., Fu, C., Deng, F., Wen, M., Guo, X. and Wan, C., 2023. Toward interpretable graph tensor convolution neural network for code semantics embedding. *ACM Transactions on Software Engineering and Methodology*, 32(5), Article 115, pp.115:1–115:40. https://doi.org/10.1145/3582574

Younis, M.I., 2020. DEO: A dynamic event order strategy for t-way sequence covering array test data generation. *Baghdad Science Journal*, 17(2), pp. 575–582. https://doi.org/10.21123/bsj.2020.17.2.0575

Zabil, M.H.M., Zamli, K.Z. and Lim, K.C., 2018. Evaluating Bees algorithm for sequence-based t-way testing test data generation, *Indian Journal of Science and Technology*, 11(4), pp. 1–20. https://doi.org/10.17485/ijst/2018/v11i4/121086

Zamli, K. and Kader, M., 2021. Sequence t-way test generation using the Barnacles mating optimizer algorithm. In: Proceedings of the 10th *International Conference on Software and Computer Applications (ICSCA 2021)*, Kuala Lumpur, Malaysia, pp. 88–93. https://doi.org/10.1145/3457784.3457797

Zeng, M.-F., 2016. Generating covering arrays using ant colony optimization: Exploration and mining. *Journal of Software*, 27(4), pp. 855–878. https://doi.org/10.13328/j.cnki.jos.004974

تسريع توليد وتحسين مصفوفات التغطية المتسلسلة باستخدام التقييم التفاضلي والتوازي على مستوى العمليات

زينب احمد نجم، محمد عصام يونس*

قسم هندسة الحاسبات، كلية الهندسة، جامعة بغداد، بغداد، العراق

الخلاصة

تُعد مصفوفات التغطية المتسلسلة امتدادًا لأساليب الاختبار التوافقي، وقد طُورت خصيصًا لاكتشاف الأخطاء المرتبطة بترتيب الأحداث. إذ تضمن هذه المصفوفات احتواء جميع التفاعلات المرتبة لعدد محدد من العناصر على شكل تتابعات جزئية ضمن بنية الاختبار. إلا أن عملية توليد هذه المصفوفات وتحسينها تُعد مكلفة من الناحية الحسابية، مما حدّ من استخدامها في التطبيقات العملية .يقدم هذا البحث إطارًا جديدًا لتسريع عمليات بناء مصفوفات التغطية المتسلسلة. وتتمثل الفكرة الأساسية في إعادة تنظيم الخوارزميات المستخدمة ضمن الإجراءات الحالية بهدف تحقيق تحسين في الأداء دون التأثير على خصائصها الأساسية. يعتمد إطار التسريع على استخدام التقييم التفاضلي لتقليل العمليات الحسابية المتكررة، كما يوظف تراكيب بيانات ذات حجم ثابت لتبسيط إدارة التغطية، ويستفيد من التوازي على مستوى العمليات لتوزيع عمليات التقييم المستقلة على عدة أنوية من وحدة المعالجة .وقد أظهرت النتائج تحسنًا واضحًا في الأداء عبر جميع حالات الاختبار. ففي المرحلة الأولى، تحقق تحسن محدود نتيجة التسريع الخوارزمي البحت، إلا أنها أظهرت تسارعًا يتراوح بين ٢،٥٨ و٣،٨٠ مرة عند تطبيق التسريع والتوازي معًا. أما المرحلة الثانية، فقد شهدت تحسنًا ملحوظًا نتيجة إعادة تنظيم الخوارزمية، حيث تراوح التسارع بين ٢،٢١ و١٠،٤١ مرة .وعند دمج التسريع مع التوازي في هذه المرحلة، بلغ التسارع بين ٨٠،٦٥ و٢٥٩،٧٠ مرة عبر جميع الإعدادات، مع تسجيل أفضل حالة تسارع بلغت ٢٥٩،٧٠ مرة .وتُظهر هذه النتائج أن بناء مصفوفات التغطية المتسلسلة القابلة للتوسع أصبح ممكنًا من خلال إعادة التنظيم الداعمة للتقييم المتوازي، مع الحفاظ على دقة التغطية.

الكلمات المفتاحية: مصفوفات التغطية المتسلسلة، الاختبار التوافقي، التسريع الخوارزمي، التقييم التفاضلي، التمثيل المعتمد على المصفوفات متعددة الأبعاد، التوازي على مستوى العمليات.