Improved Path Clustering for Adaptive Path-Delay Testing

Tuck-Boon Chan†, Andrew B. Kahng‡
UC San Diego ECE†‡ and CSE‡ Departments, La Jolla, CA 92093
E-mails: tbchan@ucsd.edu†, abk@cs.ucsd.edu‡

Abstract— Adaptive path-delay testing is a testing methodology that reduces redundant test patterns based on the measured process condition of a die under test (DUT). To improve testing efficiency, process conditions are clustered into a limited number of clusters, each of which has a corresponding set of test patterns. The test pattern set of a cluster must include all potential timing-critical paths of all process conditions in the cluster. Hence, high-quality clustering is needed to minimize redundant test paths. In this paper, we propose a new clustering heuristic to minimize the expected number of redundant test paths in adaptive path-delay testing. Our experimental results on randomly generated testcases show that the proposed clustering heuristic can reduce the expected number of test paths by up to 40% compared to the previous Greedy clustering algorithm of Uezono et al. [5]. To address unique attributes of an industrial testcase obtained from the authors of [5], we integrate the dynamic-programming restricted-partitioning technique of [1], which improves the expected number of test paths by up to 5% compared to the Greedy algorithm.

Keywords— adaptive testing, clustering, partitioning

I. Introduction

In conventional VLSI path-delay testing, a set of test patterns is applied to every die under test (DUT). This is inefficient because the test patterns are extracted from different process corners, while a given DUT only requires the test patterns corresponding to its specific (i.e., as-manufactured) process condition. To improve path-delay testing efficiency, Shintani et al. in [4] proposed an adaptive testing methodology as illustrated in Figure 1.

In the adaptive testing approach, a set of test patterns is prepared for each process condition. During testing, a testing machine applies a set of test patterns (i.e., a test program) according to the identified process condition of a given DUT. Although adaptivity can reduce redundant test patterns, it requires large memory space on a testing machine to store test patterns for each process condition. In practice, the memory space on a testing machine is limited, hence process conditions – with their corresponding test patterns – must be clustered. Clustering of process conditions saves memory on the testing machine but typically results in redundant test patterns for any given process condition in a cluster.

Figure 2 illustrates the clustering problem with three sets of critical paths S_1, S_2 and S_3 that respectively correspond to process conditions V_1, V_2 and V_3. In this example, the number of critical paths in S_1, S_2 and S_3 is 15, 30 and 25, respectively (some of the paths belong to multiple process conditions). Given an upper limit of 2 test-pattern sets, i.e., clusters, clustering solution A merges S_1 and S_2 into a cluster C_1. As a result, whenever either process condition V_1 or V_2 occurs, the test patterns of

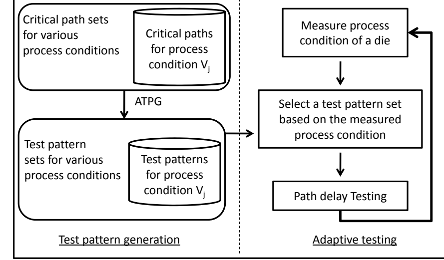


Fig. 1. Adaptive path-delay testing flow proposed in [4]. The test program (set of test patterns) applied to a given die under test (DUT) is selected based on the process condition of the die.

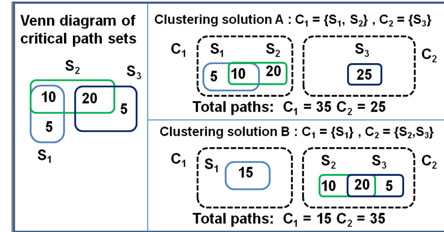


Fig. 2. An adaptive testing clustering problem with three sets of critical paths. Total paths in a cluster is determined by the clustering solution.

critical paths in $S_1 \cup S_2$ will be applied. Since 20 critical paths in S_2 do not belong to process condition V_1, test patterns of these paths are redundant for a DUT with process condition V_1. In contrast to clustering solution A, solution B does not merge S_1 with any other critical path set. Therefore, there is no redundant test pattern for a DUT with process condition V_1. Clearly, the efficiency of the adaptive testing – and hence the cost of test – is affected by the quality of the process condition clustering.[1] In this paper, we propose a new clustering algorithm that significantly improves efficiency of the adaptive test methodology, compared with a recent greedy approach of Uezono et al. [5].

II. Problem Formulation

In adaptive path-delay testing, we extract a set of critical paths for each process condition. Formally:

$$\begin{aligned} V_j &= \text{the } j^{th} \text{ process condition in a segmented} \\ &\quad \text{process parameter space}, j = 1, \ldots, M \\ P &= \{P_1, \ldots, P_N\} = \text{set of all critical paths} \\ S_j &\subseteq P = \text{set of critical paths for process condition } V_j \\ S &= \{S_1, S_2, \ldots, S_M\} = \text{set of all } S_j \\ Q_j &= \text{occurrence probability of process condition } V_j. \end{aligned} \tag{1}$$

[1]Note that we are using the number of critical paths tested as a proxy for the complexity of testing. We understand that for each critical path, some number of tests and test patterns will be synthesized, that test compression methods will be applied, etc. The clustering methods that we study can be applied as easily to test patterns as to critical paths, but we follow previous work and testcase data in focusing on critical paths.

Given process conditions $V = \{V_1, V_2, ..., V_M\}$, their respective occurrence probabilities $\{Q_1, Q_2, ..., Q_M\}$, and an upper limit k on the number of clusters, we seek to partition S into k disjoint clusters[2] of paths $C_1, C_2, ..., C_k$, with each $C_i \subseteq P$, to minimize redundant test patterns. In other words, each cluster C_i is a nonempty subset of S, and each process condition's set of critical paths S_j belongs to exactly one cluster. In the adaptive testing methodology [4], when the DUT has process condition V_j, all test patterns in the cluster C_i that contains S_j will be applied. Thus, test patterns $\in C_i \backslash S_j$ are redundant. We assume henceforth that test patterns are linearly proportional to critical paths (i.e., as noted earlier, the number of critical paths is a proxy for the number of test patterns and testing costs). The total test cost $f(C)$ of a clustering result C is defined as

$$f(C) = \sum_{i=1}^{k} (\sum_{S_j \in C_i} Q_j) \times (\sum_{P_h \in C_i} |P_h|) \tag{2}$$

where $|P_h| = 1$ when we assume that the number of test patterns is proportional to the number of paths, and every path has equal test cost.

III. Previous Work

Uezono et al. in [5] propose a *Greedy* algorithm for the clustering problem in Section II. The Greedy algorithm first generates M clusters of critical paths by defining $C_j = S_j$, then iteratively merges two clusters at a time until the total number of clusters is equal to k. In each iteration, the algorithm merges two clusters C_i and C_j with minimal *distance* ($d_{i,j}$) defined as

$$d_{i,j} = Q_i \times (\sum_{P_j \in C_j} |P_j| - \sum_{P_c \in C_i \cap C_j} |P_c|) + Q_j \times (\sum_{P_i \in C_i} |P_i| - \sum_{P_c \in C_i \cap C_j} |P_c|) \tag{3}$$

In other words, $d_{i,j}$ is the incremental test cost of merging C_i and C_j. A motivating observation for our present work is that although the Greedy algorithm merges the clusters with minimal $d_{i,j}$ in each merging operation, this can yield solutions with considerable suboptimality. (And, since the cost measure is essentially the expected cost of manufacturing test, there is a strong motivation to devise an improved heuristic.) We have devised a family of adversarial instances that forces the performance ratio of Greedy for a k-cluster solution to approach 2 as k grows large. Our construction is described as follows.

- There are $2k$ process conditions. We refer to $S_1, S_2, \ldots, S_k$ as *Type-A* process conditions, and $S_{k+1}, S_{k+2}, \ldots, S_{2k}$ as *Type-B* process conditions.
- The Greedy k-cluster solution, denoted C^G, will be forced to have structure $C_1^G = \{S_1, S_{k+1}, S_{k+2}, ..., S_{2k}\}$, and $C_i^G = \{S_i\}$, $i = 2, 3, ..., k$.
- The optimal k-cluster solution, denoted C^{opt}, will be forced to have structure $C_i^{opt} = \{S_i, S_{k+i}\}$, $i = 1, 2, ..., k$.

We then establish a number of constraints on how the Greedy algorithm arrives at its solution C^G; this enables us to set up and

[2]Each cluster contains all test patterns of all process conditions in the cluster. Hence, assigning a process condition to two or more clusters – i.e., a non-disjoint partition – will incur redundant test patterns and need not be considered.

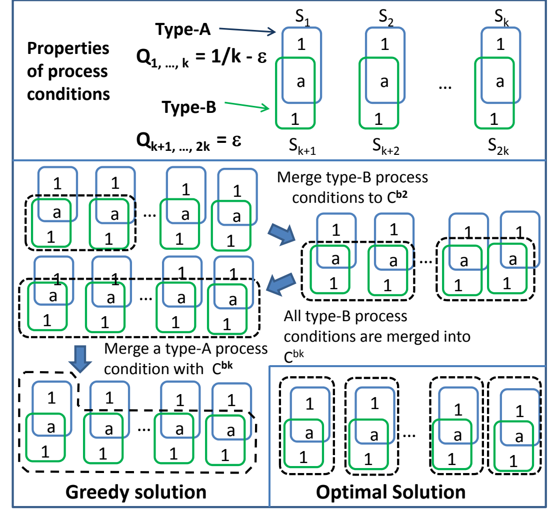


Fig. 3. Parameterized adversarial instance for the Greedy algorithm, with imbalance Q_j.

solve a mathematical program that maximizes the performance ratio of Greedy subject to these constraints.

- The greedy solution will be forced to merge all *type-B* process conditions first (to achieve $k+1$ clusters), before merging one *type-A* process condition to obtain k clusters.
- More specifically, at the first $k/2$ clustering steps, each *type-B* process condition will be forced to merge with another *type-B* process condition. This will lead to $k/2$ clusters, each of which contains two *type-B* process conditions. We denote each such merged cluster as C^{b2}, where the value "2" indicates that there are two *type-B* process conditions in the cluster.
- After that, the Greedy heuristic will continue to be forced to merge the resulting C^{bn} clusters until $n = k$.
- Finally, C^{bk} will be forced to merge with a *type-A* process condition to form $C_1^G = \{S_1, S_{k+1}, S_{k+2}, ..., S_{2k}\}$.

Each of the merging steps mentioned above forces our adversarial instance to satisfy the constraint that the two merging clusters have the smallest distance (defined in Equation (3)) among all possible cluster pairs. For instance, in the first merging step, each process condition is a cluster by itself. The cost of merging any two *type-B* process condition must be less than the cost of merging a *type-A* process condition with a *type-B* process condition. This constraint is established by the first inequality in Equation (4) with $n = 1$. Similarly, when merging C^{bn} clusters, the distance between any two C^{bn} clusters must be (i) smaller than the distance between a C^{bn} and a cluster with a *type-A* process condition (C_2^G), and (ii) smaller than the distance between two *type-A* process conditions (C_2^G). Thus, the following inequality constraints must hold[3] for $n = 1, 2, \ldots, k/2$:

$$\begin{aligned} &d(C^{bn}, C^{bn}) \le d(C_2^G, C^{bn}) \\ &d(C^{bn}, C^{bn}) \le d(C_2^G, C_2^G) \\ &d(C^{bn}, C^{bn}) = 2n^2\varepsilon(a+1), n = 1, 2, ..., k/2 \\ &d(C_2^G, C^{bn}) = n\varepsilon + (\frac{1}{z} - \varepsilon)(n(a+1) - a), n = 1, 2, ..., k/2 \end{aligned} \tag{4}$$

[3]It is clear that $d(C^{bn}, C^{bn}) < d(C^{bn/2}, C^{bn})$ for all n, so we do not include this inequality as a constraint when constructing the adversarial instance.

where $d(C^{bn},C^{bn})$ is the distance (cost) of merging any two C^{bn} clusters with n number of *type-B* process conditions, $d(C_2^G,C^{bn})$ is the distance of merging a *type-A* process condition with any C^{bn} in the first $k-1$ steps of the Greedy algorithm, and $d(C_2^G,C_2^G)$ is the distance of merging two *type-A* process conditions. Equation (4) can be simplified to three dominant constraints

$$
\begin{aligned}
F(n) &= d(C^{bn},C^{bn}) - d(C_2^G,C^{bn}) \\
&= 2n^2\varepsilon(a+1) - [n\varepsilon + (\frac{1}{k} - \varepsilon)(n(a+1)-a)] \\
&\leq 0 \\
F(1) &= 2\varepsilon(a+1) - \frac{1}{k} \\
\varepsilon &\leq \frac{4}{k^3}
\end{aligned} \tag{5}
$$

From this, we obtain $2\varepsilon(a+1) \leq 1/k$; then, substituting $2\varepsilon(a+1)$ as a lower bound for $\frac{1}{k}$ in the expression for $F(n)$,

$$
\begin{aligned}
&2n^2\varepsilon(a+1) - [n\varepsilon + (2\varepsilon(a+1)-\varepsilon)(n(a+1)-a)] \leq 0 \\
&\implies 2n^2(a+1) - [n + (2(a+1)-1)(n(a+1)-a)] \leq 0 \\
&\implies 2n^2(a+1) - n(2a^2+a) + 2a^2 - a < 0 \\
&\implies 2n^2(a+1) - 2(n-1)a^2 - (n+1)a < 0
\end{aligned} \tag{6}
$$

Applying an optimization tool (CPLEX from ILOG) to maximize the performance ratio subject to the inequalities (5) and (6), we obtain

$$
\begin{aligned}
&\text{performance ratio} = \frac{C^G}{C^{opt}} \\
&\frac{(k-1)(1/k-\varepsilon)(a+1) + (1/k+(k-1)\varepsilon)(k(a+1)+1)}{a+2} \\
&\approx (k-1)(1/k-\varepsilon) + (1+k(k-1)\varepsilon) \\
&= 2 - 2k\varepsilon - 1/k + \varepsilon + k^2\varepsilon
\end{aligned} \tag{7}
$$

That is, the performance ratio ≈ 2.0 as $\varepsilon \to 0$ and k grows large. For example, when $a = 100000, \varepsilon = 10^{-8}, k = 300$, and $n = 1,2,\ldots,150$, inequalities (5) and (6) are satisfied with performance ratio = 1.997.

IV. Our Proposed Method

Inspired by the Fiduccia-Mattheyses (*FM*) algorithm [2], we formulate the clustering problem as a hypergraph (network) partitioning problem and solve it using an iterative hill-climbing approach. Figure 4 depicts a hypergraph that represents an instance of the clustering problem. The clustering problem has four process conditions and three critical paths ($S_1 = \{P_1\}$, $S_2 = \{P_2, P_3\}$, $S_3 = \{P_3\}$ and $S_4 = \{P_1, P_3\}$). In the figure, each vertex represents a process condition and each hyperedge represents a critical path. If a critical path belongs to only one process condition, it is represented as a dangling edge for the process condition. After representing the clustering problem as a hypergraph, we may formulate a new partitioning problem that is equivalent to the original problem. I.e., given a hypergraph with N hyperedges and M vertices, we want to partition the vertices into k clusters (subgraphs) with minimal cost as defined in Equation (2).

To solve the partitioning problem, we recursively bipartition a hypergraph into two smaller subgraphs[4] recursively until there are k subgraphs $(C_1, C_2, \ldots, C_k)$. After obtaining k subgraphs, we check whether there is any improvement by *merging and re-splitting* any two subgraphs C_i and C_j, where $i \neq j$. During the merging and re-splitting operation, we update the partitioning solution only if there is any improvement after the operation. For a solution with k subgraphs, we define a *refinement step* as an operation that performs all feasible $k(k-1)/2$ merging and re-splitting operations. Since the refinement step has long runtime, we repeat the refinement step either until there is no improvement or until a *maximum refinement iterations* limit has been reached. We also apply the refinement step only when the number of clusters does not exceed a user-defined *refinement upper bound*. The refinement step is important for two reasons.

- The bipartitioning operation must search over a combinatorial number of candidate solutions, with only unclear criteria relative to a globally good outcome. In top-down recursive partitioning, each bipartitioning optimization is susceptible to missing solutions that lead to the best possible solution.
- Even if the bipartitioning operation chooses the best candidate for splitting, it only minimizes the cost of splitting that particular subgraph. The refinement step explores other possible solutions which can improve the overall cost.

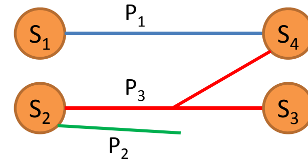


Fig. 4. Four process conditions (vertices) S_1, S_2, S_3 and S_4 are connected by hyperedges defined by paths that are timing-critical in respective subsets of the process conditions.

A. Overview of Our Algorithm

Given subgraphs (clusters) C_A and C_B, we move one vertex (process condition) at a time from one subgraph to another in an attempt to minimize total cost as defined by Equation (2). At each move, we move the vertex that brings the largest cost reduction. To prevent the moving process from going into an infinite loop, each vertex is "locked" in its new subgraph after the move. The moving process stops after all vertices have been moved. Then, we update the vertices in both subgraphs according to the best solution encountered during the moves, and "unlock" all vertices. A complete cycle of these procedures is defined as a *pass*. To improve the partitioning solution, we perform multiple passes until there is no improvement. Our proposed method resembles the FM algorithm [2] in its neighborhood structure for iterative search, and in its pass-based hill-climbing strategy. It differs from [2] in that the cost calculation is not defined by net "cuts", but rather by the number of nets in each cluster.[5]

[4] Henceforth, use of 'subgraph' for 'sub-hypergraph' is understood when the context is clear.

[5] Also, some efficiencies in, e.g., gain bucketing or gain calculation and update that are available to FM for hypergraph min-cut partitioning are not available in our context. We discuss the runtime complexity implications in the next subsection.

B. Gain Calculation for Moving a Process Condition

Based on Equation (2), we define g_j, the *gain* of moving a process condition j, as follows. We use T ("to") to denote the subgraph that the vertex is moving towards, and F ("from") to denote the subgraph that contains the vertex before the move. It is not hard to see that

$$g_j = |C_F|Q_F + |C_T|Q_T - [(|C_F| + \Delta|C_F|[j])(Q_F - Q_j) + (|C_T| + \Delta|C_T|[j])(Q_T + Q_j)] \quad (8)$$

where $|C_T|$ and $|C_F|$ are the respective total number of paths in the two subgraphs, and $\Delta|C_T|[j]$ and $\Delta|C_F|[j]$ are the respective changes in the number of paths in C_F and C_T if we were to move vertex j from C_F to C_T. Q_T (resp. Q_F) gives the sum of probabilities of all process conditions in C_T (resp. C_F) before a move. Note that the gain of moving a vertex (process condition) is dependent on the sum of the probabilities Q_T and Q_F, as well as on the number of paths in the subgraphs before and after moving. Therefore, moving a vertex will affect the gains of all other vertices in subsequent moves. As a result, all g_j values need to be recomputed in each move, which requires $O(M)$ gain calculations.

C. Maintaining Gains

During the bipartitioning process, we maintain the data in $\Delta|C_T|[j]$ and $\Delta|C_F|[j]$ after each move so that we can efficiently calculate the gains of subsequent moves. Pseudocode for moving a given vertex S_j from C_F to C_T is given in Figure 5, and pseudocode for maintaining gains is given in Figure 6.[6]. In Figure 5, Lines 1-5 update the cost when a vertex S_j is moved by subtracting g_{max} from current cost and the values of Q_F (Q_T). Then in Lines 6-7, we update $|C_T|$ ($|C_F|$) based on the values maintained at $\Delta|C_T|[j]$ ($\Delta|C_F|[j]$). At Line 8, the algorithm moves vertex S_{j_max} and locks it after the move. Finally, Lines 10-12 update the values of g and $\Delta|C_T|[j]$ ($\Delta|C_F|[j]$) using the *update_gain* procedure. Moving a vertex S_{j_max} from a subgraph to another subgraph only affects the $\Delta|C_T|[j]$ ($\Delta|C_F|[j]$) values of vertices that share the same edges connected to S_{j_max}. Therefore, it is sufficient to check each S_j that is connected to P_h when we calculate $\Delta|C_T|[j]$ ($\Delta|C_F|[j]$).

In Figure 6, $x(P_h,T)$ (resp. $x(P_h,F)$) is the number of S_j connected to P_h in subgraph T (resp. F). In Lines 2-7 of Figure 6, the algorithm checks whether $x(P_h,T)$ = 0 or 1, and updates $\Delta|C_T|[j]$ ($\Delta|C_F|[j]$) accordingly. For example, when $x(P_h,T)$ = 0, the cost of moving another vertex with edge P_h into the T subgraph will be reduced by one because P_h will be in the T subgraph right after this move. When $x(P_h,T)$ = 1, $\Delta|C_T|[j]$ of the single vertex will be increased by one because the T subgraph will still contain P_h after the move. Lines 8-9 update the value of $x(P_h,T)$ ($x(P_h,F)$) to "move" the vertex S_j. Lines 10-15 check the value of $x(P_h,F)$ and perform similar steps as in Lines 2-7 to update $\Delta|C_T|[j]$ ($\Delta|C_F|[j]$). Finally, the algorithm calculates the gain for each possible move g_j by using Equation (8). In this gain calculation, updating $\Delta|C_T|[j]$ ($\Delta|C_F|[j]$) takes at most O(N) time, where N is the total number of paths.

[6]When more than one movable vertices have identical gain in a pass, our implementation will break the tie in favor of the vertex with smallest process condition index.

Procedure *move*()

1. g_{max} = the maximum gain among all movable S_j
2. j_max = the index of movable S with maximum gain
3. cost = cost - g_{max}
4. $Q_F = Q_F - Q_[j_max]$
5. $Q_T = Q_T + Q_[j_max]$
6. $|C_F| = |C_F| + \Delta|C_F|[j_max]$
7. $|C_T| = |C_T| + \Delta|C_T|[j_max]$
8. Move process condition S_{j_max} to the other cluster
9. lock S_{j_max}
10. **for** (P_h connected to S_{j_max}) **do**
11. update_gain(cost,h, j_max)
12. **end for**

Fig. 5. Moving a vertex.

Procedure *update_gain*($cost,h,j_max$)

1. **for** (each S_j connected to P_h) **do**
2. **if** ($x(P_h,T) == 0$) **then**
3. $\Delta|C_F|[j]--$
4. **end if**
5. **if** ($x(P_h,T) == 1$) **then**
6. $\Delta|C_T|[j]++$
7. **end if**
8. $x(P_h,T)++$
9. $x(P_h,F)--$
10. **if** ($x(p_h,F) == 0$) **then**
11. $\Delta|C_T|[j]++$
12. **end if**
13. **if** ($x(P_h,F) == 1$) **then**
14. $\Delta|C_F|[j]--$
15. **end if**
16. $g_j = cost - (|C_F| + \Delta|C_F|[j_max])(Q_F - Q_j) + (|C_T| + \Delta|C_T|[j_max])(Q_T + Q_j)$
17. **end for**

Fig. 6. Incremental gain calculation.

V. TESTCASE GENERATION

As described above, we use a hypergraph to represent the problem of clustering process conditions for adaptive test. In the hypergraph, process conditions and critical paths are represented as vertices S_j and P_h, respectively. If a critical path h belongs to a process condition j, we may think of h's hyperedge as "containing" or inducing an edge $e_{h,j}$ between vertices S_j and P_h. Figure 7 shows a process condition S_1 which has two critical paths P_1 and P_2. There is an edge $e_{1,1}$ between P_1 and S_1, and an edge $e_{2,1}$ between P_2 and S_1. In general, if a path h belongs to different k process conditions, its hyperedge will induce k edges in this way.

Given a hypergraph, we want to find the connection ($b_{j,l}$) between S_j and C_k that minimizes testing cost as defined in Equation (2). Note that each process condition has exactly one edge to a cluster. Any additional edge on a process condition will increase overall testing cost.

We represent an instance of the clustering problem with vertices for N critical paths and M process conditions, edges $e_{h,j}$, probability R_h of a path belonging to process conditions, and occurrence probabilities Q_j for the process conditions. To explore different input instances, we generate the edges $e_{h,j}$ using the procedure shown in Figure 8, with the following input parameters:

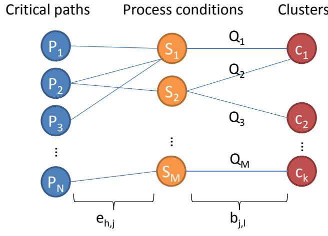


Fig. 7. Graphical depiction of clustering of process conditions for adaptive testing. A process condition S_j needs to test path P_h if there is an edge $e_{h,j}$ between the corresponding vertices. The clustering problem entails finding optimal connections (assignments) from the S_j to the C_k to minimize testing cost.

- N = total number of critical paths
- M = total number of process conditions
- α = probability of $P_1 \in S_j, j = 1,2,...,M$
- β = probability of $P_N \in S_j, j = 1,2,...,M$
- *proc_dist* = {uniform, Gaussian, power-law} distribution of process condition occurrence probabilities

The connections between process conditions and critical paths are affected by the timing slack of each critical path. Moreover, the connections are strongly dependent on the the paths' sensitivities to process variations. Since we do not have the circuit and process information to generate input instances, we generate the connections between process conditions and critical paths using the random graph model $G(n,P)$ in [3]. In the proposed random graph, we assume that every process condition is equally likely to be connected to a path. However, the probability of adding an edge from P_h to any process condition is determined by R_h, which varies linearly from α to β, i.e., $R_h = \alpha - (\alpha - \beta) \times h/N$. Further, we define the average number of edges in the hypergraph (μ) to be the average of α and β. Hence, the α and β parameters allow us to change the expected total number of edges in the hypergraph as well as the distribution of edges.

Procedure $genTestcase(N, M, \alpha, \beta, proc_dist)$
1. **for** $(h = 1\ ; h \leq N\ ; h{+}{+})$ **do**
2. **for** $(j = 1\ ; j \leq M\ ; j{+}{+})$ **do**
3. edge_prob = $\alpha - (\alpha - \beta) \times \frac{h}{N}$
4. **if** $((\mathrm{rand}(100) < (edge_prob \times 100))$ **then**
5. $e_{h,j} = 1$
6. **end if**
7. **end for**
8. **end for**
9. **for** $(j = 1\ ; j \leq M\ ; j{+}{+})$ **do**
10. $Q_j = cal_Q(j, proc_dist)$
11. **end for**
12. **for** $(cnt = 1\ ; cnt \leq M\ ; cnt{+}{+})$ **do**
13. a = rand(M)
14. b = rand(M)
15. swap $(e_{x,a}, e_{x,b}),\ x = 1,2,...,N$
16. **end for**

Fig. 8. Testcase generation algorithm.

In Figure 8, we first calculate the R_h for each path, and assign edges $e_{h,j}$ with a random number function $Rand(X)$ that generates an integer from 1 to X. We then calculate and assign Q_j to process conditions randomly using a sub-function $cal_Q()$:

$$cal_Q(j, uniform) = 1/M$$

$$cal_Q(j, Gaussian) = cdf(-3 + \frac{6j}{M}) - cdf(-3 + \frac{6(j-1)}{M})$$

$$cal_Q(j, powerlaw) = \frac{1}{j \times \sum_{m=1}^{M} \frac{1}{m}}$$

Here, $cdf(\cdot)$ is the cumulative distribution function for the normal distribution.

VI. Experimental Results

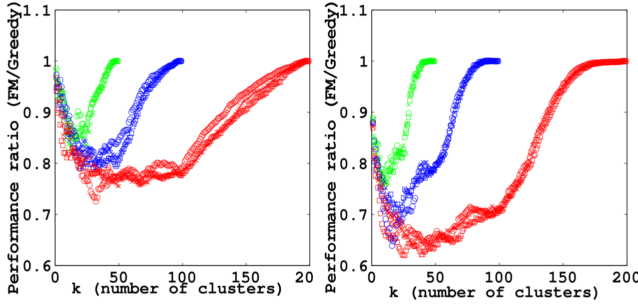
We have implemented both the Greedy algorithm and our proposed method in C++, and have run experiments on a 3.3GHz CPU. We compare the algorithms' solution quality using random testcases generated as described in Section V. We also compare the algorithms using a design testcase provided by the authors of [5]. In the following, we report a *relative performance ratio*, $f(C^M)/f(C^{Greedy})$, where $f(C^{Greedy})$ and $f(C^M)$ are the test costs for the Greedy algorithm and our algorithm, respectively. To reduce impacts of randomization in testcase generation and (tie-breaking) in algorithms, each experiment is performed 10 times, and we report the average of the 10 resulting performance ratios.

A. Experiments with Generated Testcases

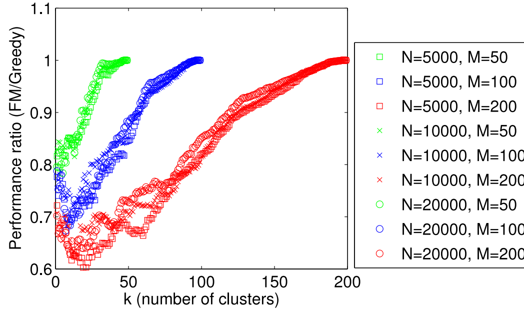
Performance Ratio. Figure 9 shows that the performance ratios of our algorithm versus the Greedy algorithm are consistently less than 1.0 for all values of $k < M$, and for all the testcases. This means that our algorithm gives consistently better clustering results than the Greedy algorithm for the randomly generated testcases. (When $k = M$, performance ratio is 1.0 because there is only one feasible solution.)

As mentioned in Section III, the merging operation in the Greedy algorithm can be prone to generating suboptimal solution choices. Since the number of merging operations increases as k decreases, it is reasonable to suspect that the accumulation of "wrong choices" by the Greedy algorithm also increases. This would be consistent with the observed decrease in performance ratio as k decreases (merging operations in Greedy algorithm = $M - k$). With the generated testcases, performance ratios reach their minimum values at $k < M/2$ and increase sharply after that. The performance ratios increase because absolute costs of both algorithms increase much faster than the difference between them when k approaches 1 (e.g., see Figure 10). Also, the experimental data in Figure 9 show that there are negligible differences from $N = 5000$ to $N = 20000$. This is because the total number of process conditions is much smaller than the number of critical paths. In other words, when each process condition has many critical paths, increasing the total number of critical paths changes the absolute cost but not the performance ratio between the algorithms. The minimum performance ratio seen in our experiments is about 0.60 for testcases with power-law distributed Q_j.

Impact of μ and R_h on Performance Ratio. Experimental results in Figure 11 show that performance ratio increases when μ increases for testcases with uniform and linearly decreasing R_h. When μ is greater than 50% the performance ratio is greater than 1.0 for testcases with large k. This means that the solution obtained by our algorithm has higher cost than the Greedy solution. From Figures 11 (a) and (b), we can see that performance ratios for testcases with nonuniform R_h are slightly higher than for those with uniform R_h. The trend is clearly shown in Figure 12, in which the performance ratios increase along with the variance of R_h or, equivalently, $\alpha - \beta$. We observe that the Greedy algorithm outperforms our algorithm as variance of R_h and μ increases, especially when k is large. This is likely



(a) Uniform Q_j (b) Gaussian Q_j



(c) Power-law Q_j

Fig. 9. Performance ratio of our algorithm versus Greedy for different numbers of paths (N) and process conditions ($\mu = 2\%/M$). Performance ratio smaller than 1.0 means that the clustering solution obtained by our algorithm is better than that of the Greedy algorithm.

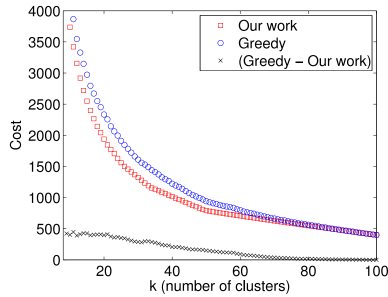


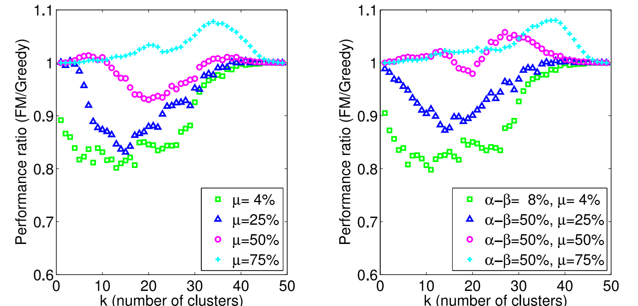
Fig. 10. As k approaches 1, test costs of both solutions increase faster than the difference between them. ($N = 5000$, $M = 100$, Gaussian process distribution.)

due to the number of total splitting operations in our algorithm being monotone in k; more splitting operations reduces the probability of finding an optimal solution. Meanwhile, the trend of optimality versus k is opposite for the Greedy algorithm, which is more likely to obtain an optimal solution for $k \approx M$. Also, when $\alpha - \beta$ is large, many of the process conditions will have similar paths. Such an instance favors the Greedy algorithm, which tends to merge highly overlapped clusters.[7]

Runtime and Performance Ratio Summary. Tables I and II summarize the average performance ratio and the total CPU runtime to generate $M - 1$ clustering solutions for different testcases. The results show that the runtime of our algorithm is proportional to $\mu M^3 N$ while the runtime of Greedy algorithm is proportional to $M^2 N$. The performance ratios (averaged over different k) for different testcases range from 0.78 to 1.03.

Choice of Refinement Steps and Number of Starts. Table III

[7]A metaheuristic that runs both Greedy and our algorithm, and returns the better solution, may be useful in practice.



(a) Uniform R_h (b) Linearly decreasing R_h

Fig. 11. Performance ratio of our algorithm versus Greedy with Gaussian process distribution, $M = 50$ and $N = 10000$. Performance ratio increases along with μ for uniform and linear path distributions.

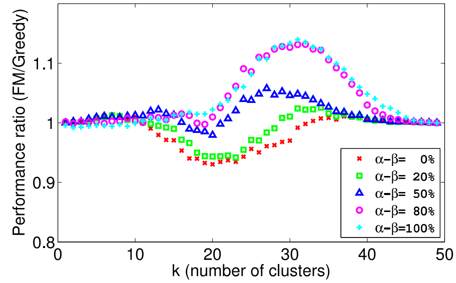


Fig. 12. Performance ratio increases along with the variance of R_h. When $\alpha - \beta > 50\%$ and $k > n/2$, our solution from our algorithm has higher cost than the Greedy solution. Shown: $\mu = 50\%$, $M = 50$, $N = 10000$, Gaussian process distribution.

shows that noticeable improvement results from performing a refinement step (i.e., 0 refinement steps versus 1 refinement step). However, the incremental improvement reduces drastically with further increase in the maximum number of refinement iterations. At the same time, CPU time increases linearly as we increase the maximum number of refinement iterations. We believe that using one refinement step is a good strategy to limit CPU runtime without losing much solution quality. The bipartitioning operation starts from a single randomly generated initial solution. We have tried bipartitioning with multiple initial solutions and choosing the best outcome, i.e., a multi-start approach. However, since multi-start gives very little ($\approx 1\%$) improvement over a single start, we do not pursue this further.

TABLE II

AVERAGE PERFORMANCE RATIO FOR $k = 1, 2, \ldots, M$ AND TOTAL CPU RUNTIME TO GENERATE M SOLUTIONS. $N = 10000$.

M	μ (%)	Uniform R_h perf. ratio	Uniform R_h Runtime (s) Our work	Uniform R_h Runtime (s) Gd.	Linear R_h perf. ratio	Linear R_h Runtime (s) Our work	Linear R_h Runtime (s) Gd.	$\alpha - \beta$ (%)
50	4	0.90	446	21	0.90	444	21	8
50	25	0.94	1077	20	0.95	1193	20	50
50	50	0.98	2319	20	1.01	2603	21	50
50	75	1.03	4412	21	1.03	4534	20	50
25	25	0.98	104	5	1.00	113	5	50
50	25	0.94	1076	20	0.95	1193	20	50
100	25	0.93	12836	85	0.95	14874	84	50

B. Experiments with Industrial Testcase

We have also compared our algorithm and the Greedy algorithm using an industrial testcase obtained from the authors of [5]. In this testcase, threshold voltage variations of PMOS and NMOS transistors are considered. The range of threshold voltage deviation is from -80mV to +80mV for both PMOS

TABLE I

AVERAGE PERFORMANCE RATIO FOR $k = 1, 2, \ldots, M$ AND TOTAL CPU RUNTIME TO GENERATE M SOLUTIONS. UNIFORM PATH DISTRIBUTION ($\alpha - \beta = 0$).

N	M	μ (%)	Uniform process distribution			Gaussian process distribution			Power-law process distribution		
			performance ratio	Our work CPU time (s)	Greedy CPU time (s)	performance ratio	Our work CPU time (s)	Greedy CPU time (s)	performance ratio	Our work CPU time (s)	Greedy CPU time (s)
5000	50	4	0.92	207	10	0.88	206	11	0.91	251.70	10.80
5000	100	2	0.88	1571	43	0.84	1566	44	0.86	1755.90	52.60
5000	200	1	0.85	12845	189	0.78	12826	194	0.81	12633.00	176.10
10000	50	4	0.92	414	21	0.88	423	22	0.92	798.70	36.90
10000	100	2	0.89	3249	89	0.85	3207	91	0.87	4374.80	114.60
10000	200	1	0.85	26476	411	0.80	26552	408	0.83	26644.00	388.10
20000	50	4	0.92	860	43	0.89	853	44	0.92	890.40	40.10
20000	100	2	0.90	7050	194	0.84	7092	204	0.87	7406.60	188.70
20000	200	1	0.86	60398	928	0.80	59216	937	0.84	61543.30	897.90

TABLE III

TRADEOFF BETWEEN IMPROVEMENT AND RUNTIME. INCREMENTAL IMPROVEMENT IS LESS FOR SUCCEEDING REFINEMENT ITERATIONS.

maximum refinement iterations	performance ratio	CPU time (s)	
		Our work	Greedy
0	0.963	2.80	10.70
1	0.920	206.70	10.70
3	0.918	506.80	10.70

(ΔVt_p) and NMOS (ΔVt_p). These ranges of ΔVt_n and ΔVt_p are each divided into 9 sections, creating 81 process condition combinations, as shown in Figure 13. For ΔVt_p we adopt the sign convention that -80mV is "fast" and +80mV is "slow". Each process condition is associated with a set of critical paths, with the total number of critical paths being 6027. Because of the simple relationship between delay and Vt, in this testcase the set of paths that are critical (and hence tested) for a given $(\Delta Vt_n, \Delta Vt_p)$ process condition is a subset of the paths that are critical at any "dominating" process condition with higher ΔVt_n and/or ΔVt_p. In other words, the testcase has structure

$$f_{path}(\Delta Vt'_n, \Delta Vt'_p) \subseteq f_{path}(\Delta Vt_n, \Delta Vt_p) \quad \forall \Delta Vt'_n \leq \Delta Vt_n, \quad \Delta Vt'_p \leq \Delta Vt_p \tag{9}$$

where $f_{path}(\Delta Vt_n, \Delta Vt_p)$ is the set of critical paths at process condition $(\Delta Vt_n, \Delta Vt_p)$. The exact number of paths at each process condition is given in Figure 13: for example, the total number of critical paths at the SS (+80mV,+80mV) corner is 6027, while the total number of critical paths at the FF (-80mV, -80mV) corner is 33. In our study, we consider two types of probability distributions for the process conditions: (1) ΔVt_n and ΔVt_p have independent Gaussian distributions with zero mean and 20mV standard deviation, and (2) each of the 81 process conditions is equiprobable.

Since the computation time of our algorithm is mainly spent on the refinement steps, we experiment with different refinement settings to study the tradeoff between observed performance ratio and the CPU resource. Specifically, we perform refinement only when the number of clusters during partitioning is less than or equal to a *refinement upper bound* that takes on values {0, 15, 30, 81}.

Experiment results in Figure 14 show that our algorithm always achieves a lower test cost than the Greedy algorithm for both process condition probability distributions. The minimum values achieved for the relative performance ratio are 0.56 and 0.65 for the Gaussian and uniform process condition distributions, respectively. Values of performance ratio are similar for different refinement settings, except for the case

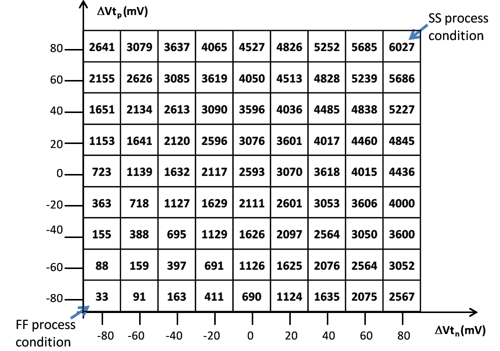


Fig. 13. Number of paths at each condition in the industry testcase supplied by the authors of [5].

where refinement upper bound = 0 (no refinement) case. This implies that refinement steps at the beginning of partitioning (when the number of clusters is small) have more benefit than later refinement steps.

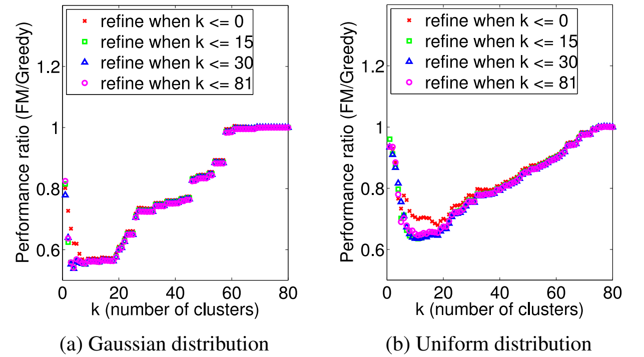


Fig. 14. Relative performance ratio of our algorithm versus the Greedy algorithm with different refinement options and process condition probability distributions, for an industry testcase received from the authors of [5]. Minimum achieved values of performance ratio are 0.56 and 0.65 for Gaussian and uniform process condition distributions, respectively.

Table IV shows incremental benefits of increasing the refinement upper bound. The performance ratio does not improve further with refinement upper bound greater than 30. The data suggest that use of a smaller refinement upper bound can substantially reduce CPU cost with negligible degradation of test cost.

C. Testcase-Specific Algorithms

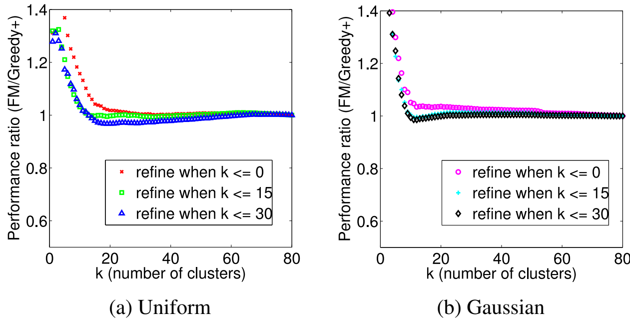
The testcase in Figure 13 has the unique property that *adjacent process conditions* (i.e., two process conditions are next to each other in the gridded 2-D process space) have

TABLE IV

TRADEOFF BETWEEN TEST COST IMPROVEMENT AND RUNTIME, ACCORDING TO THE REFINEMENT UPPER BOUND. INCREMENTAL BENEFITS DECREASE WITH FURTHER INCREASES IN THE UPPER LIMIT ON NUMBER OF CLUSTERS BEING REFINED.

	Gaussian			Uniform		
refine	perf.	CPU time (s)		perf.	CPU time (s)	
when	ratio	Our Work	Greedy	ratio	Our Work	Greedy
$k \leq 0$	0.796	15	36	0.838	15	32
$k \leq 15$	0.786	88	36	0.820	87	32
$k \leq 30$	0.784	716	36	0.818	627	32

many shared critical paths. The Greedy algorithm in [5] exploits this property and applies an additional constraint such that its merging operation only considers adjacent process conditions. This additional constraint ensures that the Greedy algorithm only merges process conditions with overlapping paths. We have implemented this testcase-specific Greedy clustering algorithm (*Greedy+*) with the added constraints, and have compared the performance with the proposed method. Figure 15 shows that the modified Greedy approach outperforms our algorithm especially when k is small. This is because our algorithm ignores the systematic process condition distribution and randomly selects an initial solution. As a result, the performance ratio is larger than 1.0 at the beginning and eventually converges to 1.0 as the number of clusters increases.



(a) Uniform (b) Gaussian

Fig. 15. Relative performance ratio of our algorithm versus the Greedy algorithm with different refinement options and process condition probability distributions, for the 9x9 testcase received from the authors of [5].

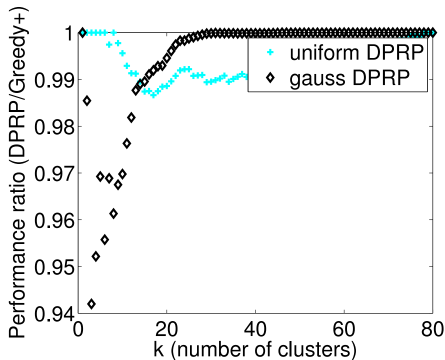


Fig. 16. Relative performance ratio of our DPRP algorithm versus the Greedy+ algorithm with different process condition probability distributions, for the 9x9 and 33x33 testcases received from the authors of [5]. A higher performance ratio indicates relatively stronger performance of Greedy+ relative to our algorithm.

To better match the unique properties of the industrial testcase, we have implemented the dynamic-programming restricted partitioning (DPRP) approach of [1]. In the DPRP algorithm, all process conditions in Figure 13 are ordered in a 1-D array based on the merging sequence of the Greedy+ algorithm. If a merging operation in the Greedy+ algorithm merges two clusters C_1 and C_2, then the process conditions of C_1 and C_2 are adjacent to each other in the 1-D order (all of C_1's process conditions follow all of C_2's or vice-versa, with no intervening other conditions.).

S_2 is placed after S_1 in the 1-D array. After the merging and ordering operations, we apply the k-way DPRP algorithm [1] to partition the 1-D array into k clusters of process conditions to minimize the expected test cost. Results in Figure 16 show that the performance ratio of DPRP compared to Greedy+ is always smaller than 1.0. The minimum performance ratio is 0.95, which is equivalent to 5% test path reduction. The runtime of the proposed DPRP algorithm is similar to that of the Greedy+ algorithm, which is proportional to M^2N. In our experiments, the runtime of DPRP is approximately 10% more than that of Greedy+.

VII. CONCLUSION

In this work, we have studied the path clustering problem for adaptive test. Via a family of constructions, we show that the recent Greedy algorithm of [5] can result in test costs at least twice optimal. To reduce the cost of adaptive test, we formulate the clustering problem as hypergraph partitioning, and apply a heuristic in the framework of the classic FM min-cut bipartitioning algorithm. We show that our method can reduce test cost by as much as 40% versus the Greedy algorithm for generated testcases. In case runtime is a concern, we show that the *refinement upper bound* parameter in our approach can be tuned to reduce computation time with little loss of solution quality. We also observe that while expected complexity of the adaptive (i.e., process condition-specific) testing reduces monotonically with increased number of clusters, more clusters may reduce the efficiency of tester hardware [5]. This tradeoff between test path reduction and tester efficiency should be comprehended in setting the target number of clusters.

Finally, our experiments on an industry testcase obtained from the authors of [5] show that the Greedy+ algorithm (a modified Greedy algorithm which accounts for relationships among process conditions) has lower test cost than the FM-based heuristic, especially when k is small. To improve the test cost achieved by the Greedy+ algorithm, we have proposed integration of a DPRP clustering step, which can improve the test cost of Greedy+ by as much as 5%.

ACKNOWLEDGMENT

We are grateful to Professor Takashi Sato of Kyoto University for his very helpful correspondence and feedback, and for providing the testcase from [5].

REFERENCES

[1] C. J. Alpert and A. B. Kahng, "Multi-Way Partitioning Via Spacefilling Curves and Dynamic Programming", *Proc. Design Automation Conference*, 1994, pp. 652-657.

[2] C. M. Fiduccia and R. M. Mattheyses, "A Linear-Time Heuristic for Improving Network Partitions," *Proc. Design Automation Conference*, 1982, pp. 175-181.

[3] E. N. Gilbert, "Random graphs," *Annals of Mathematical Statistics* (30) (1959), pp. 1141-1144.

[4] M. Shintani, T. Uezono, T. Takahashi, H. Ueyama, T. Sato, K. Hatayama, T. Aikyo and K. Masau, "An Adaptive Test for Parametric Faults Based on Statistical Timing Information," *Proc. IEEE Asian Test Symposium*, 2009, pp. 151-156.

[5] T. Uezono, T. Takahashi, M. Shintani, K. Hatayama, K. Masu, H. Ochi and T. Sato, "Path Clustering for Adaptive Test," *Proc. IEEE VLSI Test Symposium*, 2010, pp. 15-20.