MC-MIPOG: A Parallel t-Way Test Generation Strategy for Multicore Systems

Mohammed I. Younis and Kamal Z. Zamli

Combinatorial testing has been an active research area in recent years. One challenge in this area is dealing with the combinatorial explosion problem, which typically requires a very expensive computational process to find a good test set that covers all the combinations for a given interaction strength (*t*). Parallelization can be an effective approach to manage this computational cost, that is, by taking advantage of the recent advancement of multicore architectures. In line with such alluring prospects, this paper presents a new deterministic strategy, called multicore modified input parameter order (MC-MIPOG) based on an earlier strategy, input parameter order generalized (IPOG). Unlike its predecessor strategy, MC-MIPOG adopts a novel approach by removing control and data dependency to permit the harnessing of multicore systems. Experiments are undertaken to demonstrate speedup gain and to compare the proposed strategy with other strategies, including IPOG. The overall results demonstrate that MC-MIPOG outperforms most existing strategies (IPOG, IPOF, IPOF2, IPOG-D, ITCH, TConfig, Jenny, and TVG) in terms of test size within acceptable execution time. Unlike most strategies, MC-MIPOG is also capable of supporting high interaction strengths of $t > 6$.

Keywords: t-way testing, multi-way testing, combinatorial testing, parallel processing.

Manuscript received May 7, 2009; revised July 24, 2009; accepted Sept. 22, 2009.

This work was supported by the Research University Grant–'Imaging' & Post Graduate Research Grant–'T-Way Test Data Generation Strategy Utilizing Multicore System' of Universiti Sains Malaysia.

Mohammed I. Younis (phone: +60 4 5996003, email: younismi@gmail.com) and Kamal Z. Zamli (email: eekamal@eng.usm.my) are with School of Electrical and Electronics Engineering, USM Engineering Campus, Pulau Penang, Malaysia.

doi:10.4218/etrij.10.0109.0266

I. Introduction

As an activity for ensuring quality and improving reliability, software testing is an important phase in the software engineering lifecycle. Lack of testing often leads to disastrous consequences including loss of data, fortunes, and even lives. For these reasons, many input parameters and system conditions need to be tested against the system's specifications for conformance. Although desirable, exhaustive testing can be prohibitive due to resource and timing constraints.

Earlier works [1], [2] conclude that pairwise testing based on 2-way interaction of variables can be effective to detect most faults in a typical software system. While this conclusion may be true for some systems, it cannot be generalized to all software system faults, especially when there are significant interactions between variables. For example, the study by the National Institute of Standards and Technology (NIST) [2]-[4] reported that 95% of the actual faults on the test software involve 4-way interaction. In fact, all of the faults are detected with 6-way interaction [5], [6].

In general, the consideration of higher interaction strengths (that is, from $t = 3$ upwards) can be problematic. When the parameter interaction coverage t increases to more than 2, the number of t-way test sets also increases exponentially. For example, consider a system with 10 parameters, where each parameter has 5 values. There are 1,125 2-way tuples (or pairs), 15,000 3-way tuples, 131,250 4-way tuples, 787,500 5-way tuples, 3,281,250 6-way tuples, 9,375,000 7-way tuples, 17,578,125 8-way tuples, 19,531,250 9-way tuples, and 9,765,625 10-way tuples. From this illustrative example, it is evident that for a large system with many parameters, considering a higher-order t-way test set can lead toward a combinatorial explosion problem.

Due to this combinatorial explosion problem, there are few results reported in the literature on t-way testing for high t values greater than 6. From one perspective, the recent advances in computing and hardware technologies dictate that software applications need to incorporate many new features and functionalities based on consumer demands. As such, software applications have grown tremendously from kilobytes to terabytes. From another perspective, the net effect of software growth can often lead to intertwined dependency between parameters involved, thus justifying the need to support high interaction strength.

Facing these challenges, this paper presents a new deterministic strategy, called *multicore modified input parameter order (*MC-MIPOG). Unlike its predecessor strategy (IPOG), MC-MIPOG removes the control and data dependency to permit the harnessing of multicore systems. This paper also explores the current state-of-the-art and discusses the similarities and differences among several variants of IPOG within the literature. Additionally, a number of experiments undertaken are discussed to demonstrate the speedup gain. Finally, comparisons with other existing strategies, namely, TConfig [7], Jenny [8], TVG [9], ITCH [10], IPOG [3], IPOG_D [11], and IPOF [12], [13] are also demonstrated. For most cases, MC-MIPOG outperforms other existing strategies in terms of test size and supports a high degree of interaction (t).

The rest of this paper is organized as follows. Section II presents a state-of-the-art review of the existing strategies, section III provides the details of the proposed MIPOG strategy and how it varies from the original IPOG. Section IV provides a detailed description of MC-MIPOG and discusses its implementation. Section V reports evaluation experiments. Finally, section VI states our conclusions and suggestions for future works.

II. Related Work

Combinatorial testing strategies can be classified as either computational or algebraic strategies [14], [15]. Most algebraic approaches compute test sets directly by a mathematical function [16]. Algebraic approaches are often based on the extensions of mathematical methods for constructing orthogonal arrays (OA) [17], [18]. Some variations of the algebraic approach also exploit recursion in order to permit the construction of larger test sets from smaller ones [19]. Thus, the computations involved in algebraic approaches are typically lightweight and not subject to the combinatorial explosion problem. For this reason, strategies that are based on algebraic approach are extremely fast. On the other hand, algebraic approaches often impose restrictions on the system configurations to which they can be applied [20], [21]. This significantly limits the applicability of algebraic approaches for software testing [11].

Unlike algebraic approaches, computational approaches often rely on the generation of all tuples and search the tuple space to generate the required test suite until all tuples have been covered. When the number of tuples to be considered is significantly large, adopting computational approaches can be expensive, especially in terms of the space required to store the tuples and the time required for explicit enumeration. Unlike algebraic approaches, computational approaches can be applied to arbitrary system configurations. Furthermore, computational approaches are more adaptable for constraint handling [22] and test prioritization [23].

Earlier works in combinatorial testing identify two strategies, namely the automatic efficient test generator (AETG) [24] and input parameter order (IPO) [25]. The AETG builds a test set one test at a time until all the tuples are covered [24], [26]. AETG and its variants [27], [28] are later generalized into a general framework to support multi-way interaction ($t \leq 6$) [29], [30].

In contrast, IPO covers one parameter at a time. This allows IPO to achieve a lower order of complexity than AETG [1]. IPO is a pairwise strategy (interaction strength $t = 2$) based on vertical and horizontal extension. The IPO strategy first generates a pairwise test set for the first two parameters. It then continues to extend the test set to generate a pairwise test set for the first three parameters and continues to do so for each additional parameter until all the parameters of the system are covered via horizontal extension. If required for interaction coverage, IPO also employs vertical extension in order to add new tests after the completion of horizontal extension. Later, IPO is generalized into IPOG [3]. Several IPOG variants have been proposed to improve its performance, including IPOG-D [11], IPOF [12], and IPOF2 [12].

Both IPOG and IPOG-D are deterministic strategies. Unlike IPOG, IPOG-D combines the IPOG strategy with an algebraic recursive construction called D-construction in order to reduce the number of tuples to be covered. In fact, Lei and others reported that when $t = 3$, IPOG-D is degraded to a D-construction algebraic approach [31]. Here, when $t > 3$, a minor version of IPOG is used to cover the uncovered tuples missed during D-construction [11]. As such, IPOG-D tends to be faster than IPOG, though with a larger test set.

Unlike IPOG and IPOG-D, IPOF is a non-deterministic strategy. For this reason, IPOF produces a different test set for each run. In short, the IPOF strategy is a refinement of IPO, the base pairwise strategy for IPOG, which is used to generate test sets for the uniform distribution of variables. Unlike IPOG, IPOF rearranges the rows during horizontal extension in order to cover more tuples per horizontal extension. Published results have reported the performance of IPOF but with a small

number of parameter values. Similarly, a variant of IPOF called IPOF2 [12] is also available but it has only been demonstrated with a small number of parameter values. Unlike IPOF, IPOF2 uses a heuristic technique for covering the tuples, which allows a faster execution time than IPOF but with a larger test set. Currently, IPOG, IPOG_D, IPOF1, and IPOF2 are integrated into one tool called the Advanced Combinatorial Testing Suite (ACTS) available at NIST [32]. Four other strategy implementations that are available for download with minimal documentation are TConfig [7], Jenny [8], TVG [9], and ITCH [10]. Note that all these strategies are implemented using Java language except Jenny, which is implemented in C language.

While there are many useful variants of IPOG, little attention has been given to parallelizing the test generation process. Because existing IPOG variants appear to have many inherent unwanted control and data dependencies which hinder parallelism due to case specific optimization processes for horizontal and vertical extension, we have opted for the original IPOG as our base strategy. In fact, we have introduced a new variant of IPOG, called modified IPOG (MIPOG) which demonstrates the feasibility of removing the inherited dependencies from IPOG and thus helps in the implementation of MC-MIPOG. The MIPOG strategy will be elaborated in the next section.

```
Algorithm IPOG-Test (int t, ParameterSet ps)
{
1. initialize test set ts to be an empty set;
2. denote the parameters in ps, in an arbitrary order, as P1,
   P2,⋯, and Pn;
3. add into test set ts a test for each combination of values
   of the first t parameters;
4. for (int i = t + 1; i ≤ n; i ++){
5.    let π be the set of t-way combinations of values
      involving parameter Pi and t – 1 parameters among
      the first i – 1 parameters;
6.    // horizontal extension for parameter Pi
7.     for (each test τ = (v1, v2,⋯, vi–1) in test set ts) {
8.          choose a value vi of Pi and replace τ with τ′ =
            (v1, v2,⋯, vi–1, vi) so that τ′ covers the most
            number of combinations of values in π;
9.          remove from π the combinations of values
            covered by τ′;
10.    }
11.    // vertical extension for parameter Pi
12.    for (each combination σ in set π){
13.       if (there exists a test that already covers σ) {
14.          remove σ from π;
15.       } else {
16.          change an existing test, if possible, or otherwise
             add a new test to cover σ and remove it from π;
17.       }
18.    }
19. }
20. return ts;}
```

Fig. 1. IPOG strategy.

III. MIPOG Strategy

In this section, we introduce the MIPOG strategy and demonstrate how it can be parallelized into MC-MIPOG. We also highlight the similarities and differences between MIPOG and IPOG.

Despite the fact that it is an efficient strategy, we note that the generation of a test set (ts) can be unstable in IPOG (see Fig. 1) due to the possibility of the current test case changing during the vertical extension (especially for test cases that include "don't care" value). This raises the issue of dependency between previously generated test cases and the new one.

To address this dependency issue, we have considered variant algorithms for both horizontal and vertical extension to remove dependencies (see the MIPOG strategy in Fig. 2).

For horizontal extension, the MIPOG strategy checks all the values of the input parameter and chooses the value that contains the maximum number of combinations for the uncovered tuples in the π set. Also, MIPOG optimizes the don't care value. For this reason, MIPOG always generates a stable test case which cannot be modified by searching for tuples that can be covered by the same test. This is performed by means of exhaustive searching of uncovered tuples that can be combined with this test case during horizontal extension (to ensure that the test case is indeed optimized). For vertical extension, MIPOG rearranges the π set in decremented size order. After that, MIPOG chooses the first tuple from the rearranged π set and combines that tuple with other suitable tuples in the π set. That is, the resulting test case must have the maximum weight of the uncovered tuples found through exhaustive searching of uncovered tuples. Once combined, all these tuples are removed from the π set. This process is repeated until the π set is empty to ensure complete interaction coverage.

To illustrate the differences between IPOG and MIPOG horizontal and vertical extension, we consider a system with 4 parameters (three 2-valued and one 3-valued parameters). Figures 3 and 4 show the process of generating a 3-way test set using IPOG and MIPOG, respectively. Here, MIPOG generates a minimal test set (3×2×2=12 values), while IPOG generates 14 test cases.

As shown in Fig. 3, IPOG decides on the parameter value assignment early in horizontal extension. Apart from ensuring most pairs are covered at the instant of assignment, IPOG also ensures that each parameter value is as equally balanced as possible. In contrast, MIPOG decides on the parameter value assignment late (see Fig. 4), that is, only after first scanning all the parameter values to yield the most optimal solution (with maximum weight).

In vertical extension, IPOG iteratively checks for uncovered

```
Algorithm MIPOG-Test (int t, ParameterSet ps)
{
1. initialize test set ts to be an empty set;
2. denote the parameters in ps, in an arbitrary order, as P1,
   P2,⋯, and Pn;
3. add into test set ts a test for each combination of
   values of the first t parameters;
4. for (int i = t + 1; i ≤ n; i ++){
5.       let π be the set of t-way combinations of values
         involving parameter Pi and t -1 parameters among
         the first i – 1 parameters;
6.       // horizontal extension for parameter Pi
7.       for (each test τ = (v1, v2,⋯, vi–1) in test set ts) {
8.          if (τ does not contain don't care){choose a value vi
               of Pi and replace τ with τ'= (v1, v2, dc,⋯, vi-1,
               vi) so that τ' covers the maximum number of
               combinations of values in π;}
9.          else {choose a value vi of Pi and search all
               possible tuples that can optimize the don't care
               (dc) to construct τ' = (v1, v2,⋯, vi–1, vi) so that
               τ' covers the maximum number of
               combinations of values in π and optimized dc;}
10.         remove from π the combinations of values covered
             by τ';}
11.      // vertical extension for parameter Pi
12. while (π is not empty){
13.     rearrange π in decreasing order;
14.     choose the first tuple and generate test case (τ) to
         combine maximum number of tuples;
15.     delete the tuples covered by τ, add τ to local ts;
16. } //while
17. } // for
18. return ts;}
```

Fig. 2. MIPOG strategy.

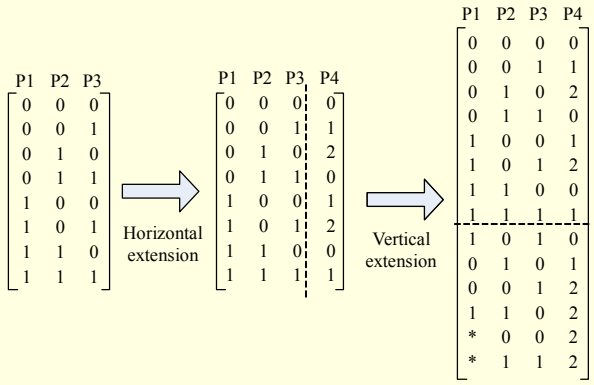

Fig. 3. Generation of test set using IPOG.

t-way combinations from the horizontal extension and adds the combination into a new test in the vertical extension, often using already covered *t*-way combinations. In a similar manner, MIPOG also checks for uncovered *t*-way combinations from the horizontal extension. However, MIPOG optimizes the addition of a new test in the vertical extension by combining

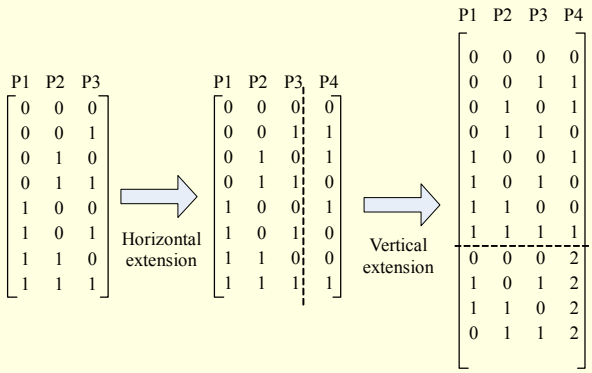

Fig. 4. Generation of test set using MIPOG.

the most uncovered *t*-way combinations whenever possible. This is efficiently done when there are don't care values. This step, while improving the test size, also increases the overall computation of MIPOG.

The net effect of the variant extension algorithms in MIPOG is twofold. First, we can always get a more optimal test set which would be at least the same size or even smaller than that of IPOG. Secondly, there are no dependencies between subsequently generated test values, thus, permitting the possibility of parallelization.

To parallelize MIPOG, we can partition the π set for parameter P*i* into *vi* partitions (see Fig. 2). As a result, the generation of each partition can be performed in a separate thread. Additionally, both horizontal and vertical extension can be performed in separate guarded (synchronized) threads.

In the next section, we discuss the parallel version of MIPOG, called MC-MIPOG designed specifically for Intel Multicore system [33].

IV. MC-MIPOG Strategy

Built from MIPOG, the MC-MIPOG strategy distributes the computational processes and memory into pieces. In summary, the MC-MIPOG strategy implementation is based on the following design criteria:

- Memory needs to be distributed in order to hold P*i* in relatively independent cells, called π[V*i*]. Here, each π[V*i*] needs to have its own memory to hold the *t*-way combinations for a unique particular value for the parameter P*i*; that is, there are V*i* partitions for π. In this case, each partition is generated by a separate thread, called a *combinatorial thread*.
- There are V*i* separate threads for horizontal extension, called *horizontal extension threads*.
- Similarly, there are also V*i* separate threads for vertical extension, called *vertical extension threads*.

- The selected test set is stored into a shared memory controlled by the test generator (master) program which controls the creation, synchronization, and deletion of all of these mentioned threads.

Note that the latest optimization in multicore systems with multitasking operating systems (as in Linux and Windows) manages processor/affinity in an optimal way [34]. This optimization enables each software thread to be mapped into the same hardware thread while keeping the data close to the processor through a process called a cache worm [33], [35]. Thus, the actual control of processor and memory affinity is automatically performed by the operating system.

1. Test Generator (Main Program)

As implied earlier, the main program roles are to manage the shared memory and to coordinate threads. Briefly, the main program works as follows:

1) Start with an empty test set (ts), and generates all tuples for the first *t*-parameters.
2) Create combinational threads (equal to the number of values in P*i*), passing to them parameters values (P1⋯P*i*-1).
3) Wait for all combinational threads to finish their generation, and then read π[V*i*'s].
4) Shut down the combinational threads.
5) Create horizontal extension threads (equal to the number of values in P*i*), passing to them π[V*i*'s], and V*i*'s for parameter P*i*.
6) For horizontal extension:
   a. For each test case τ in test set ts:
   b. Wait until all threads have validated results.
   c. Read the weight (that is, the number of covered tuples after adding the assigned value) from each thread. Then, choose the value corresponding to the maximum weight to be added to ts if no tuples match (weight zero) don't care added to τ.
   d. Notify the horizontal extension threads that validate that selection is done.
   e. According to the selection in *c*, issue command to the selected thread to delete tuples from their own π set (πv). Allow the selected threads to update τ.
   f. Wait for selected thread to finish its work.
7) Shut down the horizontal threads.
8) Create vertical extension threads equal to the number of values in P*i*, pass them to π[V*i*'s], and V*i*'s for parameter P*i*.
9) In vertical extension:
   a. Wait for the threads to finish their partial test set (tsvth).
   b. Collect tsvths from the threads. Then add each tsvth to ts.
10) Shut down the vertical threads.

```
Algorithm Main (int t, ParameterSet ps)
{
1. initialize test set ts to be an empty set;
2. denote the parameters in ps, in an arbitrary order, as P1,
   P2,⋯, and Pn;
3. add into test set ts a test for each combination of values
   of the first t parameters;
4. for (int i = t + 1; i ≤ n; i ++){
5.       //create combinational
6.       for (each value (v) in Pi) {
             start combinational thread (t, ps, i, set π[v]);
7.           wait for combinational threads to finish;}
8.           // horizontal extension for parameter Pi
9.       for (each value (v) in Pi) {
             start horizontal thread (ts, set π[v],v); }
10.      for (each test τ = (v1, v2,⋯, vi-1) in test set ts) {
11.          wait for notification;
12.          read the weight from each horizontal thread;
             choose vi corresponding to maximum weight;
13.          if (max weight !=0) {
14.              issue command delete to horizontal thread [vi],
                 and cancel commands to all others threads }//if
15.          else { append don't care to τ;
16.              issue cancel commands to all threads; } //else
17.          notify all waiting threads;
18.       } //for
19.       wait for horizontal threads to finish;
20.         // vertical extension for parameter Pi
21.       for (each value (v) in Pi) {
              start vertical thread (lts[v], set π[v],v);
22.           wait for vertical threads to finish;}
23.       for (each value (v) in Pi){
24.           add each lts[v] to ts;}// loop v
25. }//loop i
26. return ts;}// algorithm
```

Fig. 5. Algorithm for master program.

For clarity, the complete algorithm for the master program is given in Fig. 5.

2. Working Threads

In this section, we will describe how each thread work.

1) For combinational threads:
   a. Each thread generates its own partial tuples set (πv).
   b. Each thread notifies the master.
2) For horizontal extension threads:
   a. Read next test case τ in test set ts.
   b. If τ does not contain don't care, determine the weight of τ.
   c. If τ contains don't care, the thread optimizes the don't care value to have as much weight as possible.
   d. Validate the weight by notification.
   e. Wait for notification.
   f. Read the command issued from main, if it contains delete then the thread deletes tuples covered by τ from (πv); then, append *v* to τ (in case b) or delete τo (in case

```
Algorithm Combinational Thread
(int t, ParameterSet ps, int i, set πv )
{
1. generate local π, where π is the set of t-way Combinations
   of values and  t – 1 among the first i – 1 parameters;
2. notify the waiting process;
}//algorithm
```

Fig. 6. Algorithm for combinational thread.

c) from (πv); then, replace τ with τo.

g. In the case of deletion in f, notify the waiting process.

3) For vertical extension threads:

a. Arrange πv in decreasing order, choose the first tuple, and generate the test case with maximum weight.

b. Repeat step (a) until (πv) is empty.

c. Notify the waiting process.

The complete algorithms for the combinational thread, horizontal extension thread, and vertical extension thread are given in Figs. 6, 7, and 8, respectively.

V. Evaluation

Our evaluation has three main aims. First, we compare the behavior of MC-MIPOG to that of IPOG in terms of the test size ratio. Secondly, we investigate whether there is speedup gain from parallelizing MIPOG in MC-MIPOG. Finally, we compare the effectiveness of the MC-MIPOG strategy to that of other strategies (including that of other IPOG variants) in terms of the generated execution time and test size.

1. MC-MIPOG Behavior against IPOG

To compare the behavior of MC-MIPOG and IPOG, we performed a group of experiments adopted from Lei and others [3]. In these experiments, we are interested to compare the test sizes of MC-MIPOG and IPOG. Note that the IPOG test size is obtained from [3].

- Group 1: The number of parameters (P) and the values (V) are constant, but the coverage strength (t) is varied from 2 to 7.
- Group 2: The coverage strength (t) and the values (V) are constant to 4 and 5, but the number of parameter (P) is varied from 5 to 15.
- Group 3: The number of parameter (P) and the coverage strength (t) are constant from t to 10 and 4, respectively, but the values (V) are varied from 2 to 10.

The results of the experiments are shown in Tables 1, 2, and 3, respectively. Here, we define the size ratio as the size of the test set from MC-MIPOG to the size obtained from IPOG.

```
Algorithm Horizontal Extension Thread
(test set ts, set π, value v)
{
 1. for (i=1··ts size) {
 2.   τ=ts[i]+v;
 3.   if (τ not contains don't care) {
 4.     determine the weight of τ in π;
 5.     notify the waiting process;
 6.     wait for notification of master command;
 7.     if (delete command){
 8.         delete tuples covered by τ from π;
 9.         ts[i]=ts[i]+v;
10.         notify the master that the delete done;
11.     }//if delete
12.   }
13.   else {produce τo (optimize don't care in τ );
14.     determine the weight of τ in π;
15.     notify the waiting process;
16.     wait for notification of master command;
17.     if (delete command{
18.       delete tuples covered by τ from π;
19.       ts[i]= τo;
20.       notify the master that the delete done; }//if delete
21.   }//else
28. }//loop i
29. } //algorithm
```

Fig. 7. Algorithm for horizontal extension thread.

```
Algorithm vertical extension thread
(local test set lts,set π,value v)
{
1. while (π!empty){
2.   arrange π in decreasing order;
3.   choose the first tuple and generate test case that
      combine maximum number of tuples (τ);
4.   delete the tuples covered by τ;
5.   append v to τ;
6.   add τ to lts;
7. } //while
8. notify waiting process;
9. } //algorithm
```

Fig. 8. Algorithm for vertical extension thread.

From Tables 1 to 3, it is evident that MC-MIPOG performs better than IPOG in terms of test size because the size ratio is always < 1. In Table 3, NS indicates that the parameter and values chosen with a given strength are not supported.

Although comparing quite well with IPOG, MIPOG's test size is not the most optimal compared to Colbourn's best-known published results [36]. Nonetheless, on a positive note, MIPOG contributes to finding the optimal test size for (t = 5, p = 10, v = 5) that yields 8,169 instead of 8,555 as reported by Colbourn. In fact, MIPOG also reports a new optimal test size for (t = 7, p = 10, v = 5) that yields 186,664. Note that this result for (t = 7, p = 10, v = 5) has not been reported by Colbourn.

Table 1. Size ratio results for 5 to 15 parameters with 5 values in 4-way testing.

# of parameters	5	6	7	8	9	10	11	12	13	14	15
MC-MIPOG size	625	625	1,125	1,348	1,543	1,643	1,722	1,837	1,956	2,051	2,150
IPOG size	784	1,064	1,290	1,491	1,677	1,843	1,990	2,132	2,254	2,378	2,497
Size ratio	0.797	0.587	0.872	0.928	0.92	0.891	0.865	0.861	0.868	0.862	0.861

Table 2. Size ratio results for 10 parameters with 2 to 10 values in 4-way testing.

# of values	2	3	4	5	6	7	8	9	10
MC-MIPOG size	43	217	637	1,643	3,657	5,927	11,355	18,036	27,306
IPOG size	46	229	649	1,843	3,808	7,061	11,993	19,098	28,985
Size ratio	0.934	0.948	0.981	0.891	0.96	0.839	0.946	0.944	0.942

Table 3. Size ratio results for 10 parameters with 5 values for $t = 2$ to 7.

t-way	2	3	4	5	6	7
MC-MIPOG size	45	281	1,643	8,169	45,168	186,664
IPOG size	48	308	1,843	10,119	50,920	NS
Size ratio	0.938	0.912	0.891	0.807	0.887	-

Table 4. Speedup results for 5 to 15 parameters with 5 values in 4-way testing.

# of parameters	5	6	7	8	9	10	11	12	13	14	15
MIPOG	0.128	0.31	0.778	1.981	3.735	6.9	10.642	19.39	44.169	71.104	143.29
MC-MIPOG	0.15	0.269	0.57	1.272	2.275	3.818	5.803	10.298	21.171	33.213	60.931
Speedup	0.8533	1.152	1.365	1.557	1.642	1.807	1.833	1.883	2.086	2.14	2.352

Table 5. Speedup results for 10 parameters with 2 to 10 values in 4-way testing.

# of values	2	3	4	5	6	7	8	9	10
MIPOG	0.148	0.408	1.39	6.9	68.031	70.495	4,767.778	5,203.01	56,786.346
MC-MIPOG	0.141	0.383	0.983	3.818	35.62	36.151	1,538.719	1,605.372	16,220.036
Speedup	1.05	1.065	1.414	1.807	1.91	1.95	3.099	3.241	3.501

Table 6. Speedup results for 10 parameters with 5 values for $t = 2$ to 7.

t-way	2	3	4	5	6	7
MIPOG	0.074	0.327	6.9	197.928	4,025.442	82,668.19
MC-MIPOG	0.09	0.281	3.818	94.498	1,311.209	23,512.7
Speedup	0.822	1.163	1.807	2.095	3.07	3.516

2. Speedup Gain in MC-MIPOG

To measure the speedup gain from parallelizing MIPOG, we subjected both MIPOG and MC-MIPOG to three experimental groups described earlier. The results of the experiments are shown in Tables 4, 5, and 6. Here, the speedup is defined as the ratio of the time taken by the sequential MIPOG algorithm to the time taken by MC-MIPOG algorithm. All the results were obtained using the Linux Centos OS with a 2.4 GHz Core 2 Quad CPU [34] and 2 GB RAM with JDK 1.5 installed. Note that the execution time is in seconds, and both MIPOG and MC-MIPOG produce the same test set in all cases.

Table 7. Comparative test size results using the TCAS module for $t = 2$ to 12.

t	MC-MIPOG	IPOG	IPOG-D	IPOF1	IPOF2	ITCH	Jenny	TConfig	TVGII
2	100	100	130	100	100	120	106	100	101
3	400	401	487	402	427	2,388	411	472	434
4	1,265	1,367	2,522	1,352	1,644	1,484	1,527	1,476	1,599
5	4,196	4,230	5,306	4,290	5,018	NS	4,680	NS	4,773
6	10,851	10,956	14,480	11,234	13,310	NS	11,608	NS	12,732
7	26,061	NS	NS	NS	NS	NS	27,630	NS	NS
8	56,742	NS	NS	NS	NS	NS	58,865	NS	NS
9	120,361	NS	NS	NS	NS	NS	NS	NS	NS
10	201,601	NS	NS	NS	NS	NS	NS	NS	NS
11	230,400	NS	NS	NS	NS	NS	NS	NS	NS
12	460,800	NS	NS	NS	NS	NS	NS	NS	NS

Table 8. Comparative test generation time using the TCAS module for $t = 2$ to 12.

t	MC-MIPOG	MIPOG	IPOG	IPOG-D	IPOF1	IPOF2	ITCH	Jenny	TConfig	TVGII
2	0.166	0.125	0.047	<0.001	0.015	0.016	0.68	0.001	>1 hour	2.56
3	0.28	0.324	0.313	0.015	0.078	0.109	1,015.2	0.697	>12 hour	2.96
4	2.303	3.027	2.156	0.302	1.805	1.52	5,102.1	3.35	>20 hour	101.1
5	21.672	32.074	13.39	3.11	8.566	8.611	NS	41.32	>1 day	1,369.3
6	194.135	288.485	60.05	33.22	55.11	46.86	NS	466.27	>1 day	>20 hours
7	959.124	1,564.332	NS	NS	NS	NS	NS	235.4	NS	>1 day
8	5,739.74	9,441.872	NS	NS	NS	NS	NS	11698.2	NS	>1 day
9	18,857.175	32,245.77	NS	NS	NS	NS	NS	>1 day	NS	>1 day
10	21,903.542	38,594.041	NS	NS	NS	NS	NS	>1 day	NS	>1 day
11	7,910.768	14,263.114	NS	NS	NS	NS	NS	>1 day	NS	>1 day
12	25.031	25.031	NS	NS	NS	NS	NS	>1 day	NS	>1 day

As seen in Table 4, the speedup increases linearly as the number of parameters increases. Here, extra overhead is added for the fifth parameters due to the need to start and shut down the corresponding threads. As seen in Table 5, the speedup gain also increases quadratically as the number of values increases. Extrapolating and performing curve fitting of the results from Table 6, we observe that the speedup increases logarithmically as the strength of coverage increases. In this case, there is also no speedup gain for this strategy when $t = 2$, possibly due to the overhead required for creation, synchronization, and deletion of threads for a small degree of interaction.

3. Comparison with Other Strategies

To investigate the effectiveness of the MC-MIPOG strategy against other strategies, including IPOG and its variants, in terms of test size and the number of generated test sets, we adopt a common configuration system, the TCAS module. The TCAS module is an aircraft collision avoidance system developed by the Federal Aviation Administration which has been used as case study in other related works [2], [11]. The TCAS module has twelve parameters; seven parameters have 2 values, two parameters have three values, one parameter has four values, and two parameters have 10 values.

As highlighted earlier, we chose the TCAS module because the same parameters and values have been used by other researchers. By adopting the same parameters and values, objective comparison may be made between various strategy implementations. To ensure that the results obtained are up-to-date given the fact that some of the implementations have evolved tremendously over the years, we downloaded all the available implementations within our environment to ensure

fair comparison. Here, we are also interested to investigate whether or not each strategy supports high t ($t > 6$).

Specifically, we downloaded ACTS (implementing IPOG, IPOG-D, IPOF1, and IPOF2) from NIST, ITCH, Jenny, TConfig, and TVGII. We were not able to download AETG because the implementation is a commercial product; therefore it was not considered for comparison in our study. To compensate the fact that that Jenny is an MSDOS-based executable program, we chose a running environment consisting of Windows XP 2.0 GHz, an Intel Core 2 Duo CPU, and 1 GB RAM with JDK 1.6 installed.

Tables 7 and 8 summarize the complete results. As in Table 3 NS indicates that the parameter and values chosen with a given strength are not supported. Also, darkened cell rows indicate the best performance in term of test size.

As seen in Table 7, MC-MIPOG, IPOG, IPOF1, and IPOF2 gave the optimum test size at $t = 2$. At $t = 3$, both MC-MIPOG and IPOG gave the optimum test size. For all other cases, MC-MIPOG always outperforms other strategies. Besides MC-MIPOG, only Jenny can support more than $t = 6$ for the TCAS module. However, we have not been successful in summoning Jenny for $t > 8$ because the program implementation crashes.

Even though TVG enables the user to select t from a range from 2 to 9, we cannot obtain any result for $t = 5$ because the program execution crashes. Note that our experiment with TVGII produced different results than the published results (here, we used the tool with the "T_ Reduced" option), perhaps due to a new update of the implementation.

Allowing the user to select t between 2 and 6, our experience indicates that for the TCAS module, TConfig merely gives a result for $t < 5$. Here, an exception occurs when we attempt to get a result for $t > 5$. A similar observation can be seen for ITCH. Note that ITCH does not support $t > 4$. Also, in the case of ITCH, the test size for $t = 3$ is greater than that for $t = 4$.

In comparison with all other IPOG variants (except MC-MIPOG), it is clear that IPOG outperformed IPOG_D, IPOF1, and IPOF2 in terms of test size for the TCAS module. Like other strategies (except MC-MIPOG), this family of strategies cannot produce a test suite for $t > 6$. That is, no option is given for $t > 6$.

In terms of execution time, IPOG-D has the fastest overall time for $t \leq 6$ (see Table 8). For $t > 6$, MC-MIPOG is fastest because no other strategies are able to provide t-way test generation support (Jenny supports up to $t = 8$). From one perspective, MIPOG is similar to IPOG and IPOG_D in the sense that they are all deterministic strategies. From another perspective, IPOF and IPOF2, are non-deterministic strategies. The general aim of IPOG_D, IPOF, and IPOF2 is to achieve a faster execution time than that of IPOG. Generally, obtaining an optimized test size and a fast execution time are two sides of the same coin. Obtaining an optimized test size requires more processing time for choosing the most optimized tuple. On the other hand, obtaining fast execution time means that little optimization is performed to obtain the optimum test size. This is evident as far as the test sizes are concerned for IPOG_D, IPOF, and IPOF2. MIPOG is a strategy that is designed to produce a smaller test size than that of IPOG under the cost of more processing time during horizontal extension. As discussed earlier, unlike IPOG, IPOG_D, and IPOF, MIPOG adopts a different type of vertical extension which is more heavyweight than that of IPOG (for optimization of vertical extension). For this reason, MIPOG's execution time tends to be slower than most of the IPOG variants. Nevertheless, the implementation of MC-MIPOG has alleviated this drawback through the adoption of a multicore architecture. In fact, MIPOG is the only strategy within the IPOG family that can be parallelized.

VI. Conclusion

As computer manufacturers make multicore CPUs pervasively available within reasonable costs, harnessing this technology is no longer a luxury but a viable and useful option.

In this paper, we investigated and evaluated a parallel strategy called MC-MIPOG for t-way test data generation on multicore architecture. Our results indicate that MC-MIPOG scales well against existing strategies. In preparation for our future work, we are currently porting MIPOG and MC-MIPOG into the grid environment. Our initial implementation results have been encouraging. We are also planning to perform more extensive comparisons with Colburn's best-known results.

Acknowledgement

We would like to acknowledge all the anonymous reviewers. We would also like to thank Jeff Lei, Rick Kuhn, and Raghu Kacker from NIST for making the ACTS tool available to us. This work is partly sponsored by generous grants–"Development of a Mobile Agent Based Parallel and Automated Java Testing Tool" from MOSTI. The first author, Mohammed Issam Younis, is a USM fellowship recipient.

References

[1] K.C. Tai and Y. Lei, "A Test Generating Strategy for Pairwise Testing," *IEEE Trans. Software Engineering*, vol. 28, no. 1, 2002, pp. 109-111.

[2] D.R. Kuhn and V. Okun, "Pseudo-Exhaustive Testing for Software," *Proc. 30th Annual IEEE/NASA Software Engineering Workshop*, Apr. 25-27, 2006, pp. 153-158.

[3] Y. Lei et al., "IPOG: A General Strategy for t-Way Software

Testing," *Proc. 14th Annual IEEE Intl. Conf. and Workshops on the Engineering of Computer-Based Systems*, Tucson, AZ, Mar. 26-29, 2007, pp. 549-556.

[4] M. Hutchins et al., "Experiments on the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria," *Proc. 16th Intl. Conf. Software Engineering*, May 16-21, 1994, pp. 191-200.

[5] R. Kuhn, Y. Lei, and R. Kacker, "Practical Combinatorial Testing: Beyond Pairwise," *IEEE IT Professional*, vol. 10, no. 3, 2008, pp. 19-23.

[6] M. Ellims, D. Ince, and M. Petre, "The Effectiveness of *t*-Way Test Data Generation," *Proc. 27th Int. Con. Computer Safety, Reliability, and Security*, Sept. 22-25, 2008, pp. 16-29.

[7] A. Williams, *TConfig Java Test Tool*. Available: http://www.site.uottawa.ca /~awilliam. Last accessed on April 30, 2009.

[8] B. Jenkins, *Jenny test tool*. http://www.burtleburtle.net./bob/math/jenny.html. Last accessed on May 7, 2009.

[9] J. Arshem, *TVG test tool*, http://sourceforge.net/projects/tvg/. Last accessed on May 7, 2009.

[10] A. Hartman, T. Klinger, and L. Raskin, *IBM Intelligent Test Case Handler*, http://www.alphaworks.ibm.com/tech/whitch. Last accessed on May 7, 2009.

[11] Y. Lei et al., "IPOG/IPOG-D: Efficient Test Generation for Multi-way Combinatorial Testing," *Software Testing, Verification and Reliability*, vol. 18, no. 3, 2007, pp. 125-148.

[12] M. Forbes et al., "Refining the In-Parameter-Order Strategy for Constructing Covering Arrays," *J. Research of the National Institute of Standards and Technology*, vol. 113, no. 5, 2008, pp. 287-297.

[13] R.N. Kacker et al., "Automated Combinatorial Testing for Software Systems," *Mathematical and Computational Sciences Division, NIST Report*, Jan. 2008, pp. 38-40.

[14] M.B. Cohen et al., "Constructing Test Suites for Interaction Testing," *Proc. 25th IEEE Int. Conf. Software Engineering*, May 3-10, 2003, pp. 38-48.

[15] A. Hartman and L. Raskin, "Problems and Algorithms for Covering Arrays," *Discrete Mathematics*, vol. 284, no. 1, 2004, pp. 149-156.

[16] M. Grindal, J. Offutt, and S.F. Andler, "Combination Testing Strategies: A Survey," *J. Software Testing, Verification, and Reliability*, vol. 5, no. 3, 2004, pp. 167-199.

[17] K.A. Bush, "Orthogonal Arrays of Index Unity," *Annals of Mathematical Statistics*, vol. 23, no. 3, 1952, pp. 426-434.

[18] D.R. Kuhn, D. Wallace, and A. Gallo, "Software Fault Interactions and Implications for Software Testing," *IEEE Trans. Software Engineering*, vol. 30, no. 6, 2004, pp. 418-421.

[19] A.W. Williams, "Determination of Test Configurations for Pair-Wise Interaction Coverage," *Proc. 13th Int. Conf. Testing Communicating Systems*, Aug. 29 -Sept. 1, 2000, pp. 59-74.

[20] M. I. Younis et al., "Assessing IRPS as an Efficient Pairwise Test Data Generation Strategy," *Int. J. Advanced Intelligence Paradigms*, vol. 2, no. 1, 2010, pp. 90-104.

[21] M.F.J. Klaib et al., "G2Way A Backtracking Strategy for Pairwise Test Data Generation," *Proc. 15th Asia-Pacific Software Engineering Conference,* vol. 3, no. 5, Dec. 03-05, 2008, pp. 463-470.

[22] M. Grindal, J. Offutt, and J. Mellin, "Conflict Management when Using Combination Strategies for Software Testing," *Proc. 18th Australian Software Engineering Conference*, Apr. 10-13, 2007, pp. 255-264.

[23] R.C. Bryce and C.J. Colbourn, "Prioritized Interaction Testing for Pairwise Coverage with Seeding and Constraints," *Information and Software Technology Journal*, vol. 48, no. 10, 2006, pp. 960-970.

[24] D.M. Cohen et al., "The Combinatorial Design Approach to Automatic Test Generation," *IEEE Software*, vol. 13, no. 5, 1996, pp. 83-88.

[25] Y. Lei and K.C. Tai, "In-Parameter-Order: A Test Generation Strategy for Pairwise Testing," *Proc. 3rd IEEE Int. Conf. High-Assurance Systems Engineering Symposium*, Nov. 13-14, 1998, pp. 254-261.

[26] D.M. Cohen et al., "The AETG System: An Approach to Testing Based on Combinatorial Design," *IEEE Trans. Software Engineering*, vol. 23, no. 7, 1997, pp. 437-444.

[27] R.C. Bryce and C.J. Colbourn, "The Density Algorithm for Pairwise Interaction Coverage," *J. Software Testing, Verification and Reliability*, vol. 17, no. 3, 2007, pp. 159-182.

[28] Y.W. Tung and W.S. Aldiwan, "Automating Test Case Generation for the New Generation Mission Software System," *Proc. IEEE Aerospace Conference*, Mar. 18-25, 2000, pp. 431-437.

[29] R.C. Bryce, C.J. Colbourn, and M.B. Cohen, "A Framework of Greedy Methods for Constructing Interaction Test Suites," *Proc. 27th IEEE Int. Conf. Software Engineering*, May 15-21, 2005, pp. 146-155.

[30] R.C. Bryce and C.J. Colbourn, "A Density-Based Greedy Algorithm for Higher Strength Covering Arrays," *Software Testing, Verification, and Reliability*, vol. 19, no. 1, 2009, pp. 37-53.

[31] M.A. Chateauneuf, C.J. Colbourn, and D.L. Kreher, "Covering Arrays of Strength Three," *Designs, Codes, and Cryptography*, vol. 16, no. 1, 1999, pp. 235-242.

[32] *Website for the NIST Automated Combinatorial Testing for Software (ACTS) project*: http://csrc.nist.gov/groups/SNS/acts/index.html. Last accessed on May 7, 2009.

[33] *Intel web site*: http://www.intel.com/products/desktop/processors/index.htm. Last accessed on May 7, 2009.

[34] *Intel Core 2 Quad Processors.* Available at http://www.intel.com/products/processor/core2quad/index.htm. Last accessed on May 7, 2009.

[35] K. Chow and D. Dagastine, "How to Get the Most Performance from Sun JVM on Intel Multicore Servers," *Sun Teach Day Developer Conference*, Oct. 23-25, 2007.

[36] C.J. Colbourn, *Covering Array Tables*, Available: http://www.public.asu.edu/~ccolbou/src/tabby/catable.html. Last accessed on July 12, 2009.

Mohammed I. Younis obtained his BSc in computer engineering from the University of Baghdad in 1997 and his MSc degree from the same university in 2001. He is a senior lecturer with the Computer Engineering Department, College of Engineering, University of Baghdad. He has been also a member of the Iraqi Union of Engineers since 1997. He is currently a PhD candidate and a USM fellowship recipient attached to the Software Engineering Research Group of the School of Electrical and Electronics Engineering, Universiti Sains, Malaysia. His research interests include software engineering, parallel and distributed computing, algorithm design, networking and security, cryptography, embedded systems, and RFID development.

Kamal Z. Zamli obtained his BSc in electrical engineering from WPI, USA, in 1992; his MSc degree in real-time software engineering from Universiti Teknologi, Malaysia in 2001; and his PhD in software engineering from University of Newcastle upon Tyne, UK, in 2003. He is currently attached to the Software Engineering Research Group of the School of Electrical and Electronics Engineering, Universiti Sains, Malaysia. His research interests include software engineering, software testing automation, parallel processing, and algorithm design.