Arab J Sci Eng (2017) 42:619–637
DOI 10.1007/s13369-016-2284-2

RESEARCH ARTICLE - COMPUTER ENGINEERING AND COMPUTER SCIENCE

An Automated Analysis of the Branch Coverage and Energy Consumption Using Concolic Testing

Sangharatna Godboley[1] · **Subhrakanta Panda**[1] · **Arpita Dutta**[1] · **Durga Prasad Mohapatra**[1]

Received: 22 November 2015 / Accepted: 17 August 2016 / Published online: 27 August 2016

Abstract The energy consumption of computer systems has become an important economic and environmental issue. Many researchers have focused on the energy consumption of hardware, but what about the software? Software energy consumption is widely adopted for Green computation of practical experimentation in research laboratories. But current researchers fail to build a consistent concept base for software energy consumption of critical applications. While branch coverage and concolic testing are very critical practices to validate the safety critical systems, very little effort is given to measure their energy consumption. The computation of the energy consumption of these techniques is an important issue in Green IT and Green Software Engineering. The contribution of this paper is to automate the computation and analysis of the energy consumption of the testing technique while enhancing the branch coverage using concolic testing. We implement our proposed automation framework in a tool, named *Green Analysis of Branch Coverage Enhancement*. The empirical study with forty Java programs and the evaluation results show that our developed tool achieves an average increase of 13.5 % in branch coverage. The average energy consumption of our automated tool is approximately 5.6 kJ to compute the branch coverage for all the forty experimental programs.

✉ Sangharatna Godboley
sanghu1790@gmail.com

Subhrakanta Panda
511cs109@nitrkl.ac.in

Arpita Dutta
215cs1067@nitrkl.ac.in

Durga Prasad Mohapatra
durga@nitrkl.ac.in

[1] Department of Computer Science and Engineering, National Institute of Technology, Rourkela, India

Keywords Branch coverage · Concolic testing · Energy consumption · Green IT · Green Software Engineering · Green and sustainable technologies

1 Introduction

A technical report in 2007 by ITC-Literacy estimated that Information and Communications Technologies (ICT) were responsible for more than 2 % of global carbon emissions [1]. PC Energy Report of USA sponsored by 1E in 2007 concluded that a company that consisted of 10,000 PCs approximately spent $165,000 in electricity consumption every year. This huge bill was incurred as computers were not switched off all night even when no one was working. Thus, the energy consumption of PCs is an important issue that needs to be addressed.

Green Information Technology (Green IT) believes that the resource and energy consumption of ICT starts right from the software development life cycle (SDLC) phase that should be reduced and minimized [2,3]. The available methods mainly focus on the hardware, such as the energy consumption of the data centers. Since most of the ICT solutions are based on software applications, the energy consumption of these applications is very crucial to analyze and control.

Software testing aims to detect and correct some bugs in a software to ensure its quality. Earlier, software testing techniques continuously failed to detect all the errors present in the software. One of the reasons for this failure is that it is not possible to exercise all the execution paths in a program. A program may have different paths because of the presence of conditional expressions and many other loop structures. It is technically a difficult task to generate test cases that could cover all execution paths in an aug-

mented manner [4]. To address these issues, CONCOLIC (CONCrete + symbOLIC) testing or dynamic symbolic execution analysis is proposed in the literature to generate test cases that explore the execution paths in a program [5–7]. The manual process of achieving this high coverage is tough, rather infeasible. In accordance with RTCA/DO178B standard [8], coverage-based testing is a measure of acceptability of the requirement-based testing in the context of executing logical statements. This is why branch coverage-based testing has become mandatory for safety critical systems such as aerospace and nuclear applications. Although CONCOLIC and branch coverage testing techniques are useful in many scenarios, their energy consumption has not yet been addressed. This is a core area of concern for Green IT and Green Software Engineering [9].

In this paper, we propose a framework that automates the calculation of the branch coverage and analyze its energy consumption to promote *Green software testing*. In this regard, we implemented our framework in a tool, called *Green Analysis of Branch Coverage Enhancement* (*Green-ABCE*) that integrates some of the available tools for different purposes. Our proposed framework integrates *Java Program Code Transformer* (*JPCT*) proposed in our earlier work [10] which is the Java version of Program Code Transformer (PCT) [11]. JPCT transforms the input Java programs to improve their branch coverage. We integrate Java Concolic Unit testing Engine (JCUTE) with our proposed framework to generate the required test cases and calculate the branch coverage percentage for both the transformed and non-transformed versions of the Java programs. Please note that though the transformed program is syntactically different as compared to original program, but both are semantical and functional equivalent to each other due to the empty body of extra expressions inserted. To avoid confusion, we strip out those extra nested if–else statements from the transformed program after the completion of experiment. By using JouleMeter, we compute the total energy consumption. We list the abbreviations used in this paper in Table 1.

The rest of the paper is organized as follows: Sect. 2 discusses some of the fundamental ideas required to understand the proposed approach. Section 3 explains the proposed framework and discusses the required algorithms for implementation. In Sect. 4, we analyze the experimental results. In Sect. 5, we discuss and compare some of the existing work related to our proposed work. There are some threats to the validity of our work, which we report in Sect. 6. Section 7 concludes our work with some insights into our future work.

2 Fundamental Ideas

In this section, we discuss some of the important concepts and definitions useful to our work.

Table 1 Abbreviations

Sl. no.	Short-form	Full-form
1	AC	Ammeter clamps
2	ATGT	ASM tests generation tool
3	BC	Branch coverage
4	BCT	Binary code transformer
5	CA	Coverage analyzer
6	CONBOL	CONcrete and symBOLIic testing
7	CONCOLIC	CONCrete +symbOLIC
8	CUTE	Concolic Unit Testing Engine
9	DAE	Data Aggregator and Evaluator
10	EC	Energy consumption
11	EDTSO	Energy-Directed Test Suite Optimization
12	EXNCT	Exclusive-NOR Code Transformer
13	GA	Genetic algorithm
14	Green-ABCE	Green Architecture model for Branch coverage enhancement
15	Green IT	Green Information Technology
16	Green SE	Green Software Engineering
17	ICTs	Information and Communications Technologies
18	JCUTE	Java Concolic Unit Testing Engine
19	JPCT	Java Program Code Transformer
20	LOC	Lines of code
21	MC/DC	Modified condition/decision coverage
22	PC	Power consumption
23	PCT	Program Code Transformer
24	PM	Power model
25	PSO	Particle swarm model
26	Power TOP	Power temporal optimization
27	TC	Test cases
28	SCORE	Scalable COncolic testing tool for Reliable Embedded software
29	SOP	Sum of products
30	UML	Unified Modeling Language
31	VIM	Virtual instrument machine
32	WG	Workload generator
33	WLS	Workloader simulator

2.1 Green Software Engineering

We derive the motivation for this proposed work from Kern et al. [9] to promote energy-efficient Green software testing. To classify and sort some concerns of Green and sustainable software and its engineering technology, Kern et al. [9] developed GREENSOFT model as shown in Fig. 1. GREENSOFT model consists of the following four parts: life cycle of software products, sustainability criteria for software products, procedure models, and recommendations for action and tools. Interested readers are advised to refer [9] for more

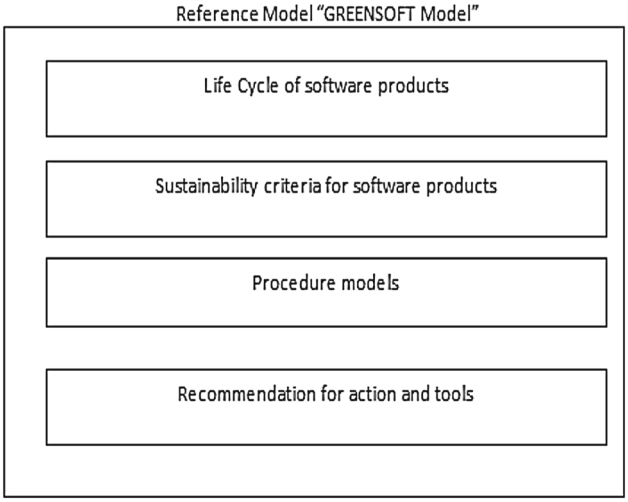


Fig. 1 GREENSOFT model

details on Green Software Engineering (Green SE). Thus with conformance to the principles of Green SE, we aimed to develop a tool that can compute the energy consumption during concolic testing. The awareness of energy consumption can later enable the testers to generate energy- efficient techniques in the direction of Green software testing.

2.2 Energy Consumption

This paper uses the existing JouleMeter[1] to compute the energy consumption. JouleMeter is integrated to compute the energy consumption (EC) of the proposed tool during concolic testing of the programs. JouleMeter gives the power consumption (PC) of the tool in use at any particular instance of the recorded timestamp. We use these power readings along with the first and last timestamps to compute the energy consumption in joules. The formula to convert the power readings to energy consumption is given below:

$$\mathrm{EC} = (T_m - T_n) \times \sum_{n}^{m} (PC_n) \tag{1}$$

where EC is the energy consumption (in Joules), PC is power consumption (in Watts), T_i is the time stamp for ith instant, n represents the first instance, and m represents the last instance.

2.3 Branch Coverage

For achieving branch coverage, each decision should take all possible outcomes at least once either true or false. For example: If $(m > 0)$, then there can be maximum of two possible test scenarios. These test scenarios are: (1) $m > 0$, and (2) $m <= 0$. Therefore, any testing technique that generates enough test cases to execute all the above-mentioned two scenarios achieves 100 % branch coverage.

[1] http://research.microsoft.com/en-us/projects/joulemeter/.

```
1  import java.io.*;
2  public class Grade {
3        public static void main(String[] args) throws IOException {
4            System.out.println("Enter The mark of the student");
5           BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
6           String ip = br.readLine();
7           int testscore = Integer.parseInt(ip);
8              char grade;
9
10             if (testscore >= 90) {
11                 grade = 'A';
12             } else if (testscore >= 80) {
13                 grade = 'B';
14             } else if (testscore >= 70) {
15                 grade = 'C';
16             } else if (testscore >= 60) {
17                 grade = 'D';
18             } else {
19                 grade = 'F';
20             }
21             System.out.println("Grade = " + grade);
22         }
23     }
```

Fig. 2 An example program to explain CONCOLIC Testing

2.4 CONCOLIC Testing

Concolic testing is a crossover methodology for program verification that consolidates *symbolic execution*. Concolic testing executes all the typical program variables for a more *concrete execution* that runs the program on some specific inputs. In concolic testing, we introduce some standard variables (with random values) to generate new path conditions along with the regular execution paths (executed when the condition satisfies to *true*) to explore alternate branch predicates. These newly generated paths connect the initial predicate with that of its negated predicates.

The program in Fig. 2 checks the mark secured by a student and accordingly assigns a grade. For concolic testing of this program, we start by assigning random inputs to the variable, named *testscore*. The execution of the program depends on the assigned value of *testscore* variable. Supposing *testscore* gets 91 as its first value, then the first predicate to be analyzed will be:

$$(testscore >= 90) \tag{2}$$

Now, Condition 2 must fail to explore another path possible to reach the next predicate. The new path is discovered by negating the predicate under consideration. This negation operation is shown below:

$$\neg(testscore >= 90) \tag{3}$$

The new predicate reached after the negation operation is as follows:

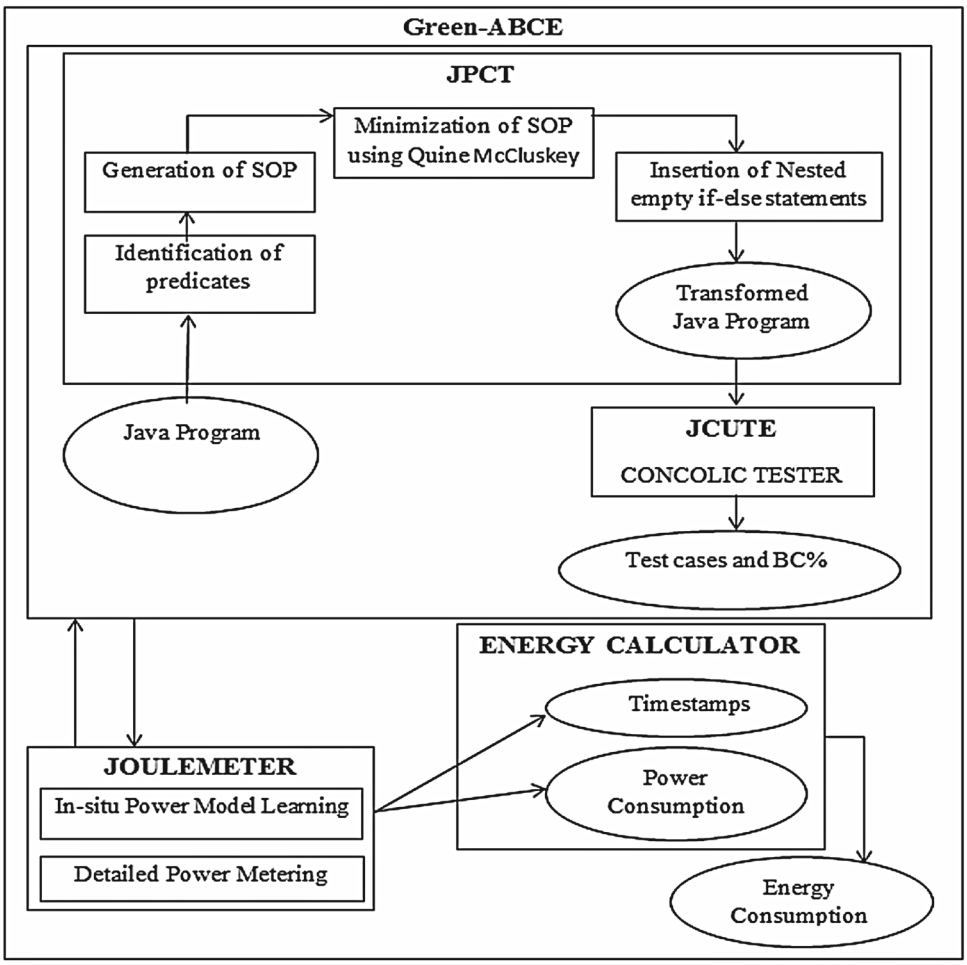


Fig. 3 Overall view of Green-ABCE

$$(testscore >= 80) \tag{4}$$

The *constraint solver* examines every constraint to discover the appropriate paths in the program. If there exists a path for an input, then that value is recorded as test data for further executions of the program. The original constraint is combined with the negated constraint to reach the next new path constraint as shown in Eq. 5.

$$\neg(testscore >= 90) \wedge (testscore > 80) \tag{5}$$

The above process will continue till the conditions get evaluated.

3 Proposed Framework

Both coverage-based testing and energy consumption analysis are necessary for many safety critical systems. Due to less branch coverage, many software testers are compelled to discard the software. The proposed framework aims to study the increase in branch coverage achieved by employing the available techniques. But this enhancement in branch coverage comes with an additional energy consumption. To measure the energy consumption incurred in achieving an increase in branch coverage, an energy computational tool, i.e., JouleMeter, is integrated into the proposed framework along with the branch coverage analyzer. Thus, our framework is implemented in a tool, named *Green-ABCE*. The objective of this tool is to enhance the branch coverage in concolic testing and compute the energy consumption. We have developed JPCT to achieve an increase in the branch coverage and integrated it in Green-ABCE. We used JCUTE as dynamic symbolic execution tool that resolves issues such as unavailability of library code and shortcomings of concolic engines.

3.1 Overview of Green-ABCE

Figure 3 shows the overall view of Green-ABCE. The block diagram consists of mainly three modules: *JPCT*, *JCUTE* and *JouleMeter*. The flow starts with the initiation of tracking record of power consumption for each 1 s of JPCT and JCUTE by JouleMeter. JPCT takes the Java program as

Table 2 Algorithm 1 for Green-ABCE

Input: ABCE module ▷ *ABCE modules consists of JPCT and JCUTE modules*
Output: Total time taken(T), Power Consumption (PC), Energy Consumption (EC)
Begin
1: Compute Power Consumption. ▷ Through JouleMeter
2: Record power readings.
3: Generate TCs_1 and compute $BC\%$ using JCUTE. ▷ *Without using JPCT*
4: Produce Transformed Program. ▷ *Invoke algorithm JPCT*
5: Generate TCs_2 and compute $BC'\%$ using JCUTE for transformed program. ▷ *With JPCT*
6: Measure Execution Time, Speed of test case generation, Difference = $BC'\%$ - $BC\%$. ▷ *To achieve high branch coverage*
7: Stop recording power readings.
8: Stop JouleMeter.
9: Compute Total time taken ($T_m - T_n$), and read Total power consumption (PC).
10: Evaluate Energy consumption using Equation 1.
11: Exit

input and transforms it by adding extra nested empty if–else expressions. Then the transformed version of the program is supplied to JCUTE to generate test cases automatically [12] and compute the achieved percentage of branch coverage. After the completion of this process, we stop the tracking of the power consumption. Finally, we get the total time taken by Green-ABCE to find the branch coverage percentage, total power and energy consumption of JPCT and JCUTE models.

3.2 Detailed Description of Green-ABCE

In this section, we provide a detailed discussion on the working of the different components of Green-ABCE.

3.2.1 Green-ABCE

The overall pseudocode for the proposed work is given in Algorithm Table 2. The proposed Algorithm Table 2 takes the ABCE module as input which consists of a Java program, JPCT to transform the Java program and JCUTE concolic tester to generate the required test cases and compute the branch coverage. Here, JPCT stands for Java Program Code Transformer, whereas JCUTE stands for Java Concolic Unit Testing Engine. More description on JPCT and JCUTE is given in Sects. 3.2.2 and 3.2.3, respectively. Finally, Algorithm Table 2 gives the total *Energy Consumption* (*EC*) for the input program as output. The first two steps are performed by JouleMeter to start the timestamp recording. Step 3 of the algorithm computes the required other parameters (i.e., generation of test cases and computing their corresponding branch coverages) for the input Java program by using JCUTE only. Step 4 invokes the code transformer, named JPCT, that transforms the input Java program. The pseudocode for the code transformer is given as Algorithm Table 3. Now, Step 5 finds the test cases and BC% (branch coverage using JCUTE + JPCT) for the transformed version of the Java program. Step 6 computes all the required parameters, i.e., computation time for branch coverage, speed of test case generation and difference. Steps 7 and 8 are used to stop functioning of JouleMeter. Step 9 computes the total power consumption in terms of timestamp difference. Finally, Step 10 calculates the energy consumption using the formula given in Eq. 1.

3.2.2 JPCT

The purpose of Java Program Code Transformer (JPCT) [10,11] is to transform the input Java program by inserting additional *if–else* statements. The pseudocode of the transformer is given in Algorithm Table 3. The steps for transforming the program are as follows:

i. *Identification of predicates* The objective of this step (Lines 1–3) is to detect the predicates in a Java program. The logic is to scan the Java program character by character and copy the line of code having a Boolean operator in some other text file. This process is repeated until all the predicates in the program are discovered.
ii. *Generation of Sum of Products (SOP)* The objective of this step (Lines 5–6) is to have a simplified representation of the identified predicates by computing their *Sum of Product* (*SOP*). This simplification is performed for every predicate discovered in the program. The SOP for a predicate is converted into minterm (binary) form so that it can be minimized using Quine–McCluskey method. These minterms are stored in the variable, $P\_Minterm$.
iii. *Minimization of the SOP using Quine–McCluskey method* Line 7 uses the Quine–McCluskey method to minimize the SOP. The Quine–McCluskey method is given in [10,13]. The hypothesis behind choosing Quine–McCluskey method over another existing technique, called Karnaugh map, to minimize the predicates is simple because the former is more efficient than the latter [13].

Table 3 Algorithm 2 for JPCT: Java Program Code Transformer

Input: J ▷ *Program J is in Java syntax*
Output: J′ ▷ *Program J′ is the transformed version of J*
Begin
1: **for** each statement s ∈ J **do**
2: **if** && or ||or unary ! detects in s **then**
3: Predicate_Identified ← Save_List(s) ▷ *Save_List(s) keeps record of all identified predicates.*
4: **for** each predicate p ∈ Predicate_Identified **do**
5: P_SOP ← SOP(p) ▷ *Generating SOP*
6: P_Minterm ← Minterm(P_SOP)
7: P_Minimized ← Minimization_QM(P_Minterm)
8: List_Statement ← Additional_if-else_for_JPCT(P_Minimized)
9: J′ ← insert_code(List_Statement,J) ▷ *The insert_code() gathers the extra generated statements and input Java program.*
10: Exit

iv. *Insertion of nested empty if–else statements* Line 8 generates the empty nested if–else statements to get additional set of statements in the program. The method to generate these additional statements is defined in [10]. These additional sets of statements comprise of empty *nested if–else* statements which are inserted into the input program at Line 9 resulting in the transformed version of the original Java program. These additional conditional expressions are useful to increase the number of generated test cases. Once we enhanced the number of test cases, we can cover more number of branches. This results in achieving high branch coverage.

3.2.3 JCUTE

We use JCUTE to generate test cases automatically for both original and transformed versions of the Java program. The output of Algorithm Table 3 is now the input for JCUTE. Then JCUTE computes the branch coverage percentage. We also record the total time taken by JCUTE to compute the branch coverage. More information on JCUTE tool is available in [5,6].

3.2.4 JouleMeter

JolueMeter is a tool to measure the power consumption of a computer system. Power consumption data can be computed for the system as a whole or for the key components. Readings may depend on individual tracking of different applications. JouleMeter computes the power usage through a model, called *power model*. This model shows the usage of computer resources and hardware power state (processor utilization, screen brightness, process frequency, monitor on/off state, utilization of memory) to draw the power value. But in our proposed work, we are not considering other hardware power state usages, we are only tracking the power usage of a particular software application. In the context of our research work, the software application refers to our proposed software testing tool. To learn power model, a manual estimate of the individual power numbers may be entered to view the power usage data. JouleMeter suggests some power number parameters for the power model. These power number parameters are: *Base (Idle) Power*, *Processor Peak Power (high frequency)*, *Processor Peak Power (low frequency)* and *Monitor Power*. Please note that the power consumption of an application program denotes only the power consumed by the application on the CPU. Here, we may consider the *monitor power* as well as the *base power* of the computer, only if the computer has been turned on to run that application only. Energy consumption may be defined as the total energy used for the entire run which may be obtained by summing up the power consumption values (present in Watts) for the duration of execution. Since each reading denotes the power usage for one second, the sum represents the total energy consumption in Joules. This whole process is represented in the form of a pseudocode given in Algorithm Table 2.

4 Experimental Studies

The personal computer (PC) is configured of 4GB memory (RAM), Intel (R) Core (TM) i5 CPU 650@3.20 GHz 3.19 GHz and 32-bit operating system. We have performed experiments on Windows 7 operating system. Using our proposed work, we can handle any of the languages such as SQL and C++. Since power consumption tracker, i.e., JouleMeter, is used to measure and record power consumption of any application, we can track power consumption of any tool which is developed in any language. Again, energy consumption calculator is also a developed tool which executes the application/tool irrespective of any language. Our

objective is to compute the energy consumption of our proposed improved concolic testing. This testing technique uses JPCT and JCUTE; therefore, we use Java language. Our objective is to compute energy consumption of our proposed improved concolic testing. This testing technique uses JPCT and JCUTE; therefore, we use Java language. The experiments are carried out on forty benchmark Java programs taken from the *Open Systems Laboratory Repository*[2] and some student assignments.

Table 4 lists the characteristics of Java programs that we have considered for the experimental studies. From Columns 3 and 4 of Table 4, it is concluded that a total of 6064 lines of code of the original programs have been analyzed. Similarly, 9601 lines of code have been analyzed for the transformed programs. While lines of code (LOC) gives the size of original program, LOC′ refers to the size of the transformed version of the program. Since we focus on coverage-based testing, it is essential to count the total number of predicates and branches that are supposed to be covered. Both these characteristics are mentioned in Columns 7 and 8, respectively.

Tables 5 and 6 show the required test data [such as *number of generated test cases* (TC, TC′), *computation time* (Time, Time′) and the *speed of generating the test cases* (Speed, Speed′)] computed for the experimental Java programs. These test data are computed under two scenarios: Scenario 1 (Table 5) corresponds to the test data generated for the original programs using JCUTE, while Scenario 2 (Table 6) corresponds to the test data computed for the transformed programs using JCUTE + JPCT. It can be inferred from the tables that by integrating JPCT, we achieved an increase in the generation of the test cases. This increase in the number of test cases helps in covering more number of branches. Thus, the program transformation technique using JPCT results in high *branch coverage percentage* due to increase in the number of generated test cases. The computation time (to generate test cases and compute branch coverage) taken in case of both the scenarios is shown as Time and Time′. In case of Scenario 1, the average computation time for the forty programs is 23,678.93 ms, while in Scenario 2, the average computation time for the forty programs is 27,176.42 ms. Thus, the speed of test case generation is given as the number of test cases generated per second (TC/s). The average speed computed for Scenario 1 is speed = 4.66 TC/s, and average speed for Scenario 2 is speed′ = 2.27 TC/s for the forty programs. This decrease in test case generation speed is attributed to the increase in time for program transformation. However, this decrease in speed is not a deterring factor as it results in the increase in the number of generated test cases and branch coverage.

[2] http://osl.cs.illinois.edu/software/jcute/.

Table 4 Characteristics of different experimental programs

Sl. no.	Program name	LOC	LOC′	# of Predicate	# of Branches
1	Condition	21	32	2	15
2	Weight	24	42	1	20
3	QuickSort	68	76	2	18
4	Nonce	286	350	25	146
5	StringBuffer	421	466	17	56
6	SwitchTest	59	71	4	15
7	ProducerConsumer	20	27	2	8
8	BSTree	255	297	6	28
9	ArraySort	27	33	2	15
10	StaticInstanceProblem	24	32	1	5
11	CAssume	43	65	1	2
12	CTest3	94	117	2	4
13	Deadlock	91	125	1	2
14	Demo	50	72	1	2
15	DemoLock	67	95	1	2
16	CTest2	96	114	2	4
17	CTest1	96	126	2	2
18	DemoL0ck2	60	971	1	2
19	DSort	95	121	4	18
20	ErrorTest	71	98	4	9
21	If–Else	77	118	1	6
22	InterfaceJava	51	350	1	2
23	LockHeld	73	97	4	12
24	MultiLock	62	106	2	4
25	NewJava	61	118	1	4
26	NoPredictive	69	108	3	8
27	NoPredictive2	58	84	2	4
28	NS	313	715	24	146
29	NS2	403	806	28	188
30	FinallyTest	45	45	0	0
31	OneWrite	45	45	0	3
32	Regression	176	256	15	8
33	SampleXACMLPolicy	90	125	1	24
34	StackTest	33	47	1	0
35	StaticInstanceProblem2	51	68	1	4
36	StringBufferModule	1364	1896	9	52
37	StringBuffer1	486	976	8	60
38	StringBuffer2	541	1012	10	64
39	Struct	47	95	2	8
40	Testme	51	78	1	4

The analysis of the branch coverage percentage is shown in Table 7. Here, BC% corresponds to scenario 1 and BC′% corresponds to scenario 2. The values of BC range from 0 to 100 %. Please note that BC% for some programs (such as *CTest1, ErrorTest, FinallyTest, OneWrite and StackTest*) is 0 %; this is because none of the branches of these pro-

Table 5 Generated test data for the original Java programs using JCUTE

Sl. no.	Program name	TC	Time in ms	Speed
1	Condition	4	1047	3.8
2	Weight	5	1295	3.8
3	QuickSort	1	305	3.2
4	Nonce	24	13,752	1.7
5	StringBuffer	8	128,124	0.07
6	SwitchTest	6	17,004	0.35
7	ProducerConsumer	1	171	5.18
8	BSTree	6	2637	2.2
9	ArraySort	4	2637	1.5
10	StaticInstanceProblem	3	530	5.6
11	CAssume	2	359	5.5
12	CTest3	2	125	16.0
13	Deadlock	1	344	2.9
14	Demo	1	156	6.4
15	DemoLock	1	249	4.01
16	CTest2	2	140	14.28
17	CTest1	1	145	6.89
18	DemoL0ck2	3	499	6.01
19	DSort	5	3408	1.46
20	ErrorTest	1	156	6.41
21	If–Else	4	562	7.11
22	InterfaceJava	2	328	6.09
23	LockHeld	1	187	5.34
24	MultiLock	1	260	3.84
25	NewJava	1	203	4.92
26	NoPredictive	2	646	3.09
27	NoPredictive2	4	572	6.99
28	NS	43	202, 155	0.21
29	NS2	30	205, 284	0.14
30	FinallyTest	1	142	7.04
31	OneWrite	1	3557	0.28
32	Regression	5	320	15.62
33	SampleXACMLPolicy	11	2211	4.97
34	StackTest	1	170	5.88
35	StaticInstanceProblem2	3	481	6.23
36	StringBufferModule	6	206, 291	0.02
37	StringBuffer1	8	74, 796	0.1
38	StringBuffer2	8	74, 427	0.1
39	Struct	5	983	5.08
40	Testme	3	499	6.01

Table 6 Generated test data for the transformed Java programs using (JPCT + JCUTE)

Sl. no.	Program name	TC′	Time′ in ms	Speed′
1	Condition	5	2338	2.13
2	Weight	6	4355	1.37
3	QuickSort	1	1128	0.88
4	Nonce	24	27, 315	0.87
5	StringBuffer	8	160, 428	0.04
6	SwitchTest	7	18, 748	0.32
7	ProducerConsumer	2	639	3.12
8	BSTree	8	3296	2.42
9	ArraySort	5	3275	1.22
10	StaticInstanceProblem	3	660	4.54
11	CAssume	2	1541	1.29
12	CTest3	3	1012	2.96
13	Deadlock	2	2665	0.75
14	Demo	3	1873	1.6
15	DemoLock	3	1181	2.54
16	CTest2	5	988	5.06
17	CTest1	2	1028	1.94
18	DemoLock2	7	1490	4.69
19	DSort	12	4480	1.11
20	ErrorTest	2	256	7.81
21	If–Else	6	2606	2.30
22	InterfaceJava	3	1234	2.43
23	LockHeld	2	1274	1.56
24	MultiLock	1	1714	1.75
25	NewJava	1	2297	0.43
26	No Predictive	5	2814	1.77
27	NoPredictive2	6	2343	2.56
28	NS	53	211, 542	0.20
29	NS2	47	215, 434	0.13
30	FinallyTest	1	187	5.34
31	OneWrite	1	3750	0.26
32	Regression	12	2088	5.74
33	SampleXACMLPolicy	19	3858	4.92
34	StackTest	2	496	4.03
35	StaticInstanceProblem2	4	780	3.8
36	StringBufferModule	11	232, 818	0.02
37	StringBuffer1	17	76, 649	0.22
38	StringBuffer2	19	82, 754	0.22
39	Struct	7	2301	3.04
40	Testme	5	1424	3.51

grams are covered by the generated test cases. This clearly indicates the inefficiency of the test cases generated in scenario 1. However, after program transformation, we are able to successfully enhance the branch coverage for the above-mentioned programs. For example, BC′ for program *OneWrite* (refer Column 4) improved from 0 to 66 % approximately. Please observe that the BC% value is 100 % only for eleven programs, while BC′% is 100 % for fifteen programs. It means that all possible branches are covered. The last column in Table 7 gives the difference of these two branch

coverage percentages. It can be seen that there is no change in the branch coverage percentage for some programs. The possible reasons for this non-enhancement in the branch coverage of these programs could be due to the fact that either all the possible branches are already covered, or the number of branches covered is same for both the scenarios. The remaining programs successfully achieved an enhancement in the branch coverage percentage. The average BC% is 66.5 % and BC′% is 80 % for the forty programs. Here, we conclude that the average increase in the percentage of branch coverage is approximately 13.5 % for the experimental programs. Figure 4 shows the branch coverage percentages for both the scenarios. X-axis represents the program number and Y-axis represents BC%.

To focus on the energy awareness, Tables 8 and 9 show the energy consumption details for Scenarios 1 and 2, respectively. Columns 3 to 5 of Table 9 show the recorded timestamps. Column 6 shows the power consumption of our tool while computing the test data for each experimental program. Figure 5 shows the power consumption for different experimental programs with respect to time (refer Column 5 of Table 9). Table 9 shows that the power consumption of our tool for 7.5 % of the experimental programs crossed 100 W. While the power consumption is below 50 W for 87.5 % of the programs, only 5 % of the programs consumed power within 50 to 100 W. We show the box-plot of power consumption with respect to the lines of code (LOC) of the transformed program, number of predicates and total time of power consumption. The result in Fig. 6 shows that the power consumption does not increase appreciably with the increase in the LOC (i.e., LOC′) due to code transformation. Hence, code transformation by JPCT to enhance the branch coverage will not incur much power consumption. Thus, JPCT is power efficient, whereas a small increase in the number of predicates in the program shows a steep rise in the power consumption (refer Fig. 7). Thus, more is the number of predicates in the programs, more is the power consumption. As the tool spends some time to cover more predicates, the power consumption increases. This increase in the power consumption as a function of the time spent by the tool in analyzing the programs is evident from Fig. 9.

The energy consumption computed using Eq. 1 is shown in the last columns of Tables 8 and 9 for all the forty programs. The range of energy consumption varies from 0.1 to 70 kJ. Figure 8 shows the programs on X-axis and their corresponding energy consumption on Y-axis. The results show that the energy consumption crossed 1 kJ for 25 % of programs. The average energy consumption of the tool without code transformer is 5.2 kJ and with code transformer is 5.6 kJ. Hence, the increase in energy consumption due to code transformation is 0.4 kJ. This increase in energy consumption with respect to the increase in computational time (Tables 5, 6) is shown in Fig. 10. Thus, with reference to the principles of Green software engineering, the energy overhead incurred to achieve 13.5 % of increase in branch coverage is only 0.4 kJ. Therefore, the testers can opt for code transformation and still support Green software testing.

Table 7 Computed branch coverage percentage

Sl. no.	Program name	BC%	BC′%	Difference %
1	Condition	40 %	93 %	53 %
2	Weight	40 %	75 %	35 %
3	QuickSort	83.33 %	83.33 %	0 %
4	Nonce	60.95 %	72.60 %	11.65 %
5	StringBuffer	58.92 %	60 %	1.08 %
6	SwitchTest	80 %	86.66 %	6.66 %
7	ProducerConsumer	62.50 %	75 %	12.5 %
8	BSTree	67.85 %	82.14 %	14.29 %
9	ArraySort	60 %	73.33 %	13.33 %
10	StaticInstanceProblem	60 %	80 %	20 %
11	CAssume	100 %	100 %	0 %
12	CTest3	75 %	100 %	25 %
13	Deadlock	100 %	100 %	0 %
14	Demo	50 %	50 %	0 %
15	DemoLock	50 %	100 %	50 %
16	CTest2	75 %	100 %	25 %
17	CTest1	0 %	50 %	50 %
18	DemoLock2	100 %	100 %	0 %
19	DSort	100 %	100 %	0 %
20	ErrorTest	0 %	0 %	0 %
21	If–Else	100 %	100 %	0 %
22	InterfaceJava	100 %	100 %	0 %
23	LockHeld	91.66 %	91.66 %	0 %
24	MultiLock	100 %	100 %	0 %
25	NewJava	50 %	75 %	25 %
26	NoPredictive	62.5 %	87.5 %	25 %
27	NoPredictive2	75 %	100 %	25 %
28	NS	82.87 %	90.41 %	7.54 %
29	NS2	65.42 %	82.97 %	17.55 %
30	FinallyTest	0 %	0 %	0 %
31	OneWrite	0 %	66.66 %	66.66 %
32	Regression	100 %	100 %	0 %
33	SampleXACMLPolicy	95.83 %	95.83 %	0 %
34	StackTest	0 %	0 %	0 %
35	StaticInstanceProblem2	100 %	100 %	0 %
36	StringBufferModule	51.92 %	67.3 %	15.38 %
37	StringBuffer1	60 %	80 %	20 %
38	StringBuffer2	60.9 %	81.25 %	20.35 %
39	Struct	100 %	100 %	0 %
40	Testme	100 %	100 %	0 %

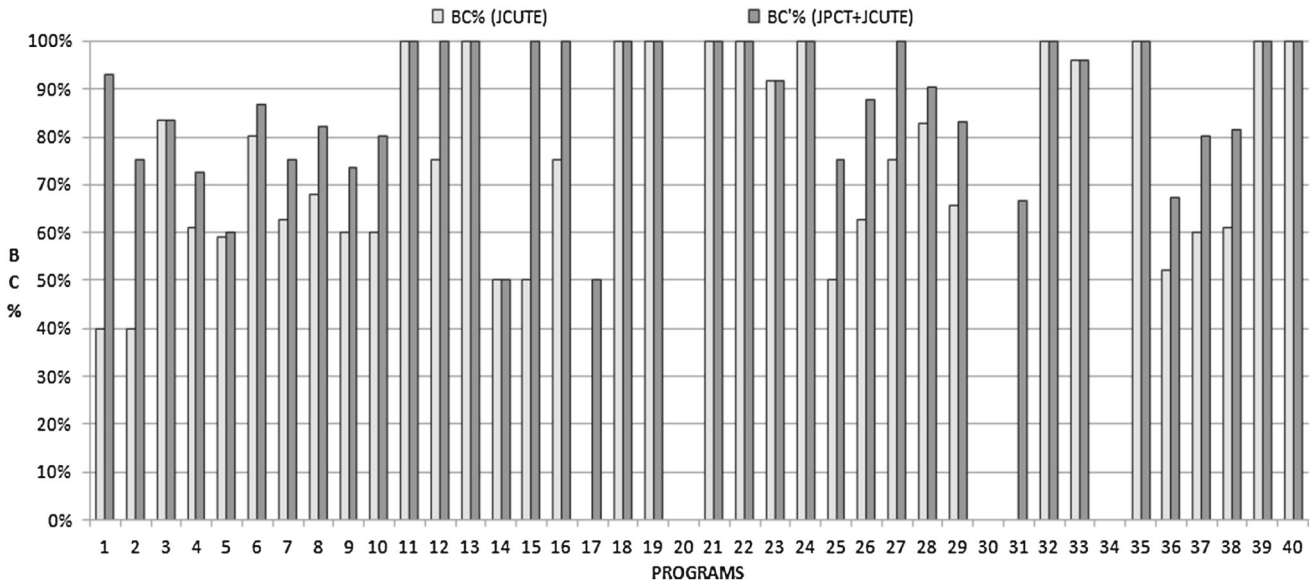


Fig. 4 Comparison of branch coverages

5 Comparison with Related Work

In this section, we discuss and compare some of the existing works that are related to our proposed work.

Li et al. [2] proposed a framework called *Energy- Directed Test Suite Optimization (EDTSO)*. They have minimized the test cases by considering their energy efficiency. They have presented a new test suite minimization technique for generating test suite with reduced energy consumption. Their approach is based on casting the energy and test coverage requirements as an ILP problem. They have performed their experiment using two applications, i.e., K-9 Mail and MyTracks. For K-9 Mail, the most expensive test case consumes almost 5000 mJ, the second almost 3,000 mJ , with the remainder ranging from approximately 1000 mJ to just under 5 mJ. Similarly, energy consumption of the test cases for MyTracks ranged from ≈5 mJ to ≈400,000 mJ. From this result, they conclude that test cases from a test suite are likely to have a high amount of variance in their energy consumption. Unlike the approach in [2], our proposed approach computes the energy consumption of concolic testing tool instead of computing the energy efficiency of individual test cases, which is currently not supported by the tool.

Amsel et al. [1] proposed *GreenTracker* for estimating the energy consumption of any software. They consider the development of this tool to be critical for establishing which software systems are the most environmentally sustainable. They also aim to spread awareness about the potential environmental hazards associated with software and to improve software engineering techniques to reduce the energy consumption of software. On their results and evaluation, they have experimented for Mozilla Firefox 3.0.10 and Safari 4.0.3 Internet browsers. They found that, on their system, the average CPU power of Firefox was lower that Safari. The average CPU of Firefox over 5 trials was 28.63 % while Safari was 30.98 %. GreenTracker is not suitable for the purpose of testing, whereas our proposed tool focuses on enhancing the branch coverage for effective concolic testing and shows the energy consumed in this process.

Dick et al. [14] presented a method to measure and rate software-induced energy consumption of stand-alone applications on desktop computers as well as interactive transaction-based application on servers. In both cases, realistic workloads are applied. Their test rig contains the tested system, an appropriate power meter (PM), a workload generator (WG) and a data evaluation system (DES). Their method is intended to support software developers, purchasers, administrators and users in making informed decisions on software architecture and implementation as well as on software products they use or plan to use. For their experiment, they have considered Firefox and Internet Explorer browsers. The difference was significant $t(30.686) = -10.981$, $p < 0.01$ (equal variances not assumed). In this scenario, the average energy conservation of Firefox compared to Internet Explorer is about 19 %, whereas our proposed approach is to compute power consumption and energy consumption of a software testing tool.

Chen et al. [15] developed a tool, named *StreesCloud*, to measure energy consumption of applications. They have observed that the energy consumption with deployment configuration "3Small(S)" increased by 0.8 % compared to "3Small(D)." The system throughput of "3Small(S)" decreased by 2.1 % compared to "3Small(D)." This is because in the client workload they have modeled in this

Table 8 Power consumption and Energy consumption of proposed tool with JCUTE

Sl. no.	Program name	Total time (ms)	Power (Watt)	Energy (Joules)
1	Condition	80,456	13.4	1078.1104
2	Weight	63,422	17.5	1109.885
3	QuickSort	133,545	10.5	1402.2225
4	Nonce	80, 452	34.5	2775.594
5	StringBuffer	49,789	14.9	741.8561
6	SwitchTest	125,871	32.5	4090.8075
7	ProducerConsumer	44,895	5.2	233.454
8	BSTree	33,754	10.4	351.0416
9	ArraySort	44,645	10.5	468.7725
10	StaticInstanceProblem	39,472	8.9	351.3008
11	CAssume	27,457	7.4	203.1818
12	CTest3	32,625	8.9	290.3625
13	Deadlock	38,589	4.5	173.6505
14	Demo	36,612	7.8	285.5736
15	DemoLock	31,278	4.2	131.3676
16	CTest2	37,895	6.5	246.3175
17	CTest1	27,954	4.5	125.793
18	DemoL0ck2	32,586	4.2	136.8612
19	DSort	25,689	8.9	228.6321
20	ErrorTest	31,375	4.4	138.05
21	If–Else	41,689	9.5	396.04
22	InterfaceJava	47,892	8.6	411.8712
23	LockHeld	43,654	5.3	231.3662
24	MultiLock	25,436	12.8	325.5808
25	NewJava	32,487	8.7	282.6369
26	NoPredictive	35412	7.8	276.2136
27	NoPredictive2	36,478	8.6	313.7108
28	NS	221,456	222.6	49,296.11
29	NS2	235,645	234.3	55,211.62
30	FinallyTest	32,456	8.5	275.876
31	OneWrite	38,724	12.6	487.9224
32	Regression	38,895	6.2	241.149
33	SampleXACMLPolicy	48,542	15.45	749.9739
34	StackTest	25,458	4.8	112.5964
35	StaticInstanceProblem2	30,578	9.2	281.3176
36	StringBufferModule	284,520	234.8	866,805.3
37	StringBuffer1	99,415	80.9	8042.679
38	StringBuffer2	100,325	80.2	8046.065
39	Struct	28,456	7.6	216.2656
40	Testme	25,789	7.9	203.7331

test, the majority of the tasks were communication-intensive. Their experimental results demonstrated that, with the support of StressCloud, the performance and energy consumption of cloud applications in realistic cloud environments can be collected and analyzed in an effective and efficient manner.

Unlike the tool *StreesCloud* that measured the performance of cloud base application, our tool measures the energy consumption for simple stand-alone Java applications.

Capra et al. [16] identified an approach for measuring application software energy efficiency, focusing on management information systems (MIS). Their approach was a first step toward defining application software energy efficiency and providing developers with guidelines for designing energy-efficient code. They have also developed a Java tool, called the workload simulator (WLS). For a given application, their tool can simulate a flow of operations and execute it a certain number of times for a given number of simultaneous users, thus generating a benchmark workload. They have measured power consumption using ammeter clamps (AC) and a system that they called the virtual instrument machine (VIM). Their experiments reveal that application software has a non-negligible impact on energy consumption. The power consumed by the server running the applications was up to 72 % higher than the power the idle server absorbed. Furthermore, different applications within the same category required significantly different amounts of energy to perform the same set of operations. For example, *OpenBravo* requires 17.5 W h, while *Adempiere* requires only 7.1 W h on Windows, with a gap greater than 100 %. These results are consistent across application categories and operating systems, with an average gap of 79 %. In our proposed work, we also developed our tool using Java language. Our tool is able to perform testing task for Java programs. Our proposed work results in total power and energy consumptions.

Brown et al. [17] discussed in their article an overall approach to energy efficiency in computing systems. They have proposed the implementation of energy optimization mechanisms within systems software, equipped with a power model for the system's hardware and informed by applications that suggest resource-provisioning adjustments so that they can achieve their required throughput levels and/or completion deadlines. For example, a current four-socket server system (based on the eight-core Sun Niagara2 CPU) with 16 DIMMs per socket using DDR2 dual-channel memory technology has 64 DIMMs total. This would increase to 24 DIMMs per socket (96 total) if its faster successor used DDR3 triple-channel memory instead. A representative DDR2 DIMM consumes 1.65 W (or 3.3 W per pair), whereas the lowest power edition of the current DDR3 DIMMs consumes 1.3 W (or 3.9 W per trio). The result appears to be an increase of only 20 % power consumption from about 100–120 W total in their example. In our proposed work, we concentrate on computation power and energy Consumptions of Software Applications instead of hardware components.

Saxe et al. [18] proposed and developed *PowerTop* tool. *PowerTop* shows the extent of the waste, while also showing which software is responsible. System users can then report the observed waste as bugs and/or elect to run more efficient

Table 9 Power consumption and Energy consumption of proposed tool with JPCT+JCUTE

Sl. no.	Program name	Timestamps			Application	Energy
		From	To	Total time (ms)	Power (Watt)	Consumption (Joules)
1	Condition	63569182997999	63569183093347	95,348	12.3	1172.7804
2	Weight	63569188654831	63569188730053	75,222	16.3	1226.1186
3	QuickSort	63569189018202	63569189170647	152,445	11.1	1692.1395
4	Nonce	63569190679184	63569190760530	81,146	36.4	2953.7144
5	StringBuffer	63569191038982	63569191090740	51,758	15.3	791.8974
6	SwitchTest	63569194744377	63569194871158	126,781	33.9	4297.8759
7	ProducerConsumer	63569195206170	63569195253022	46,852	6.1	285.7972
8	BSTree	63569195628512	63569195662995	34,483	11.7	403.4511
9	ArraySort	63569196301754	63569196347411	45,657	11.4	520.4898
10	StaticInstanceProblem	63569196675389	63569196715962	40,573	10.7	434.1311
11	CAssume	63569209614454	63569209644924	30470	7.1	216.3370
12	CTest3	63569216007321	63569210052945	45624	7.2	328.4928
13	Deadlock	63569210305932	63569210348524	42592	9.2	391.8464
14	Demo	63569210593664	63569210636276	42612	6.9	294.0228
15	DemoLock	63569211268204	63569211301679	33475	4.4	147.2900
16	CTest2	63569211543687	63569211589249	40562	7.4	300.1588
17	CTest1	63569212080269	63569212108741	28472	5.2	148.0544
18	DemoL0ck2	63569212530259	63569212565744	35485	4.4	156.1340
19	DSort	63569212795596	63569212823152	27556	8.6	236.9816
20	ErrorTest	63569213213927	63569213246397	32470	5	162.3500
21	If–Else	63569213479918	63569213522539	42621	9.4	400.6374
22	InterfaceJava	63569213776108	63569213827910	51802	8.6	445.4972
23	LockHeld	63569214296410	63569214349198	52788	5.1	269.2188
24	MultiLock	63569386907220	63569386942754	35534	11.9	398.7763
25	NewJava	63569387451868	63569387489345	33477	9.6	341.1264
26	NoPredictive	63569388631364	63569388667877	36513	8.7	317.6631
27	NoPredictive2	63569388911397	63569388948915	37518	9.6	360.1728
28	NS	63569390044291	63569390283628	239337	233.3	55837.3221
29	NS2	63569389628499	63569389874903	246404	244.6	60270.4184
30	FinallyTest	63569390701523	63569390735086	33563	8.4	281.9292
31	OneWrite	63569390919680	63569390958207	38527	13.1	504.7037
32	Regression	63569391900297	63569391940889	40,592	6.2	251.6704
33	SampleXACMLPolicy	63569392159011	63569392208760	49,749	16.5	820.8585
34	StackTest	63569392389298	63569392415695	26,397	4.5	118.7865
35	StaticInstanceProblem2	63569392640802	63569392672252	31,450	10	314.5000
36	StringBufferModule	63569392875182	63569393168289	293,107	241.9	70,902.5833
37	StringBuffer1	63569393418796	63569393519208	100,412	81.1	8143.4132
38	StringBuffer2	63569393690719	63569393801371	110,652	78.1	8641.9212
39	Struct	63569393953519	63569393983941	30,422	7.5	228.1650
40	Testme	63569394155321	63569394182701	27,380	7.7	210.8260

software. For an example of power usage, they have estimated 13.616 W on 3 h of charging using *OpenSolaris PowerTop version 1.1*. They have stated that *PowerTop* represents a great first step by providing programmers and administrators with observability into software inefficiency, a point of reference for optimization, and awareness of the important role software plays in energy-efficient computing. Through our proposed work, we compute the energy consumptions of different software testing tools. Whichever tool takes less amount of energy will be preferable to use for further task.

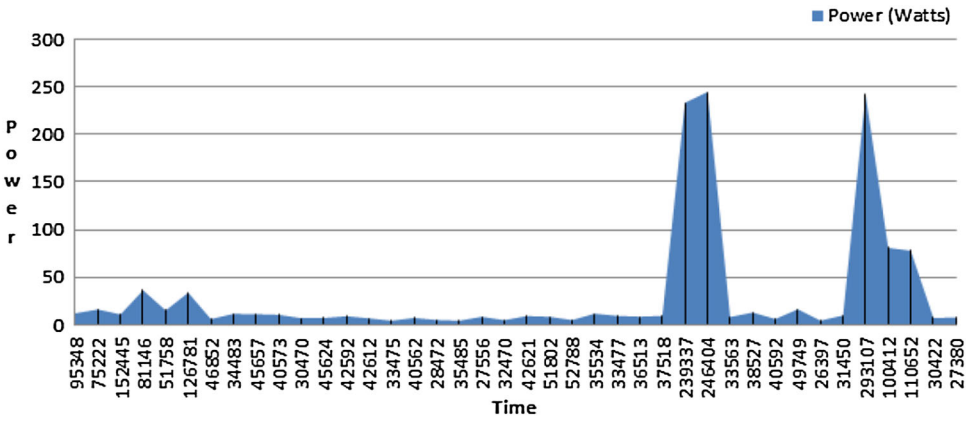


Fig. 5 Recorded power consumption versus execution time

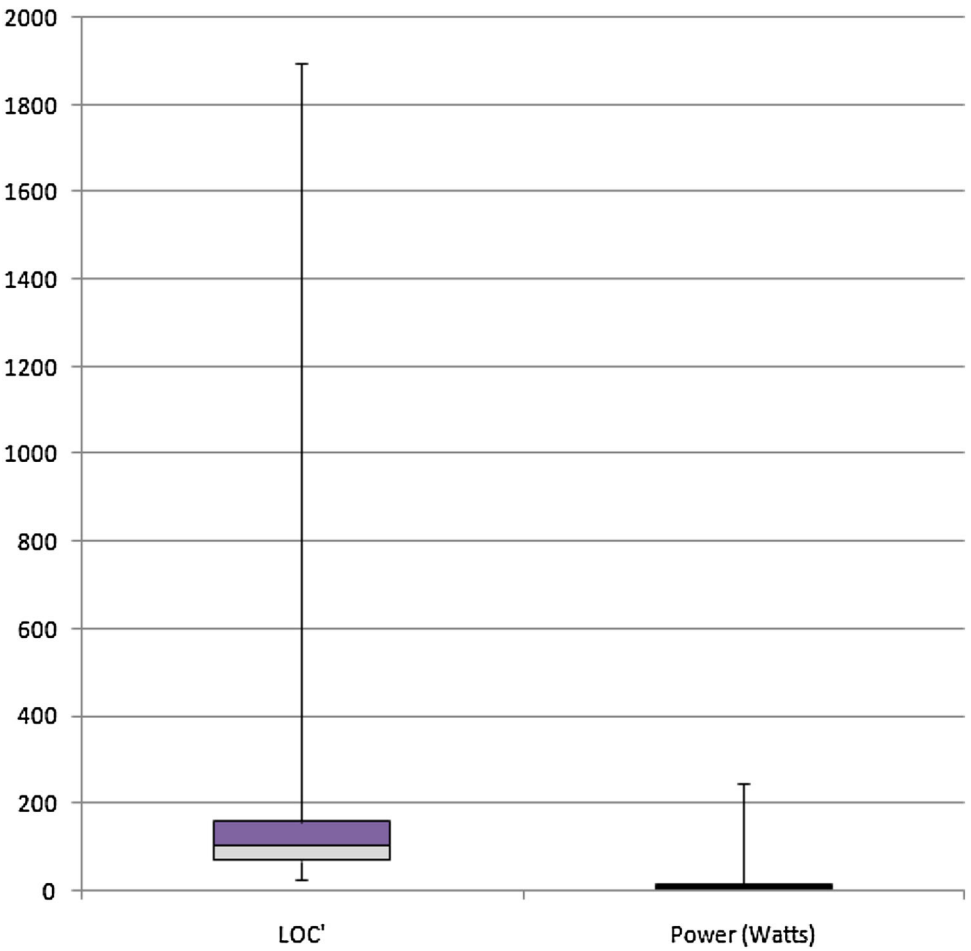
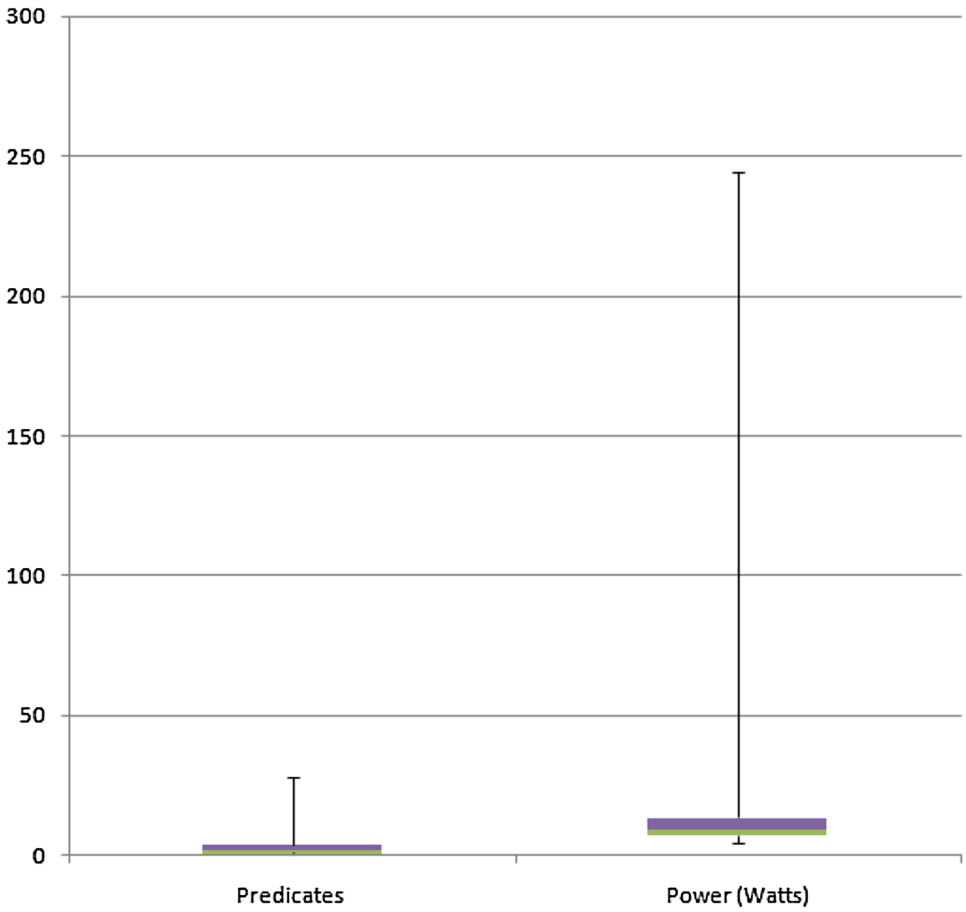


Fig. 6 LOC′ versus power (note: LOC′ is the lines of code in the transformed program)

Sen et al. [5,12] developed concolic testers Concolic Unit Testing Engine (CUTE) and JCUTE for C and Java programs, respectively. In this paper, we have used the same version of JCUTE as proposed by Sen et al. [5,12] to generate the required test cases automatically for concolic testing.

Das et al. [19] used the program transformation technique to achieve high modified condition/ decision coverage (MC/DC) for C programs. Das et al. proposed Boolean Code Transformer (BCT) to enhance coverage. They have used CREST to generate test cases and proposed coverage analyzer (CA) to measure MC/DC%. However, they did not give an analysis of the time overhead incurred due to code transformation. Godboley et al. [11] extended the code transformer in [19] using Quine–McCluskey minimization technique that was later extended in [10] to transform Java programs. Another approach of code transformation for distributed concolic testing can be referred in [20,21]. But, none of the above approaches [10,11,19] focused on energy con-

Fig. 7 Predicate versus power



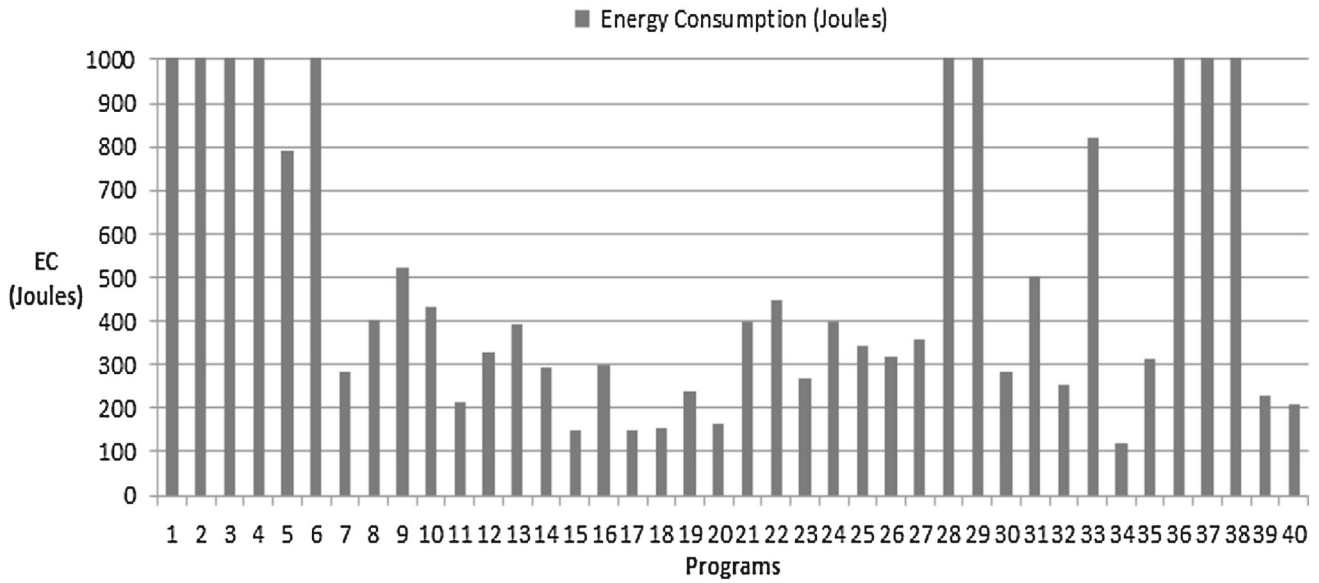
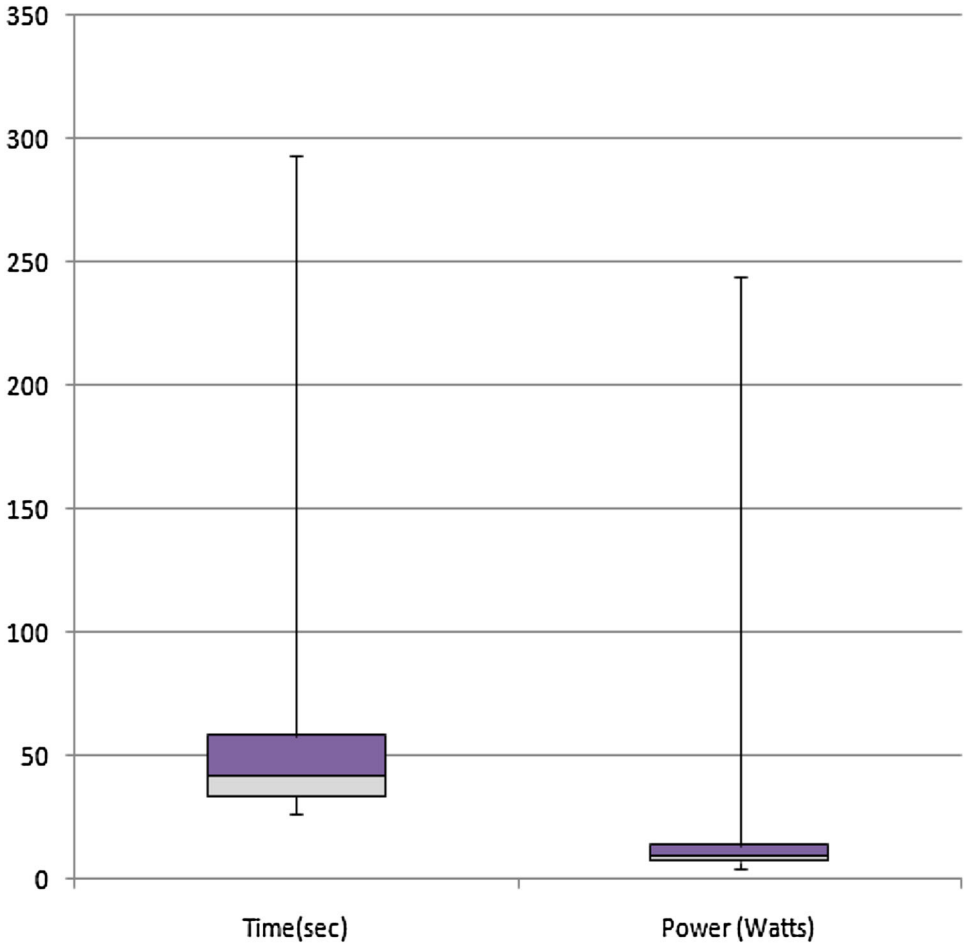


Fig. 8 Energy consumption for forty programs

sumption awareness. In this paper, the authors extended the approach in [10] to support energy awareness in concolic testing. In our proposed approach, we used JPCT and JCUTE to transform the code and generate test cases, respectively. In this work, the experimentation is carried on forty Java programs as compared to five programs in [10].

Sarkar et al. [22] proposed two new metrics RepQ and RDI to quantify whether the repetitions found in a trace are

Fig. 9 Time versus power



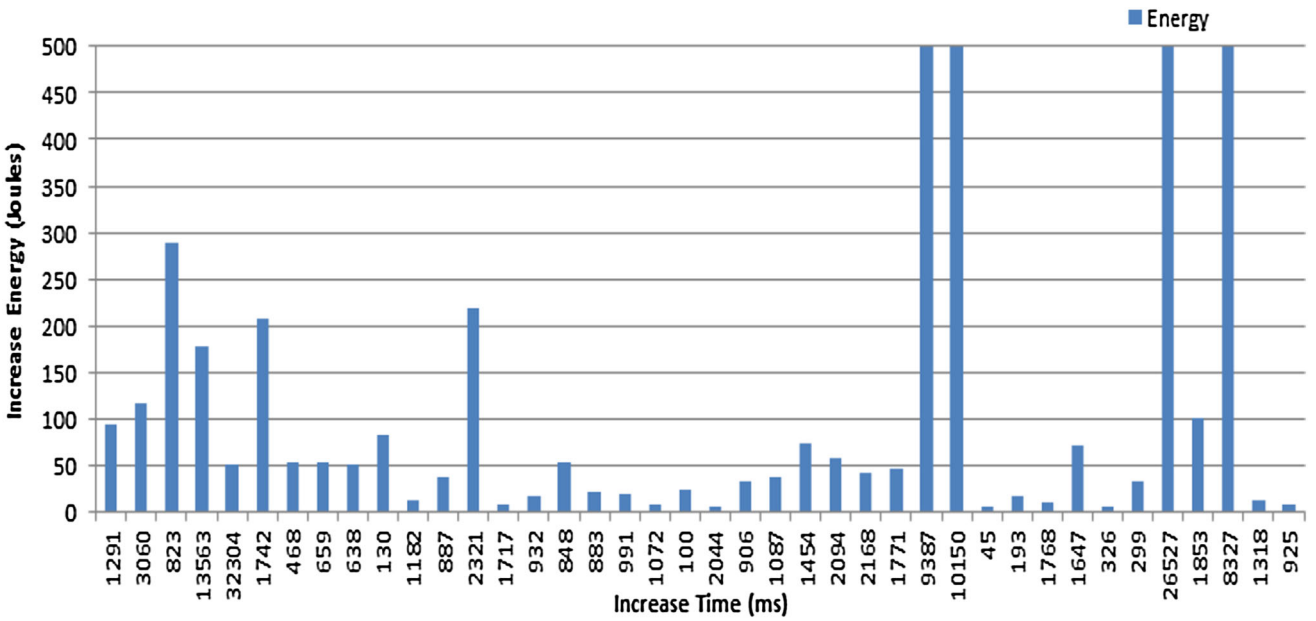


Fig. 10 Increase in energy consumption versus increase in time

good enough for loop restructuring. The empirical study also shows that RepQ and RDI values can identify data-parallel loops where loop structuring is not going to be effective. Their result indicates that they achieved a significant performance improvement in those data-parallel loops which are favorable for divergent branch elimination. They have transformed the application for GPU to achieve improved branch coverage. In our approach, we proposed a transformation technique for Java programs. So, we achieved high branch coverage by integrating this transformer to concolic tester.

Benedict et al. [23] proposed an algorithm and energy tuning mechanism. It helps the energy conscious scientific application and tool developers. EnergyAnalyzer is an online-based energy consumption analysis tool that is under development at the HPCCLoud Research laboratory. [3] Similarly, our tool also aims to enable the software testers to remain aware of the energy consumed during testing.

Hassine et al. [24] proposed a suite of AsmetaL-specific mutation operators. These are classified into four categories. Their proposed operators are used to assess the adequacy of test suites generated using the ATGT (ASM Tests Generation Tool), according to various test coverage criteria. They have demonstrated the applicability of their approach through eight publicly available AsmetaL case studies. Their results of the case studies show that the proposed mutation operators can be used to compare different AsmetaL-based test coverage criteria and successfully detect inadequacies in test suites. In our proposed work, we performed concolic testing and computed branch coverage. However, we extend our experiment by evaluating time, speed, power and energy consumption.

Mao et al. [25] mainly focused on software structural testing and proposed a search-based test data generation technique. They have proposed a framework, in which particle swarm optimization (PSO) technique is adopted due to its simplicity and fast convergence speed. For test data generation problem, the inputs of program under test are encoded into particles. After producing test data inputs, coverage information is collected by test driver. After this, the fitness value of branch coverage is calculated based on such information. Therefore, the fitness is used for PSO to adjust the search direction. At last, the test data set with the highest coverage rate is yielded. In our approach, we used dynamic symbolic execution to automatically select the input values. JCute tool generates test cases and produces branch coverage as output report. In our approach, we have also computed speed of test case generation.

Varshney et al. [26] proposed an elitist GA-based (Genetic Algorithm) technique that used the concepts of dominance and branch distance to generate test data for the data flow dependencies (all-uses criterion) of a program. Their approach can be used for programs with/without loops and procedures. In our proposed work, we use concolic testing tool to generate test cases, which are further used to compute branch coverage.

Khan et al. [27] proposed a model transformation approach from UCM notation to a Unified Modeling Language (UML) sequence diagram to facilitate the transition from requirements to high-level design artifacts. They also presented an application of the proposed model transformation to an elevator control system. In our proposed work, we transformed a Java program to its transformed version. This transformed program consists of nested if–else statements. By using the transformed version of the input Java program, we achieved higher branch coverage percentage.

Table 10 summarizes our survey on some of the available related work proposed by different authors. The third column, named *Framework*, lists the different tools and technologies adopted by the researchers in their respective work followed by a brief description given in the fourth column. The research work listed in *Sl. No. 1 to 7* has their research based on power and energy consumption. These works focused on spreading awareness concerning GREEN IT and GREEN Software Engineering. The research work listed in *Sl. No. 8 to 17* deals with concolic and coverage-based testing. The last row in Table 10 shows that our proposed work implemented on a tool, named *Green-ABCE*, confirms to both energy awareness and concolic testing.

We compare our proposed work with the existing related work in Table 11 based on some selected characteristics identified during the survey. This table clearly shows whether these characteristics shown in the columns are incorporated (✓) or not (X) in the referred related work. Among all the existing work, only Li et al. [2] have made an attempt to analyze the energy consumption in software testing. Please note that research work referred in *Sl. No. 2 to 7* has proposed some approaches to compute the consumption of energy and power. But they do not focus on software testing techniques. The research work listed in Sl. No. 8 to 17 focuses on software testing only without making any reference to the energy consumption. The last row in Table 11 shows that our proposed work (implemented on *Green-ABCE* tool) satisfies all the mentioned characteristics. Our proposed work addresses both the software testing and the Green computing aspects of Software Engineering. To the best of our knowledge, our developed tool (*Green-ABCE*) is the first concolic testing tool that can show the amount of energy consumed while enhancing the branch coverage of object-oriented (Java) programs. Thus, it is our first step toward Green software testing.

Overall, through our literature study, we conclude that some existing works are done to spread the awareness on Green Software Engineering. But, as per our knowledge and study, our proposed work is the new step toward Green Software Testing. We have integrated some open-source tools

[3] http://www.sxcce.edu.in/hpccloud/.

Table 10 Summary of concolic and coverage based testing approaches

Sl. no.	Authors	FrameWork	Description
1	Li et al. [2]	EDTSO	Based on encoding minimization problem as integer linear programming problem
2	Amsel et al. [1]	GreenTracker	Estimates the Energy Consumption of software in order to help concerned uses make informed decision about the software they use
3	Dick et al. [14]	PM, WG, DAE	How to measure the Energy Consumption of software
4	Chen et al. [15]	StressCloud	Analyzing the performance and energy consumption of a cloud application
5	Capra et al. [16]	WLS, AC, VIM	Proposed method for software energy efficiency for application software
6	Brown et al. [17]	—	Article suggests an overview of all approaches to energy efficiency in computing system
7	Saxe et al. [18]	PowerTOP	This work shows the extent of the waste, while also showing which software is responsible
8	Das et al. [19]	BCT, CREST, CA	Based on concolic testing and MC/DC testing BCT uses K-Map as minimization technique
9	Bokil et al. [28]	AutoGen	Analysis of all coverage criteria with time effort
10	Godboley et al. [10]	JPCT, JCUTE	Based on concolic testing and branch coverage testing. JPCT uses QM as the minimization method
11	Burnim et al. [7]	CREST	Worked on Heuristic Concolic Testing and branch coverage
12	Kim et al. [29]	CREST	Based on concolic testing
13	Kim et al. [30]	CONBOL	Based on concolic testing. Analysis for branch coverage and time taken
14	Majumdar et al. [6]	CUTE	Worked on HCT and branch coverage
15	Kim et al. [31]	SCORE	Approach is build on distributed concolic testing
16	Sen et al. [5]	CUTE,JCUTE	Concolic Tool developed for C and Java Programs
17	Kim et al. [4]	SMT Solver, CREST	Based on HCT and analysis on reduction ratio
18	Proposed work	Green-ABCE	Spread the awareness on energy consumption analysis on Software testing techniques

Table 11 Characteristics of different approaches on concolic and coverage based testing

Sl. no.	Authors	Generated test cases	Measuring coverage%	Determined time constraints	Measured speed	Computed power consumption	Computed energy consumption
1	Li et al. [2]	✓	✓	✓	X	X	✓
2	Amsel et al. [1]	X	X	X	X	✓	✓
3	Dick et al. [14]	X	X	X	X	✓	✓
4	Chen et al. [15]	X	X	X	X	X	✓
5	Capra et al. [16]	X	X	✓	X	✓	✓
6	Brown et al. [17]	X	X	X	X	✓	✓
7	Saxe et al. [18]	X	X	X	X	✓	✓
8	Das et al. [19]	✓	✓	X	X	X	X
9	Bokil et al. [28]	✓	X	✓	X	X	X
10	Godboley et al. [10]	✓	✓	✓	✓	X	X
11	Burnim et al. [7]	✓	X	X	X	X	X
12	Kim et al. [29]	✓	X	X	X	X	X
13	Kim et al. [30]	✓	✓	✓	X	X	X
14	Majumdar et al. [6]	✓	X	X	X	X	X
15	Kim et al. [31]	✓	✓	X	✓	X	X
16	Sen et al. [5]	✓	✓	✓	X	X	X
17	Kim et al. [4]	✓	X	X	X	X	X
18	Proposed Work	✓	✓	✓	✓	✓	✓

and developed some tools to perform Green Software Testing process. Our proposed tool produces several useful outputs, which are important in testing domain as well as Green Computing domain. We can observes that only Li et al. [2] have computed testing parameters along with energy report. None of the authors listed in Table 11 worked on both the domains (Software Testing and Green Computing). This is the main difference between our proposed work with other. Our main contribution is to achieve high branch coverage along with the power and energy reports. Therefore, we have developed Green-ABCE using some open-source and our developed tools (JPCT,jCUTE,JouleMeter and Energy Calculator), which are already discussed previously in detail in Sect. 3. In our proposed work, in case of Scenario 1, the average computation/execution time for the forty programs is 23,678.93 ms, while in Scenario 2, the average computation/execution time for the forty programs is 27,176.42 ms. The average speed computed for Scenario 1 is, speed $= 4.66$ TC/s, and average speed for Scenario 2 is speed$' = 2.27$ TC/s for the forty programs. Our proposed approach achieved approximately 13.5 % higher branch coverage for the forty programs we tested with an overhead of 0.4 kJ of energy consumption. The average energy consumption of the tool for all the programs is approximately 5.6 kJ.

6 Threats to Validity

Like many other existing works, this proposed work also suffers from some limitations. These are mentioned below as existing threats to the validity of Green-ABCE.

1. The primary threat to our approach is related to the target programs. The programs chosen for this experimentation are amenable to concolic testing and do not reveal the characteristics (such as multi-threaded, distributed programs) that, if present, may not be suitable for the proposed approach. While concolic testing is indispensable for safety critical systems, our experimental programs do not represent the safety critical category.
2. There are certain limitations of the symbolic execution engine used in JCUTE that forms another threat to the validity of our approach.
3. The constraint solver that is utilized in the tool may not be powerful enough to compute the concrete values that satisfy the constraints in very large industrial applications. We are still working to overcome this limitation.
4. The low speed of Green-ABCE (due to code transformation) in generating the test cases is another concern with this proposed work. Even though the results do not establish a direct relationship of decrease in speed with that of the increase in the program size, still the speed is expected to become slower for very large programs as the experimental programs are not good representative of industrial programs.
5. JouleMeter is developed for *Windows 7* operating system. As a result, our tool is currently restricted to this particular platform only.
6. Another threat to the validity is that the readings of JolueMeter are only suggestive values for the power consumption. But the manual of JouleMeter suggests the integration of electronic pro model such as *Wattsup* for a more accurate reading. The proposed work has not utilized any such model. Our framework uses only the proxy readings of JouleMeter owing to the cost of the electronic pro model and unavailability in the local market.

7 Conclusion and Future Work

We proposed a concolic testing framework for Java programs and implemented in a tool, named *Green-ABCE*. The proposed framework is designed and developed to measure the energy consumption while analyzing the increase in branch coverage using concolic testing. An overall view of Green-ABCE followed by a discussion on the working of its different components has been presented in detail. The experimental results show that the proposed approach of test case generation achieved better branch coverage in comparison with the existing methods. Our proposed method achieved approximately 13.5 % of the increase in branch coverage for the forty programs with an overhead of 0.4 kJ of energy consumption. The average energy consumption of the tool for all the programs is approximately 5.6 kJ.

In future, we focus on overcoming the identified threats to the validity of our approach. We will extend this work to implement the proposed approach with other energy-efficient transformation techniques such as Java version of Exclusive-Nor Code Transformer (EX-NCT) so that we will achieve high branch coverage. We are also planning to work on energy consumption for MC/DC testing method. Further, we also aim to develop a distributed framework to increase the scalability of our approach.

References

1. Amsel, N.; Tomlinson, B.: Green tracker: a tool for estimating the energy consumption of software. In: Proceedings of the 28th International Conference on Human Factors in Computing Systems, CHI, pp. 3337–3342, Atlanta, Georgia, April 10–15 (2010)
2. Li, D.; Sahin, C.; Clause, J.; Halfond, W.G.J.: Energy-directed test suite optimization. In: 2nd International Workshop on Green and Sustainable Software (GREENS), pp. 62–69, New York, USA, May (2013)
3. Kan, E.Y.Y.; Chan, W.K.; Tse, T.H.: EClass: an execution classification approach to improving the energy-efficiency of software via machine learning. J. Syst. Softw. **85**(4), 960–973 (2012)

4. Kim, M.; Kim, Y.; Choi, Y.: Concolic testing of the multi-sector read operation for flash storage platform software. Form. Asp. Comput. **24**(3) (2012)
5. Sen, K.; Agha, G.: CUTE and JCUTE: concolic unit testing and explicit path model-checking tools (Tools Paper). DTIC Document (2006)
6. Majumder, R.; Sen, K.: Hybrid concolic testing. In: Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, pp. 416–426 (2007)
7. Burnim, J.; Sen, K.: Heuristics for scalable dynamic test generation. In: Proceedings of ASE, pp. 443–446, Washington, DC, USA (2008)
8. RTCA, Inc.: RTCA/DO-178B, Software Considerations in Airborne Systems and Equipment Certification, Washington, D.C. December (1992)
9. Kern, E.; Dick, M.; Naumann, S.; Guldner, A.; Johann, T.: Green software and green software engineering–definitions, measurements, and quality aspects. In: Proceedings of ICT4S, ETH Zurich, pp. 87–94, February 14–16 (2013)
10. Godboley, S.; Sahani, A.; Mohapatra, D.P.: ABCE: a novel framework for improved branch coverage analysis. In: SCSE, Procedia Computer Science, Elsevier, University of California, Berkley, USA **62**, 266–273 (2015)
11. Godboley, S.; Prashanth, G.S.; Mohapatra, D.P.; Majhi, B.: Increase in modified condition/decision coverage using program code transformer. In: IEEE 3rd International Advance Computing Conference (IACC), pp. 1400–1407, Feb (2013)
12. Sen, K.: Automated test generation using concolic testing. ISEC, Bangalore, India, p. 9, February 18–20 (2015)
13. Liu, C.L.; Mohapatra, D.P.: Elements of Discrete Mathematics: A Computer Oriented Approach, 3rd edn. Tata McGraw-Hill Education, New York (2013)
14. Dick, M.; Kern, E.; Drangmeister, J.; Naumann, S.; Johann, T.: Measurement and rating of software induced energy consumption of desktop PCs and servers. In: EnviroInfo 2011: Innovations in Sharing Environmental Observations and Information, Shaker Verlag Aachen (2011)
15. Chen, F.; Grundy, J.; Schneider, J.G.; Yang, Y.; He, Q.: Automated analysis of performance and energy consumption for cloud applications. In: Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering, ACM, pp. 39–50, Dublin, Ireland (2014)
16. Capra, E.; Francalanci, C.; Slaughter, S.A.: Measuring Application Software Energy Efficiency. IT Prof. **14**(2), 54–61 (2012)
17. Brown, D.J.; Reams, C.: Toward energy-efficient computing. Commun. ACM **53**(3), 50–58 (2010)
18. Saxe, E.: Power-efficient software. Commun. ACM **53**(2), 44–48 (2010)
19. Das, A.; Mall, R.: Automatic generation of MC/DC test data. In: Proceeding of International Journal of Software Engineering, Acta Press **2**,(1) (2013)
20. Godboley, S.; Prashanth, G.S.; Mohapatra, D.P.; Majhi, B.: Enhanced modified condition/decision coverage using exclusive-NOR code transformer. In: 2013 International Multi-Conference on Automation, Computing, Communication, Control and Compressed Sensing (iMac4s), pp. 524-531, March (2013)
21. Godboley, S.; Panda, S.; Mohapatra, D.P.: SMCDCT: a framework for automated MC/DC test case generation using distributed concolic testing. In: International Conference on Distributed Computing and Internet Technology, LNCS, Springer, KIIT Bhhubneshwar, India **8956**, 199–202 (2015)
22. Sarkar, S.; Mitra, S.: A profile guided approach to optimize branch divergence while transforming applications for GPUs. ISEC, Bangalore, India, pp. 176–185, February 18–20 (2015)
23. Benedict, S.: Threshold acceptance algorithm based energy tuning of scientific applications using energy analyzer. ISEC, Chennai, India, pp. 11:1–11:6, February 19–21 (2014)
24. Hassine, J.; Alkrarha, O.: A mutation-based approach for testing AsmetaL specifications. Arab. J. Sci. Eng. **40**(12), 3523–3544 (2015)
25. Mao, C.: Generating test data for software structural testing based on particle swarm optimization. Arab. J. Sci. Eng. **39**(6), 4593–4607 (2014)
26. Varshney, S.; Mehrotra, M.: Search-based test data generator for data-flow dependencies using dominance concepts, branch distance and elitism. Arab. J. Sci. Eng. **41**(3), 853–881 (2015)
27. Khan, Y.; Mahmood, S.: Generating UML sequence diagrams from use case maps: a model transformation approach. Arab. J. Sci. Eng. **41**(3), 965–986 (2015)
28. Bokil, P.; Darke, P.; Shrotri, U.; Venkatesh, R.: Automatic test data generation for C programs. In: 3rd IEEE International Conference on Secure Software Integration and Reliability Improvement (2009)
29. Kim, M.; Kim, Y.; Jang, Y.: Industrial application of concolic testing on embedded software: Case studies. In: IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST), pp. 390–399 (2012)
30. Kim, Y.; Kim, Y.; Kim, T.; Lee, G.; Jang, Y.; Kim, M.: Automated unit testing of large industrial embedded software using concolic testing. In: IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), pp. 519–528, Nov (2013)
31. Kim, M.; Kim, Y.; Rothermel, G.: A scalable distributed concolic testing approach: an empirical evaluation. In: IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST), pp. 340-349, April (2012)