IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received September 22, 2018, accepted October 17, 2018, date of publication October 22, 2018,
date of current version November 14, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2877082

Energy Consumption Fuzzy Estimation for Object-Oriented Code

HUI LIU[1], FUSHENG YAN[1], JINGQING JIANG[2], AND JIE SONG[3]
[1]School of Metallurgy, Northeastern University, Shenyang 110819, China
[2]College of Computer Science and Technology, Inner Mongolia University for the Nationalities, Tongliao 028000, China
[3]Software College, Northeastern University, Shenyang 110819, China

Corresponding author: Jie Song (songjie@mail.neu.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61672143, Grant 61433008, Grant U1435216, Grant 61662057, Grant 61502090, and Grant 51606031, and in part by the Fundamental Research Funds for the Central Universities under Grant N161602003.

ABSTRACT The energy consumption (EC) estimation of a computing system is a primitive approach for evaluating its energy efficiency and for further optimization. Existing studies adopt the runtime-dependent approach to measure the EC of software; however, in this paper, the source-level and fuzzy estimation approach are employed to measure the EC of software code, especially object-oriented code, before it is executed. This approach is beneficial to source code quality improvement and EC static optimization. For runtime independence, a fuzzy energy consumption (FEC) model is proposed, in which the code and the EC model, as well as the mappings between them, are first defined; then, the process for estimating the FEC of an arbitrary statement is described, including the statement features, data preparation, fuzzy comprehensive evaluation, and fuzzy pattern matching. Finally, experiments are performed, including the regularities verification, the statement features analysis, the EC and FEC comparison, and the FEC application. The experimental results show that the mean values of EC/FEC for the selected test cases are stable, their standard deviation is approximately 0.00064, and their mean is approximately 0.0059. By FEC, it is feasible to compare the ECs of code statically with reasonable accuracy.

INDEX TERMS Fuzzy model, energy estimation, fuzzy energy consumption, object-oriented code.

I. INTRODUCTION

As the energy crisis grows, the techniques for energy-saving and emission reduction are attracting more attention from academia, business researchers and IT professionals [1]. The high energy cost of computing devices has led to green IT as a new research area. Energy consumption (EC for short) estimation and optimization of computing systems are two popular topics in this new research area, which can be studied at the resource level, platform level (middleware) or application level. Application-level EC optimization reduces the energy consumed by the consumer (applications), which is one effective approach [2]. Therefore, there are many existing studies on software-oriented EC estimation and optimization [3].

Software is composed of code, and object-oriented programming languages are the dominant type of programming language; therefore, the energy estimation on object-oriented (OO for short) code, which is treated as a static energy estimation, is an efficient approach in the application layer. Code EC is defined as the electrical energy cost by hardware while the code is executing. In OO code, the features of code include operation types (calculation, assignment, method calls, etc.), statement structure (sequence, branch, loop, etc.) and the OO features (encapsulation, inheritance, polymorphism, etc.). We defined the features above as code features. Code with the same semantics but different features have different EC. When the power of the CPU changes dynamically, the working state of the CPU is directly affected by the code features, and the power of the storage devices also changes dynamically. Code features directly affect the accessed location and the amount of accessed data.

Currently, research on source level EC estimation is considered in three different granularities: instruction level, statement level and module level. Instruction level estimation simply adds together the EC of each instruction, statement level estimation studies the effect of statement features to the EC, and module level studies the relationship among software modules (method, or class, as the estimation unit) and their

effects on EC. Existing studies on instruction level EC estimation has the limitation of applications; they are mainly used in embedded software. The estimation approach, which more accurately estimates the hardware, has worse universality. In contrast, the more abstract approach has a larger error but better universality. It is difficult to accurately determine the execution paths of code with complex structures, such as choosing a branch, or the number of times a loop executes, without executing the code. The EC estimation does not aim to replace the measuring instrument but to locate the high EC code, optimizing the EC or evaluating the optimization effect. Thus, the near accurate estimation approach is acceptable, and a reasonable simplified, fuzzy logic-based, and code features highlighted approach is more applicable.

The fuzzy energy consumption (FEC) of software code defined in this paper is a measurement that is retrieved by comparing the target code with the benchmarked code with fuzzy comprehensive evaluation and pattern matching. While assuming the EC of a statement with specific features as the unit, FEC of the code ignores the runtime situation of code, i.e., it is an aggregation of theoretical and the static EC of the statements.

In this paper, we study the FEC of OO code, which is the abbreviation for source code written in an object-oriented programing language such as Java. The proposed FEC is an estimation model of OO code. We define the FEC model, explain the estimation approach, and validate it by experiments. The code's FEC is analogous to code performance. In performance optimization, runtime performance of the code is related to not only the code but also to the amount of data, the data structure and the hardware environment; these comprehensive factors should be well considered. Static code optimization tools such as *PC-lint* can evaluate and optimize static performance according to the code features, which is also significant in avoiding low-performance programming. Similarly, the FEC of the code in this paper is also a static measurement; thus, runtime EC may differ from FEC, but FEC can evaluate the difference in code EC or the optimization effects. FEC also has the benefit of summarizing code-level EC optimization approaches.

Our solution is different from existing approaches on EC estimation; we focus more on the features of OO code, adopt fuzzy logic to implement the static estimation, and study the effects of OO statement features on statement EC. The definition of FEC is an abstraction; it is widely suitable for EC optimization in the application software environment. In programming phase of development, evaluation of the EC related features of the code and code refactoring can effectively avoid redesign and recoding caused by energy problems that may occur in later phases of the software life cycle.

The remainder of this paper is organized as follows: Section II introduces the state of the art and analyzes the advantages of our solutions. Section III proposes a theoretical model for EC estimation, i.e., the FEC model of OO code. Section VI proposes the fuzzy estimation approach. Section V validates the code EC features and the accuracy of FEC. Finally, Section VI concludes the paper and proposes future work.

II. STATE OF THE ART

The studies of EC estimation on computing systems are categorized into two aspects: hardware and software. Studies at the hardware-level consider the EC characteristics of CPUs, memory and disks to evaluate consumed energy or to design more energy-saving hardware components; the solutions are mature with years of development [4]. On the other hand, there are many studies in EC estimation and optimization of software, which are grouped into instruction level, statement level and module level [5]–[7]. The dominant approach in these studies is cumulatively mapping the EC of the hardware unit to the software unit, considering the perspective of the latter, such as the instructions, statements and modules, to study the EC estimation and optimization.

A. INSTRUCTION LEVEL

The instruction-level energy estimation first collects the runtime EC of the instructions generated by the code on the target hardware (such as the processor) and estimates the EC of the code by aggregating the EC of the instructions. This approach is treated as a ''white-box'' approach because predefined hardware information is required. The advantage of the white-box approach is a high estimation accuracy, but the disadvantage is the tight coupling to the hardware environment, as well as the lack of availability of instruction simulators with accurate cycles in many hardware environments, or environments for interpreting the execution of programming languages. Therefore, instruction-level estimation has been widely applied to embedded software [5].

Some researchers have noticed that the difficulties of the white-box approach lie in mapping the EC of code to the EC of the instructions. It is much easier to simplify the execution of the corresponding instructions if the code is treated as a black box. The parameters provided by the operating system represent the execution of instructions, by which the EC is ''measured'' or ''predicted''. For example, in [7], the different test cases such as the Burn CPU, MemLoop, Network, Tar Kernel, Disk Read, Disk Write, were used to analyze the relationship between the instruction and the EC and to predict the code EC. In [8], a set of process-level power measurement tools were developed that accurately evaluate the energy usage of every process running on Linux, and then evaluated the EC of the software system.

B. STATEMENT LEVEL

Compared with the instruction level, the statement-level EC estimation is more coarsely granular since a statement may contain many instructions. Most statement-level approaches evaluate the code EC by executing it or building a tool for evaluating EC during code execution [9], such as Eprof [10] designed by Noureddine *et al.* [11], and process-level and device-level monitoring frameworks, as well as [6] and [12], which are also similar studies. To compare with our solution,

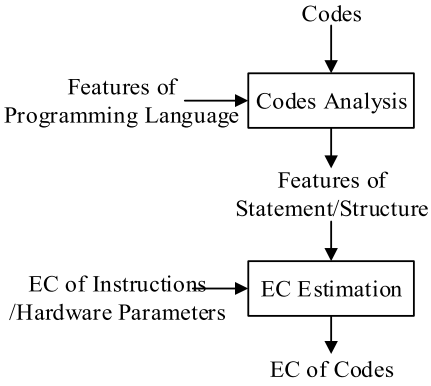

FIGURE 1. Statement-level EC estimation.

in this section, we focus on the approaches that use static analysis. The approach divides the code into the form of statement blocks and evaluates the EC of each block. Statement-level energy estimation considers the logical relationship between the statements, that is, the impact of the statement structure on EC. The general approach of statement-level EC estimation is shown in Figure 1.

Brandolese [13] followed the approach shown in Figure 1 and built a ParseTree by source code analysis. Each node is an atomic unit consuming energy, and the edges of the tree represent the assembly relations to characterize the features of the structure of the source code and the statements. Zhou *et al.* [14] proposed a code EC model for C language. They argued that the execution of an instruction contains three phases: instruction acquisition, decoding and execution so that the EC estimation should take the instructions of the former two phases into consideration, not only the instructions of the latter phase.

C. MODULE LEVEL

From a code perspective, code units with appropriate cohesion and weak coupling are called modules. Module-level software EC evaluation not only considers the statement of the EC and the impact of the statement structure on the EC but also considers the dependencies between modules on EC [15]. To compare with our solution, the object-oriented programming related approaches are explained in this section. In object-oriented programming languages, the method can be treated as the smallest module, while the class or package is treated as a module with a fair granularity.

Seo *et al.* [16] proposed the EC estimation approach of Java code from the perspective of software modules, including code EC corresponding to the CPU operations, memory accesses, and I/O operations. Different than [16], FEC is based on a fuzzy model but not the concrete codes. Similarly, some studies adopt abstract modules rather than OO code, such as flowcharts [17] whose elements encapsulate modules categorized into processes, I/O and process control, a ''flowchart of concurrent software modules'' [18], or UML and Petri nets, which characterize the invocation relationship among modules to estimate the EC of the code [19].

D. OUR SOLUTION

Based on the above discussion, the code-oriented software EC is considered from the three different granularities of the instruction level, the statement level and the module level. The estimation that is closer to the hardware platform has more accuracy but poorer universality. In contrast, the more abstract estimation has a larger error but better universality. In our solution, the latter is applied for two reasons: first, the purpose of EC estimation is not to replace the instrumental measurement but to locate high EC code to optimize EC or to evaluate EC optimization effects, so that the static approach is definitely required though an accurate estimation may not be necessary. In addition, accurate estimation cannot be achieved if the code is not executed because it is difficult to accurately determine its execution path, such as the choice of the branch, the loop execution times, and the size of the data structure, without execution. Therefore, accurate estimation is difficult to achieve. Consequently, an appropriate simplified approach that highlights the characteristics of the code is another option since the estimation cannot achieve high accuracy.

In this paper, fuzzy theory is adopted to evaluate the code EC statically. We neither expect exact and actual values, nor expect a coarse qualitative assessment of ''good'', ''normal'', ''bad'', but instead, a fuzzy quantitative value. The code model and the EC model are first established, then the fuzzy sets are established for the EC values of the statement features, and the comprehensive evaluation method evaluates the statement EC as the criterion; next, the fuzzy pattern recognition evaluates the EC of the target code statement-by-statement, and the aggregated EC values are finally obtained.

Compared with the state-of-the-art on software EC estimation, the hardware independent, instruction-level approaches have good accuracy, fine granularity and high estimation costs; thus, it is difficult to extend this method to high-level statement features. The module level approach is from the system architecture point of view so that it is coarse-grained, cannot show the statement features and cannot benefit code optimization. The statement level approaches focus on the procedure-oriented code which limits this approach to special systems and lacks universality. Our solution is also a statement level approach but for object-oriented programs and with wider applicability.

III. MODEL

In this section, the estimation model for the EC of OO code is introduced. The EC estimation is not a replacement of equipment-based EC measuring because the former method is a ''white-box'' approach and the latter method is a ''black-box'' approach. By the latter approach, the EC value is obtained accurately; however, the EC regularities of the code cannot be further analyzed; also, the measurement is tied to the power meters and runtime environment. The software-based EC estimation has significant value in both theory and practice. Essentially, software estimation is the mapping

between code and EC, and then the code and the EC model, as mentioned in this section, should be defined first.

A. CODE MODEL

This section defines the code model including statements, methods, classes and modules.

Definition 1 (Statement): The statement S is the unit with the finest granularity for FEC estimation. The statement FEC is independent of its runtime environment. The same statements have the same FEC values, while the ratio between FECs for different statements is also constant.

From the aspect of their functions, the statements are categorized into arithmetic and logical expressions, jump statements, loop statements, call statements, dynamic storage management statements, etc. However, according to both the functions and the energy consumption characters, we divided the statements into three categories as follows.

- **Operation Statement:** Operation statement refers to the statement that performs arithmetic and logical operations on the data or the flow control operation, which characterizes the internal structure of the code, such as sequence, branch, and iteration.
- **Storage Statement:** Storage statement refers to statements that create, acquire, modify, and destroy objects, arrays, etc. in code. In addition, it characterizes the dynamic data management and storage of code.
- **Call Statement:** Call statement refers to the statement for invoking methods or functions. A call statement is normally short but requires a huge energy cost, and it characterizes the interclass structure of the code, i.e., the relation between classes. As an explanation, the dependence, polymorphism, and inheritance are represented as different call statements, such as normal call, virtual call, and constructor call, respectively.

Definition 2 (Method): The FEC estimation, method statistically contains the ordered statements. Methods are categorized as the follows:

- **Explicit method:**Explicit methods are methods whose definitions are contained directly in the code.
- **Implicit method:** Implicit methods are those that actually exist but have no code definition or are defined in a special format. For example, object creation and destruction implicitly call all of the superclass's constructors and destructors, or calling the virtual method would implicitly call the ''virtual method lookup'' method.
- **Interface method:**Interface methods are methods whose definitions are not contained in the code, but in an external program, such as a system library or third party components.

Definition 3 (Class): From the aspect of FEC estimation, a class is a natural container for methods; that is, not only the explicit methods defined in the class but also the noninterface methods called from the in-class methods, level-by-level until all noninterface methods have been included.

Definition 4 (Module): From the aspect of FEC estimation, the module, as a collection of multiple classes whose code are available, is the estimation target.

For example, in Java, a module represents the package. Programmers generally implement their modules for business functions and invoke the APIs provided by the libraries for basic services. The code in the libraries is unavailable, and their EC is fixed and cannot be optimized. This is why the interface method is defined, and by which the module excludes the methods in the libraries from its FEC. We will explain in the next section that the interface methods are contained in the code, but their definitions are unavailable, and their energy consumption is ignored.

B. ENERGY CONSUMPTION

A program consists of continuous statements. Thus, studying source-level EC estimation usually starts with defining the EC of each statement. However, we cannot trace the execution of all statements, or statically analyze the EC of all statements exhaustively. As a solution, we define the concept of fuzzy energy consumption (FEC).

The runtime EC of a statement is the amount of energy consumed by the hardware during the execution of the statement. Runtime EC can be measured by equipment. In contrast, the FEC of a statement is the amount of consumed energy represented by the static features of the source code. It is independent of the hardware and runtime environments so that it may differ from, but should be in accord with the runtime EC.

Definition 5 (Fuzzy Energy Consumption) (FEC): FEC is an EC with an uncertain value. It is employed to handle the concept of a partial true value of the EC, where the true value may range between completely true and completely false. FEC is measured by fuzzy pattern matching with the criteria that are abstracted from the statement features. FEC is estimated through the static analysis of the code, with joules as the unit.

Definition 6 (FEC of Code): Given a section of OO code, its FEC is defined as the sum of the FEC of the classes it contains, and the FEC of the class is the sum of the contained none-interface methods (see Definition 7) and statements. Let O be a finite set of classes contained in module M, and W be a set of statements of the module's method, and Υ is a finite set of interface methods in the statement. w is a statement The function $F(s)$ represents the EC of a statement evaluated by fuzzy estimation, then:

$$\begin{aligned} fec(M) &= \sum fec(O) = \sum \sum fec(W - \Upsilon) \\ &= \sum \sum \sum F(w|w \in W - \Upsilon) \end{aligned}$$

When measuring the FEC of a section of code, we first traverse through all the classes of the code one by one and determine the explicit and implicit methods in each class; second, the statements in each method are traversed, and the statement features are compared with the features of the criteria. Next, the fuzzy pattern matching approach is adopted

to determine which criteria the statement belongs to so that the statement FEC is obtained. Finally, method-by-method and class-by-class, the static and environment-independent FEC of the code is obtained by summing the statement FECs.

IV. FUZZY ESTIMATION

In this section, the approach and process to estimate the EC level of an arbitrary statement are described, including the statement features, data preparation, fuzzy comprehensive evaluation and fuzzy pattern match.

A. STATEMENT FEATURES

In the paper, the statement is the atomic unit of code EC and the statement features that should be quantitatively analyzed. The statement features are a finite set. Let the set U be all features, where $U = \{U_1, U_2, \ldots, U_k,\}$. Each element in the set U_i is one of the feature group. Different feature groups contain different features. Therefore, when describing the set of statement features, the elements in the set need to be analyzed according to the code.

Definition 7. (Statement Feature): Statement features are properties of a statement that represent its EC characteristics. Let set U be the statement features, and it contains a subset as the group of these features. The features in the same group are exclusive; thus, a statement may satisfy the features in different groups but not the features in the same group. We set up the set U as follows:

$$U=\{Operator, I/O, Method, Data, Virtualcall, Inheritance\}$$

Where: *Operator* group has the features of *bitwise*, *relation*, *logic*, *arithmetic*, *assignment*, *condition*, *loop*; I/O group has the features of *read* and *write*; *Method* group has the features of *explicit*, *implicit* and *interface*, in which the *explicit method* is the common method, *implicit method* is the method invoked by the runtime environment, such as constructors and finalizers, and*interface method* is the method of a three-party library whose EC is not included;*Data*group has the features of *primitive*, *primitive array*, *object*, and *object set*; *Virtual call* group has the features of *v-none*, *v-single* and *v-multiple*, which means a method has no virtual version, one virtual version and multiple versions, respectively; *inheritance* group has the features of *h-none*, *h-single*, and *h-multiple*, which means a statement is in a constructor of a class that has been inherited from no class, one class or multiple classes, respectively.

B. DATA PREPARATION

We defined the feature set U. In this section, the EC value set V (comment set), which represents the *levels* of EC, is defined in *Definition 8. In addition,* the relation matrix R, which is known as the importance of the features to the EC expressing the relationship between features and EC, is defined in Definition 9. With the prepared data U, V and R, fuzzy comprehensive evaluation is performed next.

People usually evaluate things by "very good", "good", "general", "bad" and so on. Our comment set is made up of similar kinds of elements but in the finest granularity. FEC adopts linguistic variables such as positive big, negative small, very small, and very large instead of numerical values. Through experience and experiments, the EC of Java operations and storage statements is *n*J order of magnitude (10^{-9}J) in most hardware environments. We divide EC under 4000 *n*J into 20 levels, each level represents 200*n*J distance, as *i-th* level is equal to $i \times 200$ *n*J, and we express this in "levels".

Definition 8 (EC Values): An EC value is a set that consists of the levels of the EC values. The continuing EC values from 0 to 4000 nJ are partitioned into 20 equal-width levels with the step of 200 nJ, and denoted as $V = \{1L, 2L, \ldots, 20L\}$, where "L" represents "level". Elements of V correspond to the EC level of 1L to 20L.

Definition 9 (EC Relation Matrix): An EC relation matrix R which represents contributions of statement features to the EC, the elements r_{ij} in R means the EC contribution of the statement features u_i on the values v_j.

The relation matrix R is determined by data statistics after obtaining a larger number of basic data. We modify the SPECJVM2008 test suite and design 20 sets of test cases. Each test case has a large number of statements and related features, and the statements are tested one-by-one. The results are as shown in Figure 2. Then, the contributions of these features to the EC are determined.

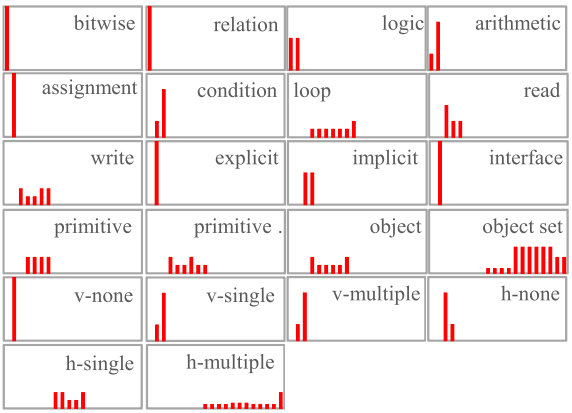

FIGURE 2. Contributions of the statement features to the EC.

In each chart of Figure 2. The x-axis is the level of the FEC from 1L to 20L, and the y-axis shows the contributions of statement features (chart name) to the level of the FEC in the range of 0 to 1. For example, the contribution of *bitwise* to FEC 1L, 2L, 3L, ..., 20L are 1, 0, 0, ..., 0, respectively.

Aforementioned:

$$U = \{U_1, U_2, \ldots, U_6\} = \{u_1, u_2, u_3, \ldots, u_{22}\},$$
$$V = \{1\text{L}, 2\text{L}, \ldots, 20\text{L}\}.$$

For each u_i, r_{ij} represents the degree of membership on u_i to v_j, $(i = 1, 2, \ldots, 22, j = 1, 2, \ldots, 20)$, where $r_{ij} \in [0, 1]$. R is denoted as the fuzzy matrix of feature u_i on

level v_j:

$$R = \begin{pmatrix} r_{11} & r_{12} & r_{13} & \cdots & r_{1(20)} \\ r_{21} & r_{22} & r_{23} & \cdots & r_{2(20)} \\ r_{31} & r_{32} & r_{33} & \cdots & r_{3(20)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ r_{(22)1} & r_{(22)2} & r_{(22)3} & \cdots & r_{(22)(20)} \end{pmatrix}.$$

C. FUZZY COMPREHENSIVE EVALUATION

In fuzzy evaluation, every evaluation element has a different contribution to the statement EC. Thus, for a specific statement s, we have to obtain the weight coefficient vector A, namely, the property values of the code of the statement:

$$A = \{a_1, a_2, \ldots, a_{(22)}\}$$

Then, we can use the weight coefficient vector A of s and fuzzy relation matrix R to evaluate our specific statement. We compute the comprehensive evaluation matrix (vector) B as follows:

$$B = A \circ R = \{a_1, a_2, \ldots, a_{(22)}\} \circ \begin{pmatrix} r_{11} & r_{12} & r_{13} & \cdots & r_{1(20)} \\ r_{21} & r_{22} & r_{23} & \cdots & r_{2(20)} \\ r_{31} & r_{32} & r_{33} & \cdots & r_{3(20)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ r_{(22)1} & r_{(22)2} & r_{(22)3} & \cdots & r_{(22)(20)} \end{pmatrix},$$

where B is a fuzzy set on V, which is denoted by $B \in F(V)$, B is a fuzzy vector which not only represents all evaluation elements contributions but also reserves all degrees of membership of every level.

There are two typical models for fuzzy comprehensive evaluation. Namely, the operator "$\circ$" can represent two kinds of composited operations, M $(\wedge, \vee)$, and M $(\bullet, \oplus)$, where operator $a \vee b = min(a, b)$, $a \wedge b = max(a, b)$, $a \bullet b = ab$, and $a \oplus b = min\{1, a + b\}$. Yang *et al.* [20] identified these typical fuzzy composite operators and how they are selected. Basically, the former composite operator is major-factor-dominated composition, and the latter is weighted-mean composition. In addition, the latter operator is more comprehensive than the former operator is. In this paper, we prefer that the EC related statement features of a statement both contribute to the EC of the statement, and the more comprehensive information can be utilized. The features defined in Definition 7 are dominated one. Due to these reasons, M $(\bullet, \oplus)$ is chosen, and before evaluating, we must adjust the weight coefficient vector A and the relation matrix R such that:

$$\sum_{i=1}^{22} a_i = 1, \quad \sum_{j=1}^{20} r_{ij} = 1.$$

Assuming that there are 22 statements and each of them satisfies only one distinct feature, the $F(V)$ of them is shown in Figure 3. In each chart of Figure 3, the x-axis is the level of the FEC from 1L to 20L, and the y-axis is the degree of membership.

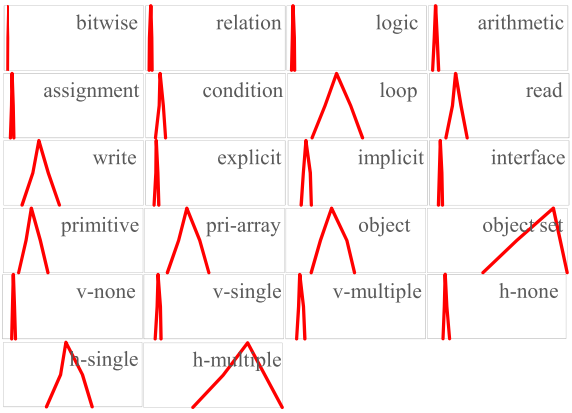

FIGURE 3. *F(V)* **of 22 distinct features.**

As described in definition 7, a statement may have many features in different feature groups, so that a benchmarked statement is selected, and each of them has a typical weight coefficient matrix A by which its FEC is evaluated. These statements are treated as criteria.

D. FUZZY PATTERN MATCH

Through the above discussion, we can use the multilevel fuzzy comprehensive evaluation to obtain the final evaluation matrix (vector) $B \in F(V)$. Then, the Hamming closeness degree is selected to recognize the weight coefficient vector of an arbitrarily given statement. Let A and A' be fuzzy sets on U, then

$$N(A, A')1 - \frac{1}{n} \sum_{i=1}^{k} |a_i - a'_i|$$

is named the Hamming closeness degree between fuzzy sets A and A'.

After all the fuzzy rule bases for all the features are deduced, we can immediately compare a new given weight coefficient vector with our fuzzy rule base to find a minimum value of the Hamming closeness degree representing that these two vectors are the most similar. In addition, we can take this corresponding EC level as the EC level of the new given statement.

V. EXPERIMENTS

In this section, we design a group of experiments to verify the basic regularities between the EC and the statements, the statement features analysis, the comparison between the EC and the FEC, and the application cases of the FEC.

A. SETUP

We perform experiments in a real environment and measure the EC of the computer during the execution of the program. The experimental environment, as shown in Table 1 and Table 2, includes the experimental computers, monitoring computer, algorithms and procedures as cases.

TABLE 1. Testbed description.

Item	Description
Computer	Tsinghua Tongfang Z900, CPU Intel i5-2300 2.80 GHz, 8GB memory, 1TB hard disk. Power is from 60 watts to 110 watts.
Operation System	CentOS 5.6, Linux 2.6.18 core, CPUFreq Governor is set to Conservative mode
Power meter	HOPI-9800, power is less than 1 watt, the sampling rate is 1 second.
Monitor Software	Using power meters, through USB, we transfer EC data that is collected by the HOPI-9800 power meter on the experimental computer to the monitoring computer.
Programming Language	Java (jdk-1.7.0)
IDE	Eclipse Oxygen
Measurement Units	EC unit: Joule (J); Time unit: Second (s);

TABLE 2. Description of test cases.

Test cases of scientific computing		
FFT	Fast Fourier transform	Modified from the cases in JVMSPEC2008.
LU	LU Decomposition	
MonteCarlo	Monte Carlo algorithm	
SOR	Successive 0ver-relaxation	
SPARSE	One product of a sparse matrix	
Test cases of arithmetic operation		
AddBench	int, long, float, double	For several data types, and operations are in nested loops.
MulBench	int, long, float, double	
DivBench	int, long, float, double	
SubBench	int, long, float, double	
Test cases of sorting computing		
BubbleSort	Bubble Sort	Sorting integer array whose size is 100, 1000 and 10000. The items are initialized in a random order.
HeapSort	Heap Sort	
MergeSort	Merge Sort	
QuickSort	Quick Sort	
InsertionSort	Insertion Sort	
SelectionSort	Selection Sort	
ShellSort	Shell Sort	
Test cases of Java collections		
Insertion	Add item	Vector, ArrayList, LinkedList, HashMap, TreeMap, HashSet, TreeSet
Find	Find item	
Deletion	Remove item	
Test cases of customized calculation and storage statements		
CalStmt	Calculate π^2, π remains 8 decimal places	
StoStmt	(Random)Access an object in large linked lists	
Test cases of Other algorithms		
Search	Linear search	
LCS	Longest common subsequence	
Floyd	Floyd–Warshall shortest path	

B. PERFORMANCE AND EC

We first discuss the fact that the execution time of a program does not correspond with its EC; thus, we cannot estimate the latter by the former. In this experiment, we proved that the relation between the code execution time and the EC is not consistent. We compared the EC of four use cases: Insertion Sort, Merge Sort, Floyd, and LCS. By adjusting the input scale to the algorithms, we ensured that the execution times of the use cases were almost equal. Comparing their ECs, we proved that ECs of the same execution time but of different algorithms are not the same. Figure 4 shows the result of this experiment.

In Figure 4, although the execution times of the four use cases were approximately the same, their EC differences were obvious. When the execution time was approximately 30 seconds, the EC of LCS was 29.9% higher than that of Floyd. Similarly, the EC of the Insertion Sort was 16.6% higher than the EC of the Merge Sort. Reducing the input scale of the four algorithms above making the execution time approximately 15 seconds, this time, EC patterns of four algorithms remained unchanged, but the EC differences changed. The EC of LCS was 22.1% higher than that of Floyd. Similarly, the EC of the Insertion Sort was 12.3% higher than that of the Merge Sort. As a result, we can conclude that first, codes with the same execution time do not necessarily have the same EC, and we cannot simply take code execution time to represent the execution EC. Second, the EC differences of different pieces of code are unstable and vary with the input scale although the algorithm execution times are the same, but the EC differences still change.

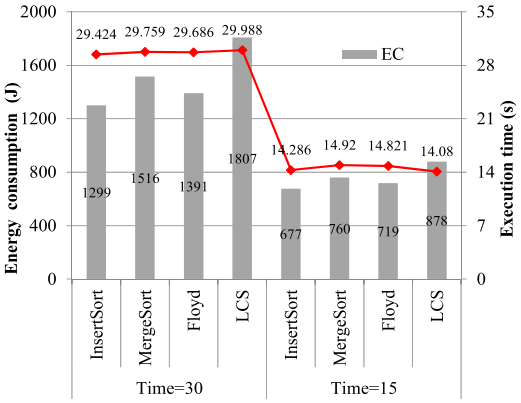

FIGURE 4. The EC comparison of different use cases when execution times are equal.

C. FEATURES ANALYSIS

The experiment analyzes the effects of statement features on the EC. By comparing the EC of the statement features, the features are abstracted properly if their EC differences are obvious. Feature are grouped as follows:

- **Operator**: bitwise, relation, logic, arithmetic, assignment, condition, loop;
- **I/O**: read, write;
- **Method**: explicit, implicit, interface;
- **Data**: primitive, primitive array, object, object set;
- **Virtual call**: none, single, multiple;
- **Inheritance**: none, single, multiple.

We analyzed the statements of the test cases in Table 2, associated them with the elements of the features mentioned above, and measured their EC consumed by the experimental computer on which the test cases were executed. Many results were collected for each feature, and the five data summaries are shown for comparison. Figure 5 shows the boxplots of these results.

Overall, the ECs of the statements with different features are distinctive. In the test cases, a statement may match several features in different groups, so that the EC regularity of each feature may overlap. For example, a *read*(I/O group) statement may also be an *object* (data group) statement. However, due to the larger number of statements that were

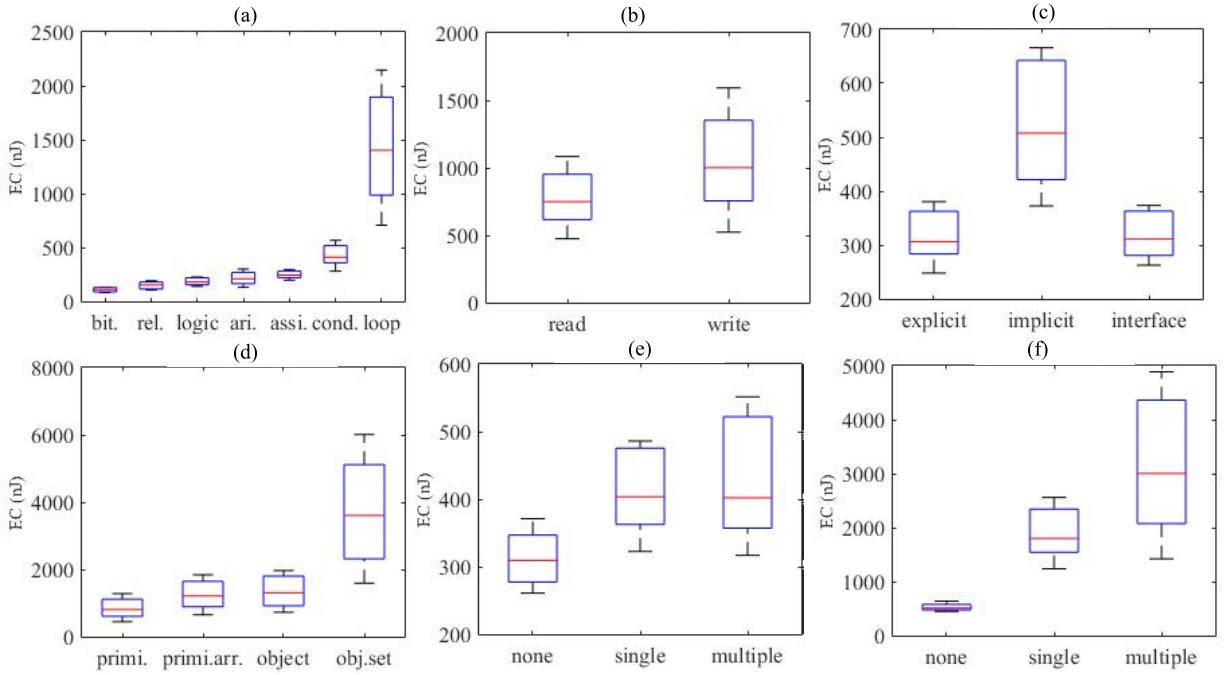

FIGURE 5. The boxplots of the statement EC with different feature values. (a) Operator. (b) I/O. (c) Method. (d) Data. (e) Virtual call. (f) Inheritance.

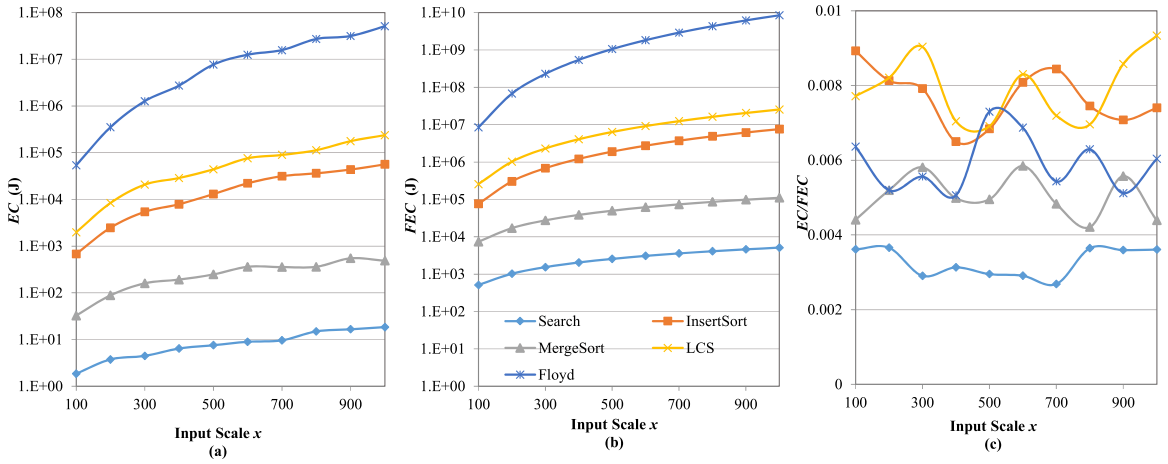

FIGURE 6. In different algorithms and input scales, the values of EC, FEC and EC/FEC.

investigated, such overlap is offset by the data summarization. For abbreviation, the details of each feature are not analyzed in this section. Notice that in Figure 5 the scales of the y-axis in different charts are not the same. For example, the difference between *none*-virtual-call and *single*-virtual-call, as shown in Figure 5-(b), looks similar to that between *read*-I/O and *write*-I/O, as shown in Figure 5-(e). However, the former (approximately 500 *n*J) is larger than the latter (approximately 10 *n*J).

D. FEC AND EC

To verify whether the FEC is consistent with the EC, we compared the measured ECs and estimated the FECs of five cases when the input scale increased. The FEC represents the EC of the code correctly if the tendency of the FEC and the EC were consistent, and the ratios of the EC and the FEC were relatively stable. Search, Insert Sort, Merge Sort, LCS, and Floyd were selected as the test cases, and their EC, FEC, and EC/FEC values were compared with different input scales (number of data items $x = 100, 200, 300, \ldots, 1000$). We expected that the values of the EC/FEC of the same case under different scales would be approximately equal, or their variances would be small.

Comparing Figure 6-(a) and 6-(b) (logarithmic coordinates), the tendencies of both the EC and FEC values for the five cases were consistent[1] regardless that their values were

[1]In fact, measured EC fluctuates, but the fluctuations are not obvious on the logarithmic coordinates, while the FEC curve corresponds with the function curve.

not equal. The EC and FEC represented the differences of five algorithms consistently. In Figure 6-(c), the curves are almost stable. The mean values of EC/FEC for Search, Insert Sort, Merge Sort, LCS, Floyd are 0.0032, 0.0077, 0.0050, 0.0078 and 0.0059, 0.0059, respectively, and the standard deviation of them is 0.00036, 0.00071, 0.00055, 0.00084, and 0.00073. The experimental results show that the proposed FEC is relatively accurate and not only statically estimates but also compares the EC of the algorithms.

E. APPLICATION CASES

In these experiments, we apply the FEC in two groups of test cases, i.e., sort algorithms and Java collections, to verify the accuracy of the estimation; the selected test cases have a common ground, that is, the statement complexity is increased with the input scale, but such complexity is difficult to be determined by code statistical analysis. Thus, FEC may be far from the actual EC.

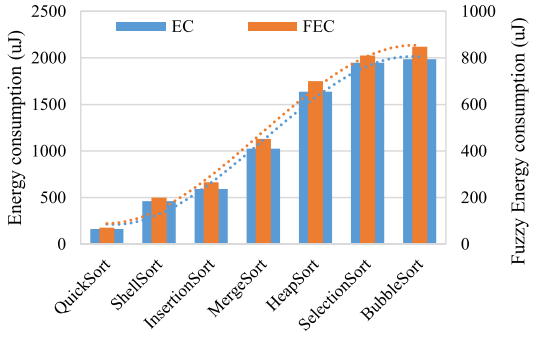

FIGURE 7. The comparison of the real and estimated EC values of the sort algorithm.

Figure 7 shows a perfect match of the FEC and the EC under the various sorting algorithms while sorting a small number of data items (100 randomly ordered items). The primary y-axis is the EC (wide bars), and the secondary y-axis is FEC (narrow bars), since the scale of the two axes, as well as the values of the FEC and EC, are different, but their tendencies[2] are almost the same.

We adopt a loop-unrolling-similar technique [20] to deal with the input scales of the sorting algorithm. Take the Merge Sort as an example, we infer the executed times of the loops approximately and estimate the FEC on the loop-unrolled code. This is why the FEC and EC are well matched. However, it is difficult to statically infer the executed times of the loop so that errors are unavoidable. However, the sorting algorithms have relatively uncomplex code, and the input scale is also small; therefore, the errors are not obvious.

Next, we adopt the relatively complex algorithms with a larger scale input. The test cases are Insertion, Find and Deletion operations on Java collections, including *Vector*, *ArrayList*, *LinkedList*, *HashMap*, *TreeMap*, *HashSet* and *TreeSet*. The initial items in the collections are 10000.

[2]polynomial regression, from energy efficient algorithm to energy consumed algorithm

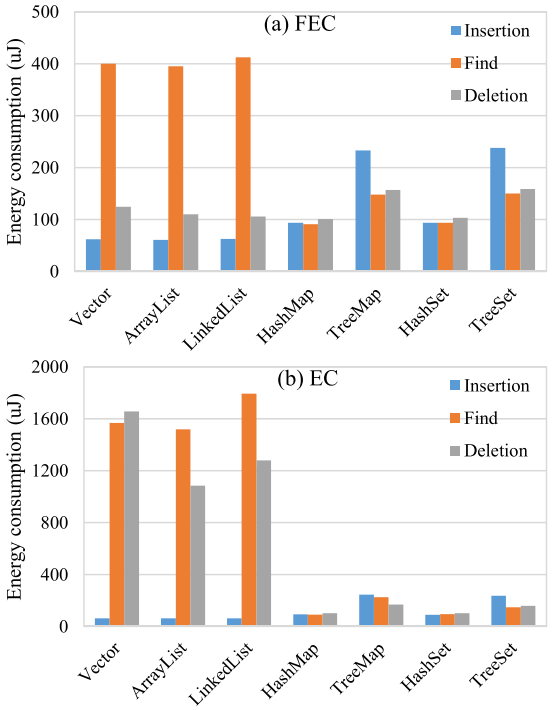

FIGURE 8. The test results of the EC of seven sets in the Java language.

The EC and the FEC of the three operations on 7 collections are shown in Figure 8. In the paper, the advantages or weaknesses of these collections are not analyzed, but the estimation accuracy of the FEC should be highlighted. Comparing Figure 8-(a) and Figure 8-(b), the FEC is not accurate enough under the given experiment. The FECs, regardless as to whether they are larger or smaller, are almost similar to the ECs, but their tendencies are inconsistent. The primary errors are at the Find and Deletion operations on *Vector*, *ArrayList* and *LinkedList* because the performance of the three operations is highly related to the input scales, i.e., more elements in the collections results in more complexity in finding the special element. The loop-unrolling-estimation mentioned above does not benefit the estimation because it is impossible to statistically infer when the loop for the collection traversal is exited. In contrast, there is not such an issue in the look-up-based collections such as *HashMap* and *HashSet* (hash table), as well as *TreeMap* and *TreeSet* (red-black tree).

Likewise, there is a collection traversal when inserting an element to the *TreeMap* and *TreeSet*, so that the FECs of the Insertion operations on *TreeMap* and *TreeSet* are relatively higher than others. However, the FECs of these two are proportionally underestimated. Comparing Figure 8-(a) and Figure 8-(b), such error is not obvious because it is hidden by the extremely underestimated FECs of the *Find* and *Deletion* operations on the preview of the three collections.

In conclusion, since the accuracy of the FEC is perfect in some situations, it is still a statistical estimation and cannot forecast the complex runtime situation. Therefore, the error is unavoidable. However, the intentions of the FEC and the

statistical estimation are for locating the energy inefficient statements, and further improving them, but not the replacement of the measured approaches. From this point of view, the errors are acceptable.

VI. CONCLUSIONS AND FUTURE WORK

This paper proposes a static EC estimation model and approach for the OO code. First, the fuzzy EC model, named the FEC, is proposed as a simplified sources-level EC model. Then, the EC related features of the statements are defined. Based on the model, the fuzzy estimation approach is introduced to quantify the effects of the statement features on the EC. Finally, the effectiveness of the FEC is verified through the designed experiments. In conclusion, the FEC has the following advantages:

(1) Independence: It does not depend on the compiling environments and runtime environments.
(2) Rich features of statements: The energy-related statement features are well abstracted and modeled.
(3) Static estimation: Compiling or executing the procedure is unnecessary. Instead, it can measure EC only by analyzing the source code.
(4) Reasonable accuracy: The accuracy requirement is appropriately relaxed by considering the consistency of the estimated values and the actual values as well as the variation trend.
(5) Fairness: The runtime environment does not affect the fairness of the FEC. In the same context, the differences of the FEC among tasks should be almost consistent with the differences of the actual EC, or they should satisfy a stable ratio.
(6) Object-Oriented: the estimation model takes the features of object-oriented programming languages into consideration.

However, as a static EC estimation model of code, the FEC does not fully consider the relationship between statements, i.e., the code structure, which will be extended in our future work.

REFERENCES

[1] M. Poess and R. O. Nambiar, "Energy cost, the key challenge of today's data centers: A power consumption analysis of TPC-C results," *Proc. VLDB Endowment*, vol. 1, no. 2, pp. 1229–1240, 2008.
[2] N. Amsel, Z. Ibrahim, A. Malik, and B. Tomlinson, "Toward sustainable software engineering: NIER track," in *Proc. 33rd Int. Conf. Softw. Eng. (ICSE)*, May 2011, pp. 976–979.
[3] M. R. Sabharwal, "Software power optimization: Analysis and optimization for energy-efficient software," in *Proc. Int. Symp. Low Power Electron. Design (ISLPED)*, Aug. 2011, pp. 63–64.
[4] D. Hackenberg, R. Schöne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer, "An energy efficiency feature survey of the intel haswell processor," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshop (IPDPSW)*, May 2015, pp. 896–904.
[5] M. Bazzaz, M. Salehi, and A. Ejlali, "An accurate instruction-level energy estimation model and tool for embedded systems," *IEEE Trans. Instrum. Meas.*, vol. 62, no. 7, pp. 1927–1934, Jul. 2013.
[6] A. Noureddine, R. Rouvoy, and L. Seinturier, "Unit testing of energy consumption of software libraries," in *Proc. 29th Annu. ACM Symp. Appl. Comput.*, 2014, pp. 1200–1205.
[7] G. Da Costa and H. Hlavacs, "Methodology of measurement for energy consumption of applications," in *Proc. GRID*, 2010, pp. 290–297.
[8] V. K. Singh, K. Dutta, and D. VanderMeer, "Estimating the energy consumption of executing software processes," in *Proc. IEEE Int. Conf. IEEE Cyber, Phys. Social Comput. Green Comput. Commun. (GreenCom)*, Aug. 2013, pp. 94–101.
[9] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier, "A preliminary study of the impact of software engineering on greenit," in *Proc. 1st Int. Workshop Green Sustain. Softw.*, 2012, pp. 21–27.
[10] S. Schubert, D. Kostic, W. Zwaenepoel, and K. G. Shin, "Profiling software for energy consumption," in *Proc. IEEE Int. Conf. Green Comput. Commun. (GreenCom)*, Nov. 2012, pp. 515–522.
[11] A. Noureddine, R. Rouvoy, and L. Seinturier, "Monitoring energy hotspots in software," *Autom. Softw. Eng.*, vol. 22, no. 3, pp. 291–332, 2015.
[12] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier, "Runtime monitoring of software energy hotspots," in *Proc. 27th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2012, pp. 160–169.
[13] C. Brandolese, "Source-level estimation of energy consumption and execution time of embedded software," in *Proc. 11th EUROMICRO Conf. Digit. Syst. Design Archit., Methods Tools (DSD)*, Sep. 2008, pp. 115–123.
[14] X. Zhou, B. Guo, Y. Shen, and Q. Li, "Design and implementation of an improved C source-code level program energy model," in *Proc. Int. Conf. Embedded Softw. Syst. (ICESS)*, May 2009, pp. 490–495.
[15] C. Seo, G. Edwards, S. Malek, and N. Medvidovic, "A framework for estimating the impact of a distributed software system's architectural style on its energy consumption," in *Proc. 7th Workshop IEEE/IFIP Conf. Softw. Archit. (WICSA)*, Feb. 2008, pp. 277–280.
[16] C. Seo, S. Malek, and N. Medvidovic, "Component-level energy consumption estimation for distributed java-based software systems," in *Proc. Int. Symp. Component-Based Softw. Eng.* Berlin, Germany: Springer, 2008, pp. 97–113.
[17] P. Heinrich, H. Bergler, and D. Eilers, "Energy consumption estimation of software components based on program flowcharts," in *Proc. IEEE Int. Conf. High Perform. Comput. Commun., IEEE 6th Int. Symp. Cyberspace Saf. Secur. IEEE 11th Int. Conf Embedded Softw. Syst (HPCC, CSS, ICESS)*, Aug. 2014, pp. 542–545.
[18] P. Heinrich, H. Bergler, and E. Oswald, "Early energy estimation of networked embedded systems executing concurrent software components," *Int. J. Model. Optim.*, vol. 5, no. 2, p. 119, 2015.
[19] E. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer, "Post-compiler software optimization for reducing energy," *ACM SIGARCH Comput. Archit. News*, vol. 42, no. 1, pp. 639–652, 2014.
[20] H. Yang, G. Gao, A. Marquez, G. Cai, and Z. Hu, "Power and energy impact of loop transformations," in *Proc. Workshop Compil. Operating Syst. Low Power (COLP)*, Sep. 2001.

HUI LIU received the Ph.D. degree from Northeastern University, Shenyang, China, in 2010. She is currently a Lecturer with the School of Metallurgy, Northeastern University. Her current research interests include green computing, sustainable computing, and computational fluids dynamics.

FUSHENG YAN received the Ph.D. degree from McMaster University, Hamilton, Canada, in 2006. He is currently a Professor with the School of Metallurgy, Northeastern University. His research interests include computational fluids dynamics and green industry.

JINGQING JIANG received the Ph.D. degree from Jilin University, Changchun, China, in 1990. She is currently a Professor with the College of Computer Science and Technology, Inner Mongolia University for the Nationalities. Her research interests include green computing.

JIE SONG received the Ph.D. degree from Northeastern University, Shenyang, China, in 2008. He is currently an Associate Professor with the Software College, Northeastern University. His research interests include green computing, big data management, and machine learning.