IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 29, 2017, accepted November 19, 2017, date of publication November 29, 2017, date of current version February 14, 2018.

Digital Object Identifier 10.1109/ACCESS.2017.2778309

Source-Level Energy Consumption Estimation for Cloud Computing Tasks

HUI LIU[1], FUSHENG YAN[1], SHAOKUI ZHANG[2], TAO XIAO[3], AND JIE SONG[2]

[1]School of Metallurgy, Northeastern University, Shenyang 110819, China
[2]Software College, Northeastern University, Shenyang 110819, China
[3]School of Computer Science and Engineering, Northeastern University, Shenyang 110819, China

Corresponding author: Fusheng Yan (yanfs@smm.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 61402090, and in part by the National Research Foundation for the Doctoral Program of Higher Education of China under Grant N161602003.

ABSTRACT In the cloud computing environment, the source-level energy consumption (EC) estimation is employed to approximately measure the EC of a cloud computing task before it is executed. The EC estimation on tasks is critical to task scheduling and source-code improvement in the aspect of EC optimization. The existing studies treat a task as a program, and EC of the task as the simple summation of each statement's EC. However, EC of two tasks consisting of the same statements with different structures is unequal; therefore, the code structure should be highlighted in source-level EC estimation. In this paper, an abstract energy consumption (AEC) model, which is static and runtime-independent, is proposed. For the model, the two quantitative measurements, ''cross-degree'' and ''reuse-degree,'' are proposed as the code structure features, and the relationship between EC and the measurements is formulated. Although AEC is not a precise EC measurement, it can properly represent the EC of a task, compare with other tasks, and verify the optimization effect. Experimental results show that the ratios between the EC and AEC with 50 test cases are stable; the standard deviation is 0.0002; and the mean value is 0.005. The regularities of EC and code structures, represented as ''cross-degree'' and ''reuse-degree,'' are also validated. Though AEC, it is easier to schedule the cloud computing tasks properly and further reduce the consumed energy.

INDEX TERMS Abstract energy consumption, code structure, energy consumption estimation, cloud computing tasks, source-level.

I. INTRODUCTION

Energy Consumption (EC) is a critical concern for the IT industry especially in the field of cloud computing. Leveraging the cloud to maintain or scale up business will be the norm for customers and enterprises while cutting down on the budget. However, EC of the cloud is predicted to increase rapidly in the next decade [1]. Therefore, the the hardware level, the early technologies on energy saving are developed on circuit [2], component [3] and architecture; At the infrastructure level, the energy-efficient scheduling algorithms and memory systems, storage systems as well as the resources management policies are also helpful to EC reduction; At the platform level, there are many energy-efficient middlewares which can not only improve the energy-efficiency greatly but also support the green applications such as mobile ones [4]. In this paper, the platform level of cloud computing is studied, where the EC of tasks is estimated.

At the platform level, the cloud computing tasks, such as data processing task, Web searching tasks, and scientific calculation tasks, are scheduled to the nodes of cloud data center and allocated with the proper resources. They compete for multiple resources available on nodes while they are in execution, and the always-on components of computer will waste the energy while they are idle. The main objectives of the task scheduling and resource allocation algorithms are to maximize the resource utilization and minimize the waiting time of resources in cloud data center. Otherwise, unbalanced task scheduling and resource allocation will cause the idleness of nodes and the waste of energy [5].

The traditional task-oriented EC optimizations are mainly applied to embedded software or mobile applications, and most of them run on battery-powered hardware [6]. On the energy restriction environment, the EC optimization could efficiently extend the service time of a system [7]. In recent years, with the popularization of cloud computing technologies, tasks oriented EC optimizations are gradually applied to the more general areas, especially those require distributed frameworks and larger-scale clusters [8]. It is

generally believed that the scheduling algorithms would be more efficient if the EC of scheduled tasks are pre-known. Thus, the EC estimation of tasks before they are executed is necessary.

The task scheduling and resource allocation should be triggered prior to the task execution, however, it is a challenge to estimate task EC without its execution.. To solve the problem, we propose a source-level EC estimation for cloud computing tasks. No matter what kinds of tasks, they all consist of source codes. Despite of differences on functions and runtime environments, the source-level EC estimation has better applicability [9]. Source codes consist of statements and the relations among the statements, while the latter is called code structure. To estimate EC, both statements and code structures should be considered by the following three principles:

(1) Complex operations and data structures cost more energy than their simple ones,.
(2) Among the tasks with the same calculation workload, the one which can maximize the CPU utilization and reduce CPU idle time has lower EC.
(3) Among the tasks with the same calculation workload, the one which prefers to use high performance storage has lower EC.

Among these principles, principle (1) relates to statements themselves, while principle (2) and (3) relate to code structures. Due to the effects of code structures, the EC of a task is not equal to the simple summation of each statement's EC. The existing researches suggest to estimate tasks' EC by source-level analysis. However, most of these estimation approaches emphasize on specific languages, code structures and scenarios. According to the previous descriptions, we may raise the following questions:

(1) What are general features of EC-related code structures.
(2) How to quantify the code structure features, as well as the relations between features and EC of tasks.
(3) Without taking the runtime environments into consideration, EC of a task is not a measured value but an estimated one. Then, how to estimate it through the statement features and structures features of source codes and whether the estimated EC is consistent with actual EC remain a problem

To our best knowledge, though several researches as explained in section 2 focus on the similar topics, but they do not solve the questions well.

In this paper, we study the source-level EC estimation which highlights the code structure features. Firstly, the Abstract Energy Consumption (AEC) is proposed, offering a static and source-level EC model. Secondly, based on the model, the cross-degree and reuse-degree are employed to quantify the effects of code structures on EC, and such effects are aggregated for source-level EC estimation. Referring to Principle (2), the cross-degree represents the interleaving of storage statements and calculation statements; and referring to Principle (3), the reuse-degree represents the possibility of reusing caches. Finally, the estimation is verified to be effective through the elaborated experiments. Our contributions are listed as follows:

(1) The two general, abstract and quantitative measurements, named as reuse-degree and cross-degree are proposed to represent the EC-related features of code structure.
(2) The relationship between EC and above measurements is concluded.
(3) The source-level EC estimation in reasonable abstraction and approximation is proposed.

The rest of paper is organized as follows: Following the introduction, section 2 introduces the related works; Section 3 introduces the source-level AEC model; Section 4 explains the EC-related features of code structure; Section 5 discusses the relationship between code structures and EC of tasks, and also derives the quantified estimation of AEC; In Section 6, plenty of experiments and results show the effectiveness of AEC estimation; Finally, conclusions and future works are summarized in Section 7.

II. RELATED WORKS

Source codes oriented EC estimation is a hot research topic. Some researchers believe that the compiling processes are too complex and the effects of instructions' execution sequence on EC are unpredictable. As a result, it is extremely difficult to analyze EC at source-level. However, many existing research results, as introduced in this section, have shown that the source-level EC estimation is feasible and effective.

Brandolese [10] built a ParseTree by source codes analysis. In ParseTree, each node is an atomic unit which consumes energy, and edges are the combinations of the units. ParseTree represents the structures and statement features of source code. To estimate EC, Brandolese proposed the instructions-level EC estimation approach, and designed a compiling and runtime environment dependent engine, by which EC of a program is consolidated line-by-line and function-by-function. Zhou *et al.* [11] proposed a C language based EC model. They considered that there are 3 steps in an execution process of instruction: acquiring, decoding and executing. Thus, code EC is NOT equivalent to the summation of instructions' EC, but the summation of EC of all the three steps. Additionally, some researches measured EC by executing the program, and proposed runtime EC profiling tools [12]. The profiling tools collected the states of computer components, such as CPU and memory, to deduce the EC of program [13]. Schubert *et al.* [14] designed Eprof tool which estimated EC precisely. It helps developers to locate high EC statements. They considered that traditional CPU-oriented EC estimation did not consider accessories, such as hard drivers and network devices, which consume energy as well. They pointed out that a program accesses CPU synchronously, so that statements execution and CPU EC are synchronous as well. On the contrary, the program may access accessories asynchronously, so that statements execution and accessory EC are asynchronous as well. Eprof focuses on both CPU synchronous EC and accessory

asynchronous EC. Developers could choose between CPU-depended codes and I/O-depended codes. For example, when transmitting data, developers decide whether data is compressed (CPU-depended) or not (I/O-depended). Noureddine *et al.* [15] argued that traditional solutions focused on coarse-granularity approaches in order to monitor EC of devices and processes. By contrast, they proposed a fine-granularity runtime EC monitoring framework in order to locate the higher EC code-blocks in source codes. The framework includes two levels monitoring components: OS-level (OS, operating system) and process-level energy monitor. The former focuses on monitoring hardware, while the latter focuses on monitoring execution of Java codes. The results from two monitors are consistent and consolidated. Noureddine's other works [16] are also concerned in this area.

Despite of instructions-level or source-level approaches, it is impossible to estimate EC of a program precisely without executing it. Because the logical structures of a program are complex, and it is difficult to determine its execution paths exactly, such as conditional branches, execution times of loops, scale of data structures, etc. Hence, power meter is a better solution for measuring EC precisely. The source-level EC estimation, which fully considering code structures, is NOT to substitute for approaches of measuring EC by instruments, but to locate high EC statements, optimize EC, or evaluate effects of EC optimization. The source-level EC estimation approaches as mentioned above are for special cases, they are not much abstract and universal, and they did not conclude code structure features to the general and quantitive models. That is why we propose a code structure highlighted, universal, coarser-grained and source-level EC estimation in this paper

III. ENERGY CONSUMPTION MODEL

A task, represented as 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 executions of all statements, or statically analyze EC of all statements exhaustively. As a solution, we define the concept of Abstract Energy Consumption (AEC).

Definition 1 (Abstract Energy Consumption (AEC)): The runtime EC of a task is the amount of energy consumed by the hardware during the execution of task. Runtime EC can be measured by equipments. In contrast, the AEC of a task is the amount of consumed energy represented by the static features of source codes. It is independent from hardware and runtime environments, so that it may differ from, but should accord with the runtime EC. AEC is estimated through the static analysis on code, with joule as the unit.

AEC is a theoretical EC quantification, so it ignores the runtime situations and environment. The task EC is analogous to the task performance. Runtime performance of a task is highly coupled with the input scale, data distribution and hardware environments, while the static optimization tools for tasks, such as *pclint*, can evaluate and optimize abstract performance according to the code features. The abstract performance is especially important in programming because it can help programmer avoid the codes with proper-function but low-performance. In the same way, the proposed AEC is also a static measurement. The AEC is not a precise EC measurement. Also, AEC and runtime EC are different, of course. However, AEC has four advantages: Frist, it is simple enough to be estimated by the static analysis; second, it is effective to represent the differences of consumed energy among tasks; third, it is beneficial to the task scheduling and resource allocation approaches; fourth, it highlights the EC-related features of source codes.

For a task, it is well known that the relation between the execution time and the execution EC should follow the physical theorem as it states that EC equals to time multiplied by power. This theorem can be applied to any runtime environment, as shown in Equation (1):

$$E = power(W) \times T \tag{1}$$

Where the symbol W represents the parameters set which has effects on computer power; the symbol T means execution time of a task; the $power()$ represents the function between computer powers and characteristic parameters. Computer real-time power shifts dynamically. It depends on whether the computer is busy or not. The computer's power is relatively low when it is idle, and vice versa. In the same way, the values of W also varies with time. So that Equation (1) is a proper approximation. The symbol W contains the representative values of characteristic parameters, and $power(W)$ is also the average value.

According to Equation (1), we define AEC of a task as:

$$AEC = power(V) \times t(n) \tag{2}$$

In Equation (2):

(1) The symbol n is the number of executing statements. It relates to the scale of task and input. If the input scale is 1, then n represents the number of statements in source code. Fox example, in section 6 the linear-search task has three statements, then $n = 3x$ if input scale is x.

(2) The function $t(n)$ means the estimated execution time of n statements. $t(n)$ is a monotonic increasing function with n. Different statement has different execution time, but we treat them as the same. For easy derivation, we deem that $t(n) \approx n$.

(3) The set V is a subset of W. V contains parameters which relates to both code execution and computer power. However, it ignores the static features of hardware.

The relation between computer power and CPU frequency is $P = P_{fix} + P_f \times f^3$. Where P_{fix} is a constant, representing powers of other devices except CPU; P_f is CPU power coefficient, and f is CPU frequency [17]; In the previous study, computer power relates to CPU utilization ω and CPU frequency f, as the function $pcpu(f, \omega) = a_1 f^3 + a_2 \omega f^3 + a_3 \omega$, where a_1, a_2, a_3 are all hardware specified coefficients. Most of researches assume that power of computer components, except for CPU, are stable [18], also others consider that working states of computer CPU, motherboard, memory

and hard disk are all typical features relating to computer power [19].

The power of common DDR3 (8GB) memory fluctuates from 4 to 8 watt. There are two working states: idle state (no data access) and loaded state (data access). The former one only consumes the minimum energy for maintaining data in memory, while the latter one requires extra energy to support reading or writing data. Thus, the memory power includes I/O power when reading or writing data, and storage power when just storing data. I/O power is twice of storage power, which can be explained as that the power of working state (accessing memory) is twice of that of idle state (only store data) [20]. We assume idle state power is p, idleness (%) is d, busy-rate (%) is 1-d, so $pmem(d) = p \times d + 1.5 \times p \times (1 - d) = 1.5 \times p - 0.5 \times p \times d = a_4 d + a_5$, where a_4, a_5 are all hardware specified coefficients.

The power of a hard disk is about 8 watt. A hard disk consumes energy in three aspects: EC of integrated circuits and chips; EC to support platters spinning; and EC to move actuator arm and position read/write head. The former two account for 90% of the hard disk EC. When the disk is idle (no read/write), only the latter one is saved [21]. As a result, power of a hard disk is almost independent from its working states, together with constant power of other devices, we define them as constant *pother*.

According to the analysis above, $V = \{f, \omega, d\}$, f is CPU frequency, ω is CPU utilization, d is memory utilization. The following sections discuss the effects of code structures on f, ω, d and $t(n)$. Therefore:

$$\begin{aligned} AEC &= [pcpu(f, \omega) + pmem(d) + pother] \times t(n) \\ pcpu(f, \omega) &= a_1 f^3 + a_2 \omega f^3 + a_3 \omega \\ pmem(d) &= a_4 d + a_5 \\ pother &= a_6 \end{aligned} \tag{3}$$

IV. STRUCTURE MODEL

As explained before, EC of a task is not the simple summation of that of its statements [22]. Statements executed in different orders consumes different amount of energy. For source-level estimation, the logical relations among statements, named as code structures, should be modeled. In this section, we propose two structural features for the code structures, and the quantitative relationship between features and EC will be derived in the next section. We firstly define the statement model because code structure features involve statement features.

Definition 2 (Calculation Statements and Storage Statements): A statement is the smallest standalone element of a task. In Equation (3), the AEC relates to the working state of CPU and memory, so that the statements are classified into CPU-depended and I/O-depended. Calculation statements, including the statements of logical operation, arithmetic operation and control flow, are CPU-depended; Storage statements, including the statements of creating, retrieving, iterating, searching, modifying, and destroying (deleting) date structures, are I/O-depended.

A. CROSS-DEGREE

The AEC of a task not only relates to the numbers of calculation statements and storage statements, but also relates to the interleaving frequency between two groups. Ignoring the concrete features of statements, if a storage (calculation) statement is adjacently followed by a calculation (storage) statement, then we consider the two statements have a interleaving (cross), otherwise they are independent. We define cross-degree as the interleaving frequency of statements.

Definition3 (Cross-Degree): Cross-degree $r(n)$ of a task is the frequency of the storage statements and the calculation statements interleaving with each other. We assume that the executing statements of a task is as a sequence $< I_1, U_1, I_2, U_2, \ldots, I_r, U_r >$. In which U_i and I_i are the set of calculation statements and storage statement, respectively, the suffix i is the order number, and the number of sub-sequences $< I_i, U_i >$ is r ($r \geq 1$). If there are n executing statements of the task, interleaving frequency is r/n, then cross-degree $r(n) = r/n$.

Fig. 1 shows examples of cross-degree and statement sequence.

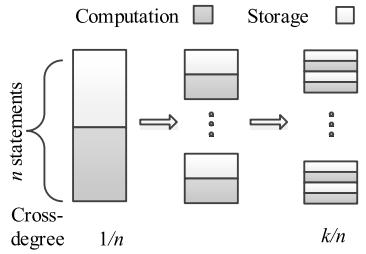

FIGURE 1. Example of Cross-degree.

Different cross-degree result in different arrangements of statements, what is more, the cross-degree has effects on EC of the task. The reasons are as follows: CPU power correlates positively with CPU frequency. According to CPU Frequency Scaling[1] and CPU Frequency Governor[2] techniques, CPU actively decreases its frequency in order to save energy when its utilization is lower. Consequently, on one hand, the cross-degree is lower when sequential storage statements are executed, and CPU is continuously idle thus Scaling technique is activated, as a result, CPU keeps in low-frequency and low-power mode; On the other hand, the cross-degree is higher when storage statements and calculation statements are alternatively executed one by another, and CPU is continuously busy (idle time slots are very short). Thus, scaling technique hardly works,[3] as a result, CPU keeps in high-frequency and high-power mode.

CPU power correlates positively with CPU utilization. For example, a multi-core processor partly closes its cores and its EC decreases linearly when CPU utilization is low.

[1]https://en.wikipedia.org/wiki/Frequency_scaling

[2]https://en.wikipedia.org/wiki/Governor_%28device%29#Computing, 2016.

[3]Linux provides CPU Frequency Governor technology, sample interval of CPU utilization rate is at millisecond level.

However, these features only relate to the number of storage statements, rather than their execution order. In other words, code transformation may changes the cross-degree rather than the number of calculation statements and storage statements, therefore, the overall workload of CPU is fixed while only the temporal distribution of workload is updated.

B. REUSE-DEGREE

Locality is a term for the phenomenon in which the same values, or related storage locations, are frequently accessed. Temporal locality refers to the reuse of specific data, and/or resources, within a relatively small time duration. Spatial locality refers to the use of data elements within relatively close storage locations. The layered storages, from the fastest one to the slowest one, is as "register → cache ($L_1 - L_3$) → primary memory → hard disk → remote memory". Level by level, the I/O performance and price decrease, while the capacity increases. When a locality-optimized task is running, most data is accessed in cache while stored in hard disk, it takes both advantages of higher level and lower level storages. Statistics show that hit rate of cache can be up to 90%.

Locality has effects on EC of a task. Lower locality means higher cache-miss rate. On one hand, cache-misses increase the utilization of the memory, as explained previously, the power of memory in loaded state doubles than that in idle state. On the other hand, cache-misses break down the I/O performance and increase the possibility of CPU await state, while the CPU's idleness wastes energy.

Let H be the cache size. The data is stored on disks but is accessed via the cache. Cache uses queue (first-in-first-out) to manage data. When a CPU tries to access data, it firstly checks whether the required data exists in the cache. If so, cache hits, the data is accessed directly from the cache, and the data is moved to the tail of the queue. Otherwise, cache misses, CPU accesses memory to retrieve the data, remove the head of the queue, and put the data in the tail of the queue. Therefore, whether the cache-miss occurs or not depends on the "interval" between the current access and the previous access. If there are too many other data been accessed in the interval, then the data item is replaced and cache-miss occurs. The "interval" , defined as reuse-distance, is not measured by duration, but by size of distinct data item be accessed.

Definition 4 (Reuse-Distance): Assume that $\{x_t | t = 0, 1, 2 \ldots\}$ is an execution sequence of storage statements. The symbol t represents the logical clock. $\forall i < j$, x_i happens before x_j. $A(x_t)$ is the data item been accessed by statement x_t. $|A(x_t)|$ is data size of $A(x_t)$. The logic $\oplus$ operator is as: If $A(x_i) = A(x_j)$, $A(x_i) \oplus A(x_j) = |A(x_i)|$; If $A(x_i) \neq A(x_j)$, $A(x_i) \oplus A(x_j) = |A(x_i)| + |A(x_j)|$. Then, if $A(x_i) = A(x_j)$, and $\forall k \in (i, j)$, $A(x_k) \neq A(x_i)$, let reuse-distance of x_j be $d(x_j)$, or denoted as d_j, then $d_j = A(x_{i+1}) \oplus A(x_{i+2}) \oplus \ldots \oplus A(x_{j-1})$. For estimating easily, assuming the size of each data item is equal and unity, then $d_j = count(distinct(A(x_{i+1}), A(x_{i+2}), \ldots, A(x_{j-1})))$. If $!\exists A(x_i) = A(x_j)$, $d_j = +\infty$.

If the reuse-distance is less than H (the size of a cache), then the cache hits, and vice versa. However, reuse-distance is theoretical and cannot be calculated by statically analyzing source codes because the sizes of data items cannot be accurately estimated. As a result, we assuming the sizes of data item are equal in definition 4.

Definition 5 (Reuse-Degree): Reuse-degree $u(n)$ is defined as the possibility of cache-hit when n storage statements are executing. In a task containing n storage statements, the function $h(i)$ returns the number of statements whose reuse-distance is i, and H represents the maximum number of the data items in cache; then reuse-degree $u(n)$ is :

$$u(n) = \frac{1}{n} \int_0^H h(x)dx \tag{4}$$

Reuse-distance histogram can be drawn according to the function $h()$. For any storage statement x_i in this function, the possibility of reuse-degree d_i being equal to d is $h(d)/n$. Cache-miss can be predicted through reuse-distance, it happens when $d_j > H$. From Fig. 2-(a), the ratio of "the area of curve h with reuse-distance lower than H (cache size)" to "n" is the cache hit rate.

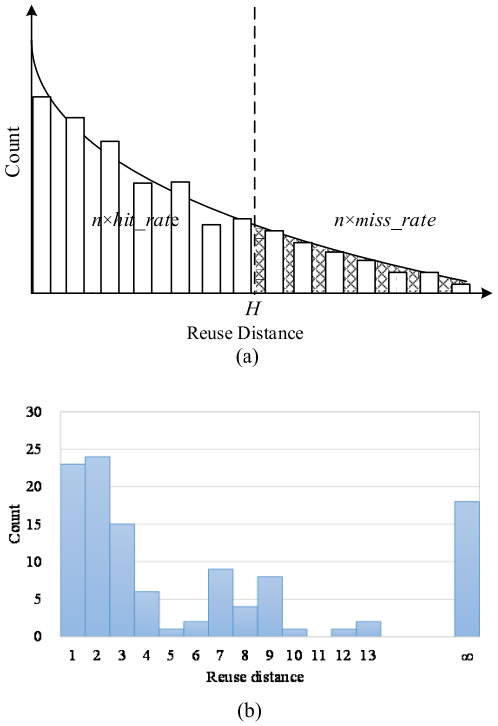

FIGURE 2. Relationship among Reuse Distance, Cache Size *H* and Cache Missing Rate.

In theory, we could statically analyze code, count sequences of storage statements, calculate the distribution of reuse-distance, deduce the h curve by mathematical approaches, and then calculate reuse-degree according to Equation (4). In practice, "h curve" is not a continuous curve but a distribution histogram. Fig. 2-(b) is the h curve concluded from a real case. We calculate the reuse-degree by

accumulating the discrete frequencies, such as Equation (5). In order to be consistent with the definition of cross-degree ($r(n) = r/n$), let $u = \sum_{i=o}^{H} h(i)$, $u(n) = u/n$.

$$\text{Let } u = \sum_{i=0}^{H} h(i) \quad u(n) = \frac{1}{n}\sum_{i=0}^{H} h(i) = u/n \tag{5}$$

V. REGULARITY

In this section, we will, according to Equation (3), derive the quantitative relationship between AEC and the cross-degree as well as the reuse-degree. We also propose the quantified estimation of AEC. Table 1 lists the major notations and their meanings

TABLE 1. Description of notation.

Item	Notation
AEC of a task	AEC
Execution time of a task	$t(n)\approx n$
Cross-degree	$r(n)=r/n$
Reuse-degree	$u(n)=k/n$
The parameters set effecting computer power	V
Computer power	$power(V)$
CPU frequency and utilization	f, ω
Memory utilization	d
The functional relation between frequency f and utilization ω	$f=scal(\omega)$
The functional relation between cross-degree $r(n)$ and CPU utilization ω	$\omega=idel(r(n))$
The functional relation between reuse-degree $u(n)$ and memory utilization d	$d=miss(u(n))$

Given a task, AEC is derived according to the equation $AEC = power(V) \times t(n) = p(r(n), u(n)) \times t(n)$. In the equation, $t(n)$, $r(n)$, and $u(n)$ represent the execution time, cross-degree and reuse-degree, respectively. They are all known functions. As a result, the key to the derivation is to determine the expression of $power(V)$.

Firstly, as mentioned in section 3, Equation (6) shows the $power(V)$ as:

$$\begin{aligned} power(V) &= pcpu(f, \omega) + pmem(d) + pother \\ &= a_1 f^3 + a_2 \omega f^3 + a_3\omega_+ a_4 d + a_5 + a_6 \end{aligned} \tag{6}$$

In Equation (6), f is CPU frequency, ω is CPU utilization, and $a_1 \sim a_6$ are all hardware-specific coefficients. Constant a_6 represents power of computer components with approximately stable power.

Secondly, the cross-degree has effects on the CPU frequency, CPU power, and further EC through CPU Frequency Scaling technique. As a result, if ω is continuously low, the CPU Frequency Scaling will reduce f value. The cross-degree $r(n)$ affects the trigger of CPU throttling. The higher the cross-degree is, the smaller possibility of ω being continuously low, which means the possibility of throttling becomes lower. Let $r(n) = r/n$, and cross-degree r_0 be the threshold of enabling throttling, therefore, when $r < r_0$, $f = scal(\omega)$; when $r \geq r_0$, $f = f_{max}$. Thus, there is a functional relation between CPU utilization ω and cross-degree $r(n)$, assuming $\omega = idel(r(n))$. $pcpu(f,\omega) = pcpu(scal(idel(r(n))), idel(r(n)))$.

Next, reuse-degree represents the possibility of cache-hit. It affects the memory utilization d. As a result, we assume that the functional relation between the reuse-degree $u(n)$ and the memory utilization d is $d = miss(u(n))$. Then $pmem(d) = pmem(miss\ (u(n)))$. Since $r(n)$ and $u(n)$ are both functions about n, let a $p(n)$ be the relationship between n and $power\ (V)$:

$$\begin{aligned} power(V) &= pcpu(scal(idel(r(n))), idel(r(n))) \\ &\quad + pmem(miss(u(n))) + pother \\ &= p(n) \end{aligned} \tag{7}$$

After that, in Equation (6), $r(n) = r/n$ and $u(n) = u/n$. The $scal()$ and $miss()$ are monotone increasing linear functions [23]; If $r \geq r_0$, $idel()$ is constant. If $r < r_0$, there is a constant k which makes $\omega = idel(r(n)) = O(k \times r(n))$. We adopt an upper-bound function of $idel()$ to substitute the unknown $idel()$. As a result, in Equation (6), $power()$, $scal()$, $miss()$ and $idel()$ are all known.

Then, it is known from Equation (6):

$$\begin{aligned} p(n) &= pcpu(scal(idel(r(n))), idel(r(n))) \\ &\quad + pmem(miss(u(n))) + pother \quad (r < r_0) \\ p(n) &= P_{max} + pmem(miss(u(n))) + pother \quad (r \geq r_0) \end{aligned} \tag{8}$$

Combined with Equation (6), by merging the coefficients, we derivate $p(n)$ as the following format:

$$\begin{aligned} &p(n) = b_1 r(n)^4 + b_2 r(n)^3 + b_3 r(n) + b_4 u(n) + b_5 \quad (r < r_0) \\ &p(n) = P_{max} + b_4 u(n) + b_5 \quad (r \geq r_0) \\ &r(n) = r/n \\ &u(n) = u/n \quad (b_1, b_2, b_3, b_4, b_5 \text{ are constants}, P_{max} > 0) \end{aligned} \tag{9}$$

To conclude, P_{max}, b_1-b_5 are both constants relating to the runtime environments. Therefore, we merge them into c_1-c_6. The r, u, n, $t(n)$ are all variables related to the source codes, which means that the values of these variables are predefined. Remaining the constants r and u, variable n and $t(n)$, equation $e(n) = p(n) \times t(n)$ is transformed as follows:

$$\begin{aligned} &AEC(r, u, n) \\ &= (c_1 r^4 n^{-4} + c_2 r^3 n^{-3} + c_3 (r + u) n_+^{-1} c_4) \times t(n) \quad (r < r_0) \\ &AEC(r, u, n) \\ &= (c_5 u n_+^{-1} c_6) \times t(n) \quad (r \geq r_0) \\ &\quad (c_1, c_2, c_3, c_4, c_5, c_6 \text{ are all code-independent constants}) \end{aligned} \tag{10}$$

In Equation (10), the main part of EC is the product of the constant power (c_4, c_6) and time, while the effect of the code structure on EC is reflected by the effect of exponent of n on the constant power (c_4, c_6). On one hand, if $r < r_0$, c_3 includes r (cross-degree) as well as u (reuse-degree),

TABLE 2. The experimental environment.

Item	Description
Computer	Tsinghua Tongfang Z900, CPU Intel i5-2300 2.80GHz, 8GB memory, 1TB hard disk. Power is from 60 watt to 110 watt.
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, sampling rate is 1 second.
Monitor Software	Using power meters, through USB, we transfer EC data that is collected by HOPI-9800 power meter on the experimental computer to the monitoring computer.
Programming Language	Java (jdk-1.7.0)
IDE	Eclipse 4.3
Measurement Units	EC unit: Joule (J); Time unit: Second (s); Cross-degree: Dimensionless; Reuse-degree: Dimensionless
Parameters	Parameters can be determined by regression analysis. In this section, the tendency of AEC and EC are consistent even all parameter values are simplified as 1.

which are major effects of code structure on EC, and c_4 is the power of components except CPU and memory. On the other hand, if $r \geq r_0$, c_5 includes only u (reuse-degree), in such situation, the CPU is fully loaded. The effect of the code structure on EC is mainly reflected by the variation of memory power, and c_6 is the power of components except the memory.

Finally, $t(n)$ is studied. The execution time of a task relates to the input scale, number of execution statements and execution time of each statement. **First** of all, the relation of input scale and execution time can be measured through algorithm complexity. However, this reflects the differences of execution times of statements eventually, the same as be reflected by the variation of n. **Next**, the number of executing statements is still not equivalent to that of statements in source codes even if the input scale is 1. Some statements are not executed and others are executed more than one time due to the branches and the loops. According to the proper approximation of AEC, let all the statements be executed only once. **At last**, different execution time of statements are ignored and simplified to 1 unit, then $t(n) = n$.

In conclusion, $e(n)$ is as:

$$e(n) = c_1 r^4 n^{-3} + c_2 r^3 n^{-2} + c_3(r+u)_+ c_4 n \quad (r < r_0)$$
$$e(n) = c_5 u_+ c_6 n \quad (r \geq r_0)$$
$$(c_1, c_2, c_3, c_4, c_5, c_6 \text{ are constants}) \quad (11)$$

VI. EXPERIMENTS

In this section, we design a group of experiments to verify the relationship between EC and cross-degree, reuse-degree, as well as AEC, respectively.

1) SETUP

We perform experiments in a real environments, and measure the EC of computer during the executions of tasks. The experimental environment, as shown in Table 2 and Table 3, includes the experimental computer, monitoring computer, data processing and analyzing tasks as cases.

TABLE 3. Description of the test use cases set (x is input scale, expressed with one decimal points).

Test Cases	Description	n	r/n	u/n
Calculation Statements	Calculate π^2, π remains 8 decimal places	x	0	1
Storage Statements	(Random)Access an object in large linked lists	x	0	0
Search	Linear Search	$3x$	0.4	0.3
InsertSort	Insertion Sort	$4x^2$	0.1	0.4
MergeSort	Merge Sort	$20x\log x$	0.4	0.2
LCS	Longest Common Sub-sequence	$16x+16x^2$	0.6	0.1
Floyd	Floyd–Warshall shortest path	$6x^3$	0.2	0.1

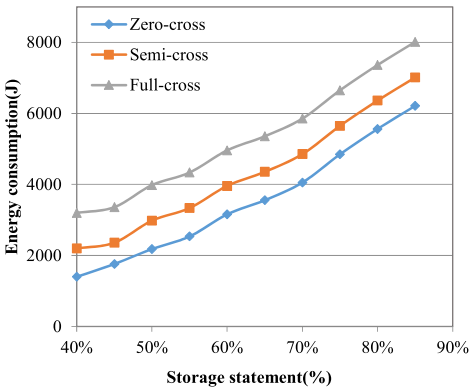

FIGURE 3. Statements EC of Different Cross-degrees.

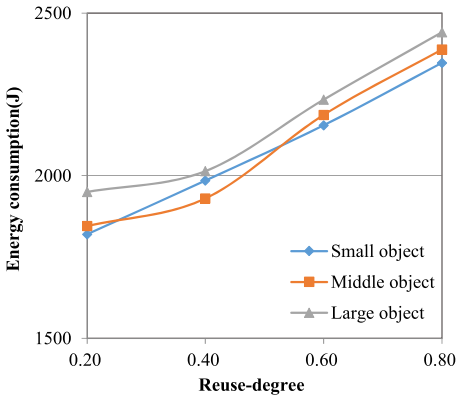

FIGURE 4. Statements EC of Different Reuse-degrees.

2) VERIFICATION OF THE RELATIONSHIP BETWEEN CROSS-DEGREE AND TASK EC

This experiment is designed to verify the effect of cross-degree on the EC. The test cases are not real algorithms, they contains the fixed number of statements. With the different proportions of storage statements (40%∼85%), we could simply change the cross-degrees as the zero-cross, semi-cross and full-cross. The Zero-cross means executing all storage statements firstly and executing all calculation statements secondly; the full-cross means that each storage (calculation) statement and each calculation (storage) statement are

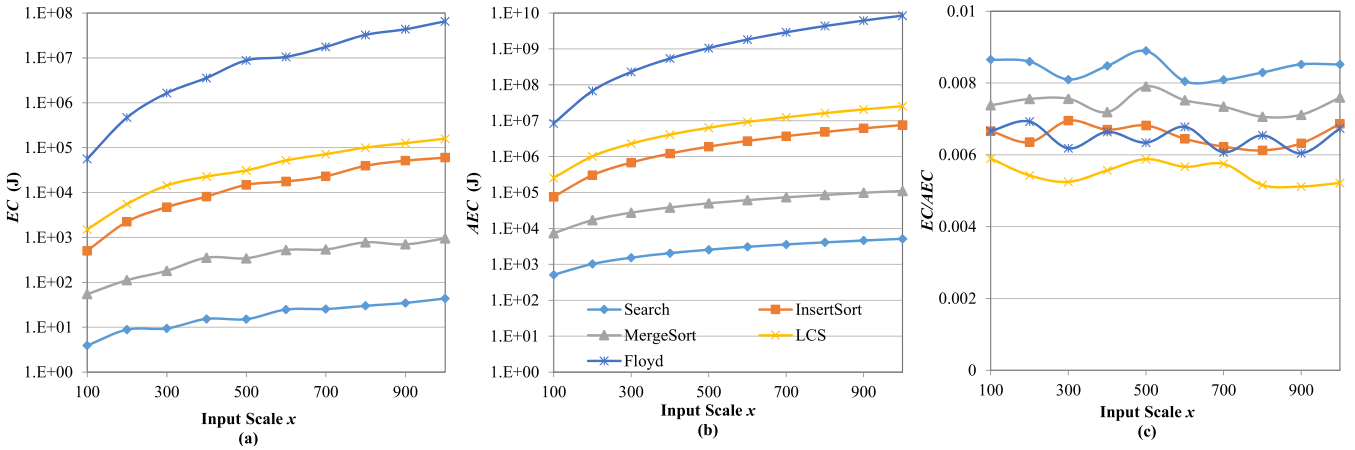

FIGURE 5. In Different Algorithms and Input Scale, Values of EC, AEC and EC/AEC.

executed alternately (statement by statement)[4]; and the semi-cross is some degree in between.[5]

In Fig. 3, the horizontal axis represents the proportions of storage statements; and the vertical axis represents the measured EC consumed by the experimental computer on which test cases are executed. On one hand, the EC increases obviously with the increment of the cross-degree in the same proportions of storage statements. It is close to a linear relation. On the other hand, in the conditions of same cross-degree, the EC positively correlates with the proportions of storage statements since the storage statements consume more energy than the calculation statements.

3) VERIFICATION OF THE RELATIONSHIP BETWEEN REUSE-DEGREE AND TASK EC

This experiment is designed to verify the effect of reuse-degree on the EC. We prepare a task containing only the storage statements which simply access an object in a large linked list. Based on it, we measure the EC in various reuse-distances adjusted by the different memory accessing ordersand various size of accessed objects. The objects are small-size (byte[100]), medium-size(byte[200]) and large-size (byte[400]). The enlargement of object sizes is equivalent to the reduction of the maximum numbers of objects stored in cache, the same as reduction of cache size (H), and vice versa.

In Fig. 4, the horizontal axis represents reuse-degree, and the vertical axis represents measured EC. For the same test case, the EC increases with the increment of reuse-degree. It is close to a linear relation, and the tendency is obvious. The memory power is small in proportion compared to the entire computer power. However, the CPU is idle and its power is lower since the test cases are all storage statements. As a result, the variations of memory powers have relatively obvious effects on EC. For different test cases, theoretically, the case that accesses the larger objects runs longer and consumes more energy. However, according to the Fig.4, EC of three cases are similar, because changing sizes of objects is equivalent to changing sizes of caches. If the reuse-degree remains the same, the case accessing large objects must contains less statements. So that the effects of accessing large object (small cache size) on EC are offset by effects of less statements on EC.

4) VERIFICATION OF THE CONSISTENCY OF AEC AND EC

To verify whether the AEC is consistent with EC, we compare the measured ECs and estimated AECs of five cases when the input scale increases. AEC represent the EC of tasks properly if the tendency of AEC and EC are consistent, and also the ratios of EC and AEC are relative stable. Search, Insert Sort, Merge Sort, LCS, Floyd are selected as the test cases, and their EC, AEC, and EC/AEC values are compared with different input scales (number of data items $x = 100, 200, 300, \ldots, 1000$). We expect that the values of EC/AEC of the same case under different scales should be approximately equal, or their variances should be small.

Comparing Fig. 5-(a) and 5-(b) (logarithmic coordinates), the tendencies of both EC and AEC values for the five tasks are consistent[6] no matter that their values are not equal. EC and AEC represent the differences of five tasks consistently. In Fig. 5-(c), the curves are almost stable. The mean values of EC/AEC for Search, Insert Sort, Merge Sort, LCS, Floyd are 0.0076, 0.0059, 0.0067, 0.0050, and 0.0059, respectively, meanwhile the standard deviation of them is 0.00026, 0.00027, 0.00024, 0.00028, and 0.00029 respectively. The experimental results shows that the proposed AEC can not only statically estimate, but also compare the EC of tasks.

[4]Full-cross: Assuming there are 2 calculation statements and 8 storage statements. Each calculation statement is followed by 4 storage statements. Vice versa.

[5]Semi-cross: Assuming there are 2 calculation statements and 8 storage statements. First 4 storage statements are followed by 2 calculation statements, and they are followed by another 4 storage statements.

[6]In fact, measured EC fluctuates, but the fluctuating is not obvious on logarithmic coordinates, while AEC curve accords with function curve.

VII. CONCLUSION AND FUTURE WORK

The paper proposes a static, code-structures highlighted EC estimation for cloud computing tasks. Firstly, the abstract EC model, named as AEC, is proposed as a simplified sources-level EC model. Then, the EC of a single statement is defined. Based on the model, the cross- degree and reuse-degree are proposed to quantify the effects of the code structures on EC. Next, the quantitative relationships between AEC and the cross-degree as well as the reuse-degree are derived, and an estimation function of AEC is given. Finally, the effectiveness of the AEC is verified through the designed experiments. In conclusion, AEC has the following advantages:

(1) Independence: It does not depend on the compiling environments and runtime environments.
(2) Rich features of code structure: The energy-related code structure features are well abstracted and modeled.
(3) Static estimation: Compiling or executing task is unnecessary. Instead, it can measure EC only by analyzing source codes.
(4) Reasonable precision: The precision requirement is appropriately relaxed by considering the consistency of the estimated values and actual values as well as the variation trend.
(5) Fairness: The runtime environment does not affect the fairness of AEC. In the same context, the differences of AEC among tasks should be almost consistent with the differences of actual EC, or they should satisfy a stable ratio.

On one hand, the AEC model does not fully consider the differences of statements, which will be extended in our future work. There are many similar or related researches to define a classification model of statements EC, thus, it is feasible to apply them to the source-level EC estimation. On the other hand, although the general estimation is easy to be applied to various cloud computing tasks, the features of a certain kind of tasks will also be considered in future. For example in the battery-powered computing environment such as wireless sensor networks [24], the proposed approach has a better application if AEC can be adapted to the tasks in the sensor networks.

REFERENCES

[1] R. Deng, R. Lu, C. Lai, T. H. Luan, and H. Liang, "Optimal workload allocation in fog-cloud computing toward balanced delay and power consumption," *IEEE Internet Things J.*, vol. 3, no. 6, pp. 1171–1181, Dec. 2016.
[2] A. Abbas *et al.*, "A survey on energy-efficient methodologies and architectures of network-on-chip," *Comput. Elect. Eng.*, vol. 40, no. 8, pp. 333–347, 2014.
[3] G. Jia, G. Han, J. Jiang, and L. Liu, "Dynamic adaptive replacement policy in shared last-level cache of DRAM/PCM hybrid memory for big data storage," *IEEE Trans. Ind. Informat.*, vol. 13, no. 4, pp. 1951–1960, Aug. 2017.
[4] G. Han, J. Jiang, M. Guizani, and J. J. P. C. Rodrigues, "Green routing protocols for wireless multimedia sensor networks," *IEEE Wireless Commun.*, vol. 23, no. 6, pp. 140–146, Dec. 2016.
[5] R. Deng, Z. Yang, J. Chen, N. R. Asr, and M.-Y. Chow, "Residential energy consumption scheduling: A coupled-constraint game approach," *IEEE Trans. Smart Grid*, vol. 5, no. 3, pp. 1340–1350, May 2014.
[6] G. Han, L. Liu, S. Chan, R. Yu, and Y. Yang, "HySense: A hybrid mobile crowdsensing framework for sensing opportunities compensation under dynamic coverage constraint," *IEEE Commun. Mag.*, vol. 55, no. 3, pp. 93–99, Mar. 2017.
[7] J. Pallister, S. J. Hollis, and J. Bennett, "Identifying compiler options to minimize energy consumption for embedded platforms," *Comput. J.*, vol. 58, no. 1, pp. 95–109, 2013.
[8] A. Hameed *et al.*, "A survey and taxonomy on energy efficient resource allocation techniques for cloud computing systems," *Computing*, vol. 98, no. 7, pp. 751–774, 2016.
[9] C. Pang, A. Hindle, B. Adams, and A. E. Hassan, "What do programmers know about software energy consumption?" *IEEE Softw.*, vol. 33, no. 3, pp. 83–89, May/Jun. 2016.
[10] C. Brandolese, "Source-level estimation of energy consumption and execution time of embedded software," in *Proc. EUROMICRO Conf. Digit. Syst. Design Archit., Methods Tools*, Sep. 2008, pp. 115–123.
[11] X. Zhou, B. Guo, Y. Shen, and L. Qi, "Design and implementation of an improved c source-code level program energy model," in *Proc. IEEE Int. Conf. Embedded Softw. Syst.*, May 2009, pp. 490–495.
[12] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier, "A preliminary study of the impact of software engineering on GreenIT," in *Proc. IEEE 1st Int. Workshop Green Sustain. Softw.*, Jun. 2012, pp. 21–27.
[13] J. Song, T.-T. Li, Z.-X. Yan, J. Na, and Z.-L. Zhu, "Energy-efficiency model and measuring approach for cloud computing," *J. Softw.*, vol. 23, no. 2, pp. 200–214, 2012.
[14] S. Schubert, D. Kostic, W. Zwaenepoel, and K. G. Shin, "Profiling software for energy consumption," in *Proc. IEEE Int. Conf. Green Comput. Commun. (GreenCom)*, vol. 268. Nov. 2013, pp. 515–522.
[15] 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.
[16] A. Noureddine, R. Rouvoy, and L. Seinturier, "Monitoring energy hotspots in software," *Automated Softw. Eng.*, vol. 22, no. 3, pp. 291–332, 2014.
[17] E. N. Elnozahy, M. Kistler, and R. Rajamony, "Energy-efficient server clusters," in *Power-Aware Computer Systems* (Lecture Notes in Computer Science), vol. 2325, B. Falsafi and T. N. Vijaykumar, Eds. Berlin, Germany: Springer, 2003, pp. 179–197.
[18] J. Song, T. Li, Z. Wang, and Z. Zhu, "Study on energy-consumption regularities of cloud computing systems by a novel evaluation model," *Computing*, vol. 95, no. 4, pp. 269–287, 2013.
[19] Q. Chen, P. Grosso, K. van der Veldt, C. de Laat, R. Hofman, and H. Bal, "Profiling energy consumption of VMs for green cloud computing," in *Proc. IEEE 9th Int. Conf. Dependable, Auto. Secure Comput.*, Dec. 2012, pp. 768–775.
[20] M. Gottscho, P. Gupta, and A. A. Kagalwalla, "Analyzing power variability of DDR3 dual inline memory modules," Univ. California, Los Angeles, CA, USA, Tech. Rep., 2011. [Online]. Available: http://nanocad.ee.ucla.edu/pub/Main/Publications/UG2_paper.pdf
[21] A. Hylick, R. Sohan, A. Rice, and B. Jones, "An analysis of hard drive energy consumption," in *Proc. IEEE Int. Modeling, Anal. Simulation Comput. Telecommun. Syst. (MASCOTS)*, Sep. 2008, pp. 1–10.
[22] 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.
[23] *CPU Frequency and Voltage Scaling Code in the Linux(TM) Kernel*. Accessed: Feb. 1, 2017. [Online]. Available: https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt
[24] G. Han, X. Yang, L. Liu, M. Guizani, and W. Zhang, "A disaster management-oriented path planning for mobile anchor node-based localization in wireless sensor networks," *IEEE Trans. Emerg. Topics Comput.*, to be published.

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 interest includes computational fluids dynamics and green industry.

SHAOKUI ZHANG is currently pursuing the B.S. degree in software engineering with the Software College, Northeastern University, Shenyang, China. His current research interest is green computing.

TAO XIAO is currently pursuing the M.S. degree with the School of Computer Science and Engineering, Northeastern University, Shenyang, China. His research interest is 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 interest includes green computing, big data management, and machine learning.