IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 2, 2018, accepted December 11, 2018, date of publication December 24, 2018,
date of current version January 11, 2019.

Digital Object Identifier 10.1109/ACCESS.2018.2889400

Do Google App Engine's Runtimes Perform Homogeneously? An Empirical Investigation for Bonus Computing

ZHENG LI[1], (Member, IEEE), AND XUE GUO[2]
[1]Department of Computer Science, University of Concepción, Concepción 4070409, Chile
[2]Software Institute, Nanjing University, Nanjing 210008, China

Corresponding author: Zheng Li (zhengli@inf.udec.cl)

This work was supported in part by CONICYT under Grant FONDECYT Iniciación 11180905, in part by the University of Concepción under Grant VRID INICIACION 218.093.017-1.0 IN, and in part by the National Natural Science Foundation of China (NSFC) under Grant 61572251.

ABSTRACT Bonus computing is a new metacomputing form that takes advantage of free computing power from the public Cloud market. To maximize the value of free Cloud resources and facilitate dividing Bonus computing tasks, it would particularly be crucial to understand the performance of candidate Cloud services before using them in production. By offering free quotas in its standard environment, Google App Engine (GAE) has become a popular public Platform-as-a-Service (PaaS) for Bonus computing. Since GAE natively supports various programming languages with flexible configurations (e.g., region selection), it will be possible and valuable to squeeze GAE's free computing power if there is an optimal choice of its different runtimes. Following the performance evaluation methodology DoKnowMe, we implemented several versions of the Fibonacci(-like) calculation as benchmarks to fundamentally investigate GAE's standard environment. Our investigation results reveal that GAE does not support its runtime environments homogeneously in terms of their computation speed and memory efficiency. The heterogeneity could be related not only to the characteristics of different programming languages but also to the diverse GAE infrastructures. For example, Go runtime seems to be a well-trade-off to satisfy Bonus computing among all the options, while the GAE service located in *southamerica-east1* and *us-central1* performs dramatically worse than that in the other regions.

INDEX TERMS Bonus computing, Google App Engine, performance evaluation, Platform as a Service, programming languages.

I. INTRODUCTION

Evolving from Volunteer computing [1] and Parasitic computing [2], Bonus computing emerged as a new metacomputing form to take advantage of free computing power from the public Cloud market [3]. On one hand, the free and high-available Cloud resources can avoid potential legal and ethical issues in Parasitic computing, and can supplement the possible shortage of volunteers especially for small-scale projects. On the other hand, the free quotas of public Cloud services generally have various limitations and constraints, which makes Bonus computing suitable mainly to carry out jobs composed of fine-grained, fault-tolerant, and compute-intensive tasks. Correspondingly, understanding the characteristics of candidate Cloud services is particularly crucial and necessary for Bonus computing, so as both to maximize the value of usually limited free resources and to develop problem solutions with proper task divisions.

Google App Engine (GAE) is currently a popular public Platform-as-a-Service (PaaS) for Bonus computing, as it offers free quotas in its standard environment with native support for various programming languages [4]. However, although the diverse language runtimes of GAE provide flexible options to developers who have different backgrounds, they could in turn incur overchoice when those options are equivalently employable [5]. Actually many practitioners have been confronted with such choice overload in the numerous types of programming languages [6], [7]. Within the context of GAE, there are also frequent discussions about decision making in the aforementioned language runtimes,

for various reasons ranging from performance comparison to exclusive characteristics identification [8], [9].

Meanwhile, when selecting GAE's runtime environments, it would be inaccurate and even inappropriate to directly reuse the existing knowledge of programming language comparisons, because it is unclear about how well GAE supports its language runtimes. After all, Cloud services' back-ends like configurations of physical infrastructure are uncontrollable and invisible for their consumers [10]. Additionally, PaaS platforms like GAE can further impose constraints on their applications due to the vendors' concerns (e.g., protecting the system integrity) [11]. Therefore, it is imperative and beneficial to understand and assess GAE (as a specific Cloud service) before using it in production [10], [12].

To facilitate employing GAE in Bonus computing, we empirically investigated its performance homogeneity with respect to different language runtimes (i.e. Java 7, Java 8, Python, Go, and PHP in the free offers) as well as different regions, by following the performance evaluation methodology DoKnowMe [13]. This paper reports our empirical investigation. In brief, we implemented several versions of the Fibonacci(-like) calculation as benchmarks to make the GAE environmental comparison fundamentally ''apple-to-apple''. The investigation results show that GAE's runtime environments perform heterogeneously not only in different programming languages but also across different geographical regions.

To our best knowledge, this is the first holistic investigation into the free quotas of GAE standard environment. Such a fundamental study makes mainly twofold contributions:

1) The revealed strengths and weaknesses of different runtimes can help practitioners better understand GAE and make trade-off decisions in its language runtime selection. For example, to squeeze GAE's free computing power, we have chosen the Go runtime for our future Bonus computing projects according to its well balance between computation speed and memory efficiency.
2) The identified peculiarities of GAE can help the provider, Google in specific, improve its product and make different language runtimes more comparable and homogeneous. For example, we believe that Google needs to pay more attention to its regions *southamerica-east1* and *us-central1* where GAE performs significantly poor.

The remainder of this paper is organized as follows. Section II briefly reviews the emergence of Bonus computing and summarizes the existing practices related to GAE's performance evaluations. Section III specifies the experimental design that drives our empirical investigation. The experimental results and analyses about the performance homogeneity in GAE's different language runtimes and regions are reported in Section IV. Section V discusses some possible threats to the validity of our study. Conclusions and future work are drawn and specified in Section VI.

II. RELATED WORK

A. EMERGENCE OF BONUS COMPUTING

Bonus computing emerged from two existing metacomputing forms, namely Volunteer computing and Parasitic computing. By exploiting idle processing resources from general public and even casual owners, Volunteer computing has been applied to various large projects and scientific collaborations ranging from discovering new stars [14] to searching Mersenne prime [15]. Nevertheless, Volunteer computing can suffer from the shortage of computing resources and even from uncertain interruptions. Firstly, it has been identified that ''the number of people participating in volunteer computing compared to the number of users on the Internet is insignificantly small'' [16]. Secondly, one volunteered device can be dedicated only to a single project at a time. Thirdly, volunteered devices can be out of usage at any time due to unexpected crashes, network disconnections, or intentional leaving.

Parasitic computing, on the other hand, utilizes the standard protocols (e.g., TCP checksum function) to exploit the trivial while pervasive computing power from the Internet communication infrastructure [2]. Unfortunately, since the Internet protocols have a low computation-to-communication ratio due to their nature of message transferring, the implementation of parasitic computing is computationally inefficient in practice. More importantly, there are ethical and legal concerns about parasitic computing in terms of using remote hosts without any authorization.

Given the booming of Cloud computing that is supposed to provide always-on or at least extremely-high-availability computational utility [10], there is an increasing amount of free-tier opportunities offered in the public Cloud market. As a compromise between, and a supplement to, the aforementioned metacomputing forms, we proposed Bonus computing to efficiently take advantage of free Cloud resources and to deal with suitable problems like Monte-Carlo simulations [3]. In specific, Bonus computing is equipped with a loose-coupling functional architecture including three roles, i.e. recipient, broker, and contributor. The recipient follows the Divide-and-Conquer (D&C) strategy and uses asynchronous mechanisms to break a whole job into tasks and receive task results. The contributor employs lightweight service technologies (e.g., Function-as-a-Service, microservice, and RESTful Web service) to donate his/her free quota of Cloud resources. Note that, since the service calls can be made independently, it is possible to deploy different project services on the same contributor's Cloud resource. The broker maintains a resource & service registry to facilitate scheduling and coordinating recipient tasks with contributor services.

B. PERFORMANCE EVALUATION OF GAE

Among the diverse types of Cloud services, PaaS is particularly attractive for its capacity of taking the full responsibility

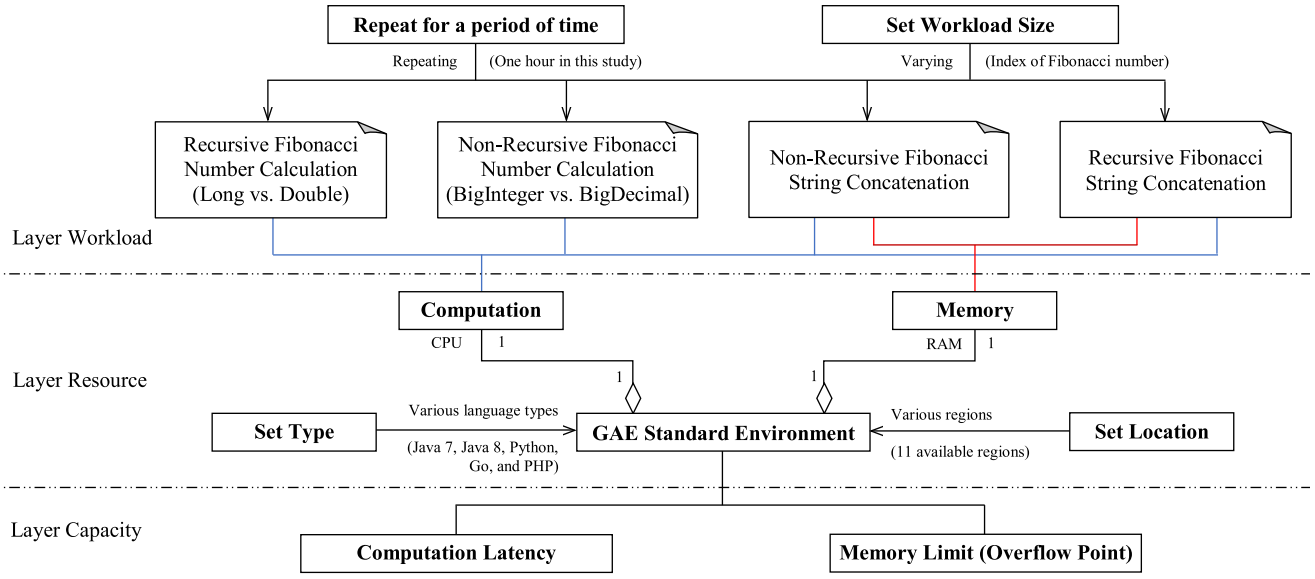

FIGURE 1. Experimental blueprint for evaluating GAE's computation and memory features.

of application runtime on behalf of developers. A pioneer work [17] extensively discussed the advantages of, and the concerns about, applying PaaS technology to developing online software systems. Since GAE is one of the most representative PaaS platforms [10], numerous experimental and experience studies have been conducted to investigate and demonstrate its wide use cases.

For example, in academia, GAE has proven suitable for designing and developing distributed and scalable systems for high-performance scientific computing [11], [18]–[21]; in education, GAE has acted as a convenient courseware particularly for teaching Web programming [22] and for mobile learning [23]; and in practice, GAE has succeeded in some outstanding projects like Snapchat and Khan Academy by addressing the challenges of quick scaling and concurrent access [24]. Furthermore, there are even plenty of books that particularly explain GAE-based application developments [25].

However, it is noteworthy that the existing studies and books tend to focus on individual-language runtimes only (e.g., either the Java-specific [26] or the Python-specific runtime [27]), while leaving the other GAE environments to be covered by external online documents and printed materials [24]. In contrast, our work delivers fundamental investigation into and "apple-to-apple" comparison between different GAE environments, in order to help satisfy the needs of clear understanding and assessment of a particular Cloud service against real-world requirements [10], [12].

In addition, the purposes of the related work and ours are different. The existing studies are mostly to demonstrate or verify the feasibility of leveraging GAE's free quotas for utilization in production, for example implementing Web services in the domain of Geographic Information System [28]. In our study, the homogeneity investigation is to identify possible performance gaps and eventually help squeeze the free computing power of GAE.

III. EXPERIMENTAL DESIGN

As mentioned previously, our empirical study is conducted by following the performance evaluation methodology DoKnowMe [13] that includes ten steps from *Requirement Recognition* to *Conclusion and Documentation*. To avoid duplication, we do not elaborate the details of all the steps of DoKnowMe. Instead, we only briefly explain the whole experimental logic through the corresponding experimental blueprint that includes three layers (namely Resource, Workload, and Capacity), as shown in Figure 1.

1) CLOUD RESOURCE TO BE EVALUATED

Driven by the requirement of investigating the GAE environments for compute-intensive tasks, we mainly focused on two physical features of GAE's standard environment, i.e. Computation and Memory, with respect to the five available language runtimes of GAE as different resource types (cf. Table 1). Actually, GAE's flexible environment contains more accessible language runtimes like Node.js, .NET and Ruby. Due to the fact that this paper only contributes to bonus computing, i.e. focusing only on the free quota of GAE, flexible environment is not included in our investigation. Moreover, in addition to the default GAE region

TABLE 1. Available language runtimes in the standard environment of GAE.

GAE Runtime	Java	Java	Python	Go	PHP
Version	7	8	2.7	1.8	5.5

asia-northeast1(Tokyo), we are also concerned with the other resource locations. Although non-billing accounts are not able to choose GAE instance classes [4], it is possible to switch among different runtime regions. Therefore, we replicated the experiments across the currently 11 available regions (cf. Section IV-E).

2) BENCHMARK WORKLOAD

To represent compute-intensive tasks, we employed Fibonacci number generation as the basic benchmark, and varying benchmark workload sizes by using different indexes of Fibonacci numbers. Recall that the calculation of a Fibonacci number has both recursive and non-recursive algorithms. We tested recursive Fibonacci calculation with build-in data types `long` vs. `double`, and tested non-recursive Fibonacci calculation with `BigInteger` vs. `BigDecimal` solutions of different languages. In addition, we also implemented Fibonacci-like string concatenation algorithms to observe the performance of `String` vs. `StringBuilder` processing on GAE.

Note that the data type names shown in this paper are in a generic sense, and they are replaced with the language-specific data types in the experimental implementation. For example, `float` in Python is used to indicate `double` precision floating point numbers; the `java.math.BigDecimal` in Java is equivalent to `decimal.Decimal` in Python; and big integer number is natively supported by Python, while other languages need to import external `BigInteger` class or package; and the `StringBuilder` in Go is `bytes.Buffer`. The specific codes of different language runtimes have been shared on GitHub.[1]

Considering the free quota limits of GAE [29], we repeated each type of the aforementioned trials for roughly one hour, with a 20-second sleep between two consecutive trials for reducing the possible impact of cache. In particular, our pilot experiments revealed that there seemed to be a ten-minute time limit of continuous execution session per request on GAE. Consequently, we had to manually issue six requests successively to satisfy the one-hour experiment of every benchmarking scenario.

3) CLOUD CAPACITY TO BE MEASURED

To reduce environmental noises and make the performance measurement fair and rigorous, we only recorded the Fibonacci calculation latency and excluded the time expenditure of result saving/transmission, as exemplified by the following Java code. In other words, the measured computation latency is independent of the distance between our location and the regions.

Thus, we naturally use computation latency (in *ms*) to reflect the capacity of GAE. Furthermore, since the length of a string is proportional to the size of memory footprinted by the string, a side benefit of the Fibonacci-like string concatenation test is being able to reveal the memory usage efficiency of different GAE language runtimes. More details of the revealed memory limits are specified in Section IV-C.

[1] https://github.com/NJUGX/BenchmarkingGAE

```
public long FibonacciLatency(int n) {
  long start, end, latency;
  start = System.nanoTime();
  long fibo = Fibonacci(n);
  end = System.nanoTime();
  latency = (end - start)/1000000;
  return latency;
}
```

IV. INVESGITATION RESULTS AND ANALYSES

We started from the Java runtime experiment, and then translated the Java codes into the other languages so as to make different runtime evaluations consistent and comparable. However, it makes no difference if starting from any other language runtime and then doing "translated" experiments.

A. RECURSIVE FIBONACCI NUMBER CALCULATION

We use the data types `long` and `double` respectively in recursive Fibonacci(-like) number calculation to investigate the integer and floating-point performance of different GAE runtime environments. The Java code for floating-point test is shown as an example below.

```
public double Fibonacci(int n) {
  double result = 0;
  if (n == 1 || n == 2)
    result = 1.1;
  else
    result = Fibonacci(n - 1) +
        Fibonacci(n - 2);
  return result;
}
```

Given the short execution timeout limit in PHP by default, our pilot experiment shows that the PHP runtime of GAE cannot carry out Fibonacci workload as large as the other runtime environments do. Using PHP as the bottleneck baseline, we decided to calculate the 30th Fibonacci number for the five-environment comparison, as illustrated in Figure 2. Note that Figure 2 does not show the complete one-hour trials, because the PHP runtime always stops response after three or four execution sessions in this experiment.

It is not surprising that PHP performs the worst, because it is not a language for dealing with compute-intensive jobs by design. Python is also tremendously worse than Java and Go in this case, which confirms that speed is not a favorable concern for Python programs in the community. In particular, a similar trend shared by the Python and PHP runtimes is that their performance would deteriorate after running a short period of time. In contrast, the Java and Go runtimes perform too fast to display clear regular patterns or trends at the current workload size and visualization scale. Thus, we rerun this experiment for Java and Go with the 40th Fibonacci

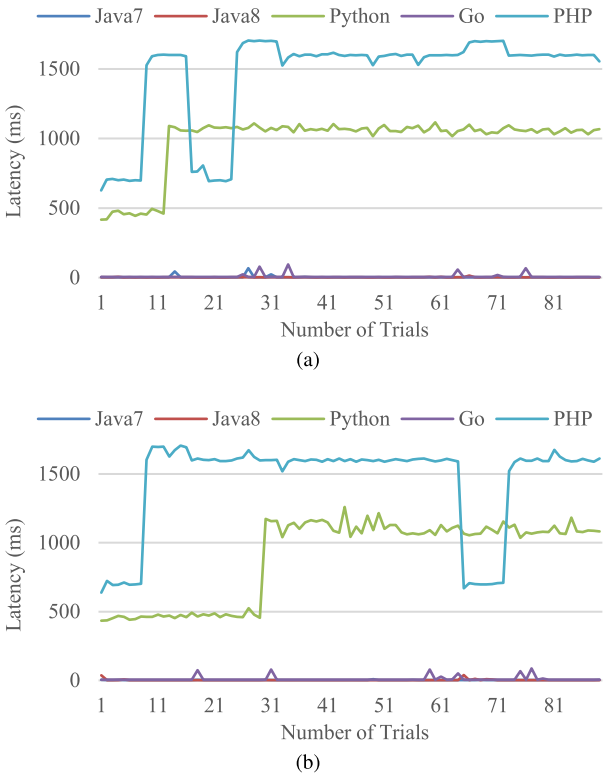

FIGURE 2. Latency of recursive Fibonacci number calculation in the five runtimes of the GAE standard environment. (a) The 30th `long` Fibonacci number calculation. (b) The 30th `double` Fibonacci number calculation.

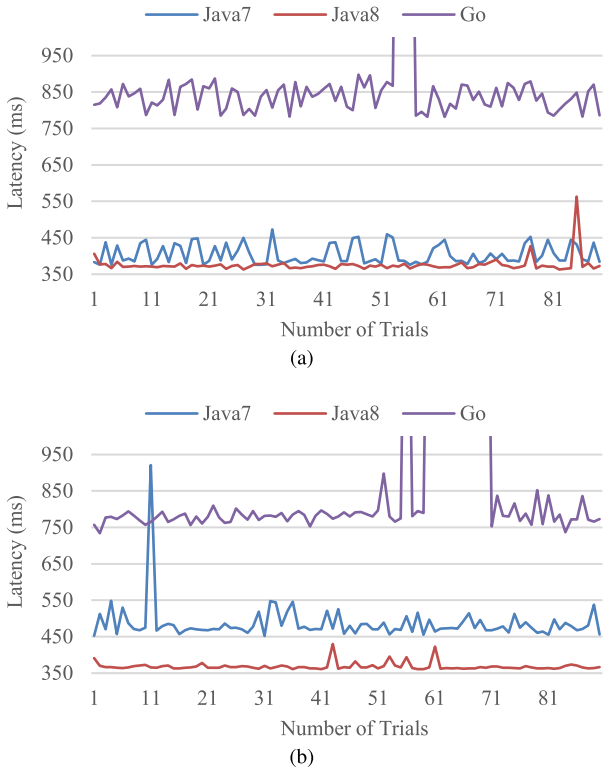

FIGURE 3. Latency of recursive Fibonacci number calculation in the Java and Go runtimes of the GAE standard environment. (a) The 40th `long` Fibonacci number calculation. (b) The 40th `double` Fibonacci number calculation.

calculation both to enlarge and to zoom in on the potential difference in their performance, as shown in Figure 3.

It can be seen that the two Java runtimes perform nearly twice as fast as Go here. In addition, the Go runtime could have relatively frequent performance drops with around 2.5 times longer latency (invisible in the figures) in both the `long` and the `double` situations. However, compared to Java 7, the Go and Java 8 runtimes seem to be optimized for floating-point computing. Go's floating-point performance is even slightly better than its integer performance on GAE.

B. NON-RECURSIVE FIBONACCI NUMBER CALCULATION

Recall that Fibonacci number calculation has both recursive and non-recursive solutions. We are also interested in the performance of GAE standard environment for non-recursive workloads. Since the non-recursive solution is merely a `for` loop (see the Java code below), calculating a large index of Fibonacci number will be needed in order to reduce the noises of performance measurement in this case.

Note that, considering the limited data ranges of the native data types, we resort to `BigInteger` and `BigDecimal` whose values have theoretically no upper/lower bounds, and compute the 10000th Fibonacci number in this experiment.

Given the experimental results visualized in Figure 4, it is clear that GAE runtimes except PHP deliver fairly similar performance (mostly less than 10*ms*) and jitters

```
import java.math.BigDecimal;
public BigDecimal NonRecurFibonacci(int n) {
   BigDecimal first = new BigDecimal("1.1");
   BigDecimal second = new BigDecimal("1.1");
   BigDecimal third = new BigDecimal("1.1");

   for (int i = 2; i < n; i++) {
      third = first + second;
      first = second;
      second = third;
   }
   return third;
}
```

(sometimes as high as more than 100*ms*) in the calculation with `BigInteger` (cf. Figure 4a). When it comes to the calculation with `BigDecimal` (cf. Figure 4b), the most comparable runtimes are Java and Go. In particular, two Java runtimes show little performance difference between the calculations with `BigInteger` and with `BigDecimal`, while the peak performance of Go runtime drops about 10*ms* in the `BigDecimal` situation compared to `BigInteger` (from 4*ms* to 14*ms*).

Unlike Java and Go, the Python runtime and especially the PHP runtime exhibit unique characteristics and enormous performance difference between the `BigInteger` test and the `BigDecimal` test, as shown in Figure 5. For example, although Python natively supports `BigInteger` and

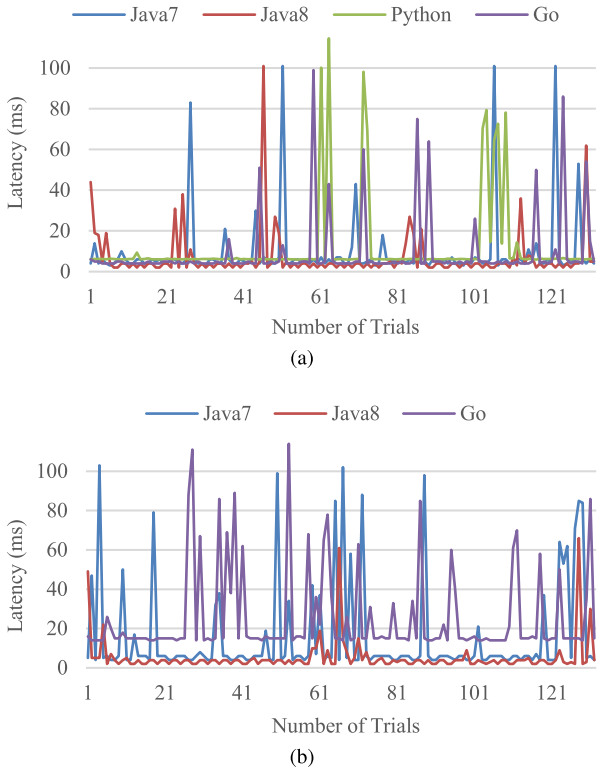

FIGURE 4. Latency of non-recursive Fibonacci number calculation in the Java, Python and Go runtimes of GAE the standard environment. (a) The 10000th `BigInteger` Fibonacci number calculation. (b) The 10000th `BigDecimal` Fibonacci number calculation.

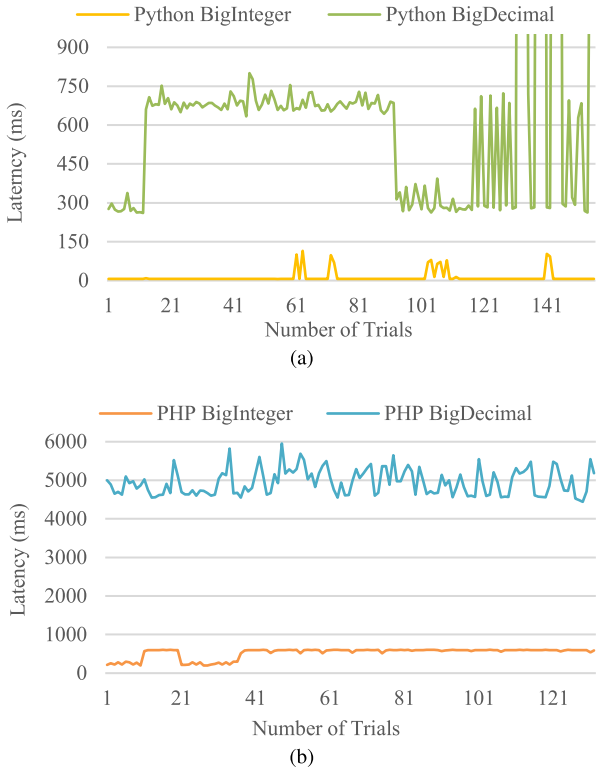

FIGURE 5. Latency of non-recursive Fibonacci number calculation in the Python and Go runtimes of the GAE standard environment. (a) The 10000th Fibonacci number calculation in Python. The 10000th Fibonacci number calculation in PHP.

its performance is similar to Java and Go (cf. Figure 4a), the latency of Python's 10000th Fibonacci number calculation with `BigDecimal` varies dramatically and mostly between 260*ms* and 760*ms*. The PHP runtime is mostly as slow as around 600*ms* when calculating the 10000th Fibonacci `BigInteger` number, while the same level of `BigDecimal` calculation can take approximately 5 seconds on average. This phenomenon confirms that, as a Web-oriented programming language, the PHP runtime is not an ideal candidate for large-scale compute-intensive jobs on GAE.

C. FIBONACCI STRING CONCATENATION

Considering that the implementations of `BigInteger` and `BigDecimal` can rely on string representations and operations. We further tested GAE runtimes against Fibonacci string concatenation. Here we only use the recursive version of Java code to exemplify the experimental benchmark, as shown below.

```
public String FibonacciString(int n) {
  String result = "";
  if (n == 1 || n == 2)
    result = "a";
  else
    result = FibonacciString(n - 1) +
        FibonacciString(n - 2);
  return result;
}
```

When determining the workload size, we found that different language runtimes met the out-of-memory error at different indexes of the Fibonacci string concatenation, as listed in Table 2. Considering that the *i*th string concatenation occupies the *i*th Fibonacci number of bytes, Go and PHP applications seem to be able to enjoy much more memory than Java and Python applications on GAE. In other words, GAE's Go and PHP runtimes have significantly higher memory efficiency. In particular, we believe it is the default JVM memory limit (about 124MB in our test of the Java 8 runtime) that causes the Java runtimes the worst in this case. Note that JVM's default maximize size of the memory allocation pool (`MaxHeapSize`) is one fourth of the "physical" memory or 1GB (whichever is smaller).

To avoid possible out-of-memory errors, we decided to choose 35 as the index of Fibonacci string concatenation. Also to facilitate observing the performance of five runtimes in the non-recursive experiment, we triple the 35th concatenation to act as the benchmark workload of each trial. The experimental results are visualized in Figure 6.

In the recursive experiment, although performing better than PHP on average (cf. Table 2), the Python runtime exhibits surprisingly high variation roughly between 16 and 40 seconds. By zooming in on the Java and Go runtimes only,

TABLE 2. Comparison between different GAE runtimes w.r.t. recursive string concatenation.

Runtime	Average Latency (35th)	Memory Overflow Index
Java 7	$1479.0ms$	$37th$
Java 8	$990.4ms$	$37th$
Python	$8833.7ms$	$40th$
Go	$805.5ms$	$44th$
PHP	$10058.1ms$	$45th$

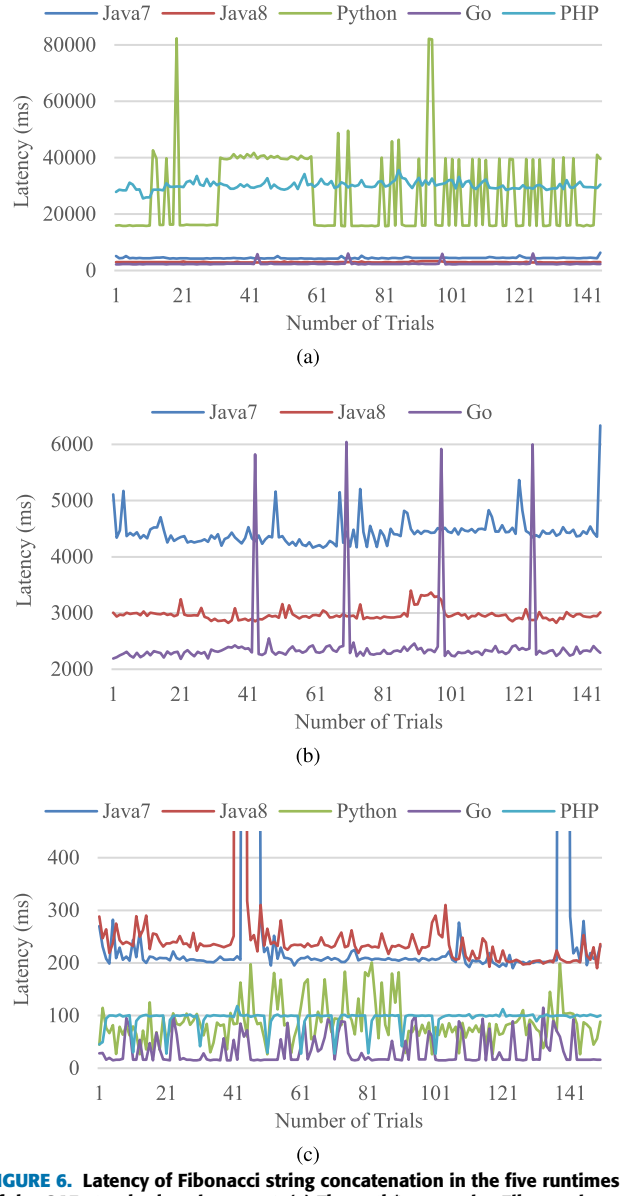

FIGURE 6. Latency of Fibonacci string concatenation in the five runtimes of the GAE standard environment. (a) The 35th*3 recursive Fibonacci string concatenation. (b) The 35th*3 recursive Fibonacci string concatenation. (c) The 35th*3 non-recursive Fibonacci string concatenation.

Java 8 seems to be a significant improvement from Java 7 at least for this type of jobs, and Go clearly shows its advantage while still having regular performance drops.

As for the experiment of non-recursive Fibonacci string concatenation, surprisingly, the Java runtimes perform the worst with even large variations. Go still wins in such work, followed by the Python and PHP runtimes that are both close to Go in terms of the latency or speed.

D. INVESTIGATION INTO REGULAR PERFORMANCE DETERIORATION OF THE GO AND JAVA 8 RUNTIMES

By focusing on the Go runtime, we see regular performance deterioration in many experiments especially the recursive ones (e.g., Figure 6b). As mentioned in the experimental design (cf. Section III), each continuous execution session of our experiments on GAE has about a ten-minute limit, and thus we had to manually issue a new request (by refreshing the webpage) roughly every dozen or fifteen minutes to conduct relatively long-term experiments. Interestingly, the performance deterioration of Go runtime seems to happen at every request. Such a phenomenon reminds us of the observations on two Java runtimes' performance during our pilot experiments. When dealing with light workloads like calculating the 22th or so Fibonacci number, the Java runtimes have the similar regular pattern. Thus, we reran the two typical experiments where Java 8 and Go first showed up with regular performance drops, namely the 22th `long` Fibonacci number calculation in Java 8 (cf. Figure 7a) and the 35th*3 `String` Fibonacci string concatenation in Go (cf. Figure 7b).

Considering that an application running on GAE can dynamically launch multiple instances to satisfy requests [21], we hypothesize that those performance drops correspond to the lifecycles of different application instances on GAE. Following the idea of instance tracking in [21], we also use a global static variable (i.e. `instanceID` in the Go code below) to track GAE application instances. It is noteworthy that global static variables of an application instance can be initialized only once and then their values will never change during the instance's lifecycle. Thus, by trying to vary the value of the static variable as the benchmarking trials proceed, we will be able to identify new application instances if the variable's value changes. To emphasize this generic feature, we use the Go code to demonstrate our solution:

```
func handler(w http.ResponseWriter, r
    *http.Request) {
  ctx := appengine.NewContext(r)
  fiboIndex := 35
  for instanceID := 1; instanceID <= 150;
      instanceID++ {
    totalTime := FiboTime(fiboIndex)
    dataStore(ctx, instanceID, totalTime)
    time.Sleep(20 * time.Second)
  }
   fmt.Fprint(w, "Finish")
}
```

The instance tracking results from our benchmark applications are illustrated in Figure 8. It is interesting to see

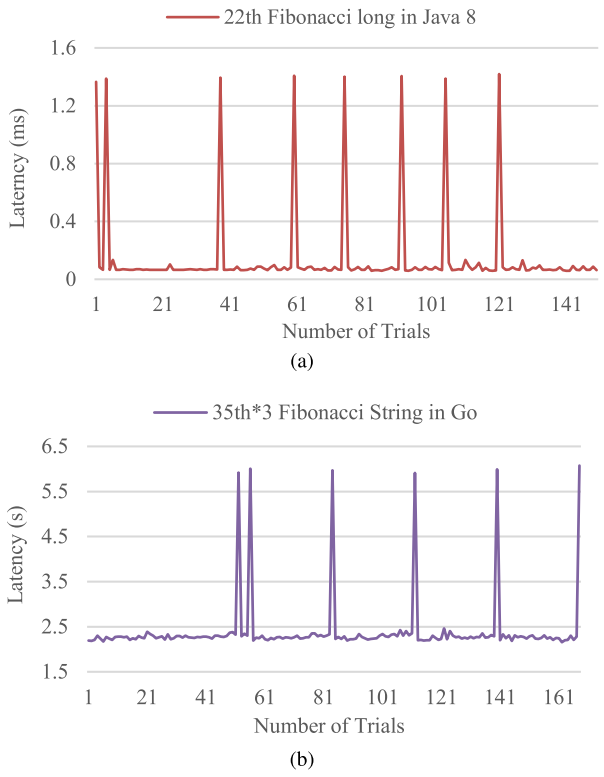

FIGURE 7. Typical performance drops in the Java8 and Go runtimes. (a) Performance drops of the Java 8 runtime. (b) Performance drops of the Go runtime.

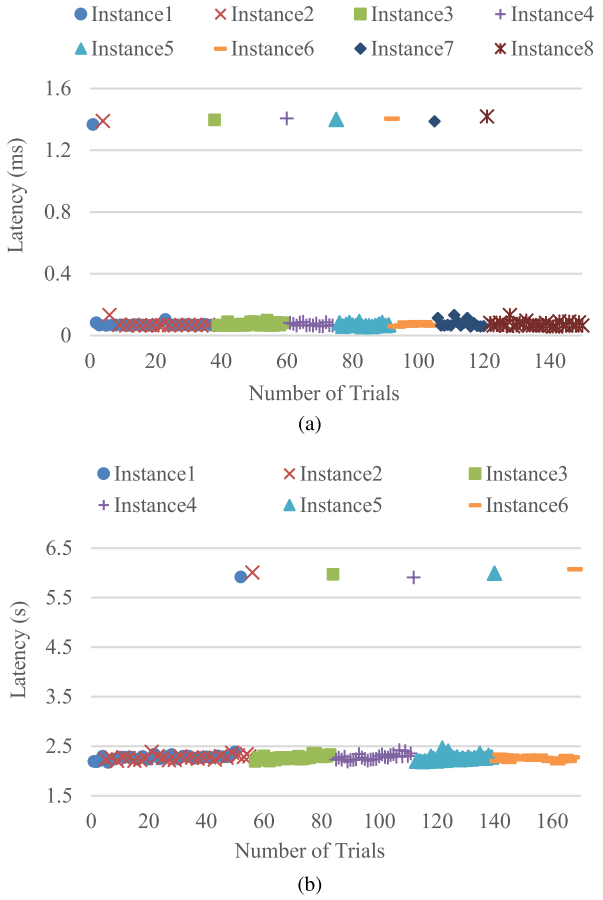

FIGURE 8. Tracking instances of the two benchmark applications corresponding to Figure 7. (a) Instance tracking in the Java 8 runtime. (b) Instance tracking in the Go runtime.

that the identified eight and six instances exactly match the aforementioned performance jitters of the Java 8 and Go runtimes respectively (cf. Figure 7). However, the two runtimes' performance deteriorations have different occurrence patterns. On the Java 8 runtime, it seems that performance drops at the beginning of each application instance's lifecycle (cf. Figure 8a); whereas the performance deterioration seems to happen at the end of every instance lifecycle on the Go runtime (cf. Figure 8b). In addition, we reckon that GAE tries to launch multiple (two in this case) instances simultaneously to deal with potential request flood when running a new application.

Recall Java's Just-In-Time (JIT) compilation mechanism that compiles bytecodes into machine code to improve applications' performance at the first run. Since JIT compilation also consumes processing resources as well as processing time, the performance drops of Java 8 runtime can be explained to be the extra resource/time consumption (about 1.3 milliseconds according to Figure 7a in this experiment) by the JIT compiler. In fact, it has been confirmed that JVM needs a "warm up" period before actual working [21] to initiate a high-efficient and consistent computation. However, the performance drops of Java runtime become negligible (and invisible) for heavy workloads, as demonstrated in Figure 3, which indicates that the overhead of JIT compilation could be generally trivial at least in this type of workloads.

As for the Go runtime, given the occurrence moments of the performance deterioration, we believe that the root cause lies in Go's garbage collection (GC) mechanism, and the overhead of GC could take as high as 3.5 seconds in our experiment. Since our benchmark application footprints both stack space (recursive Fibonacci) and heap space (string concatenation) in memory, it would not be surprising for heavy workloads to trigger GC activities at the end of instance lifecycles. On the other hand, we do not see any performance deterioration for lightweight jobs in the Go runtime.

E. COMPARISON BETWEEN THE 11 CURRENTLY AVAILABLE REGIONS

Developing datacenters in different geographical regions has become a common practice to improve the reliability and scalability of Cloud services. Considering the possible inconsistency in the datacenter development from different locations, we are also concerned with the performance homogeneity across all the available regions of GAE. Given the previous experimental results, Java 8 is clearly the most stable one among all the runtimes especially for recursive programs. Therefore, we choose the Java 8 runtime to represent GAE in the region-related

performance study. And also to demonstrate our experiment with different sizes of workloads, we employ the recursive calculation of the 45th `long` Fibonacci number to benchmark GAE standard environment within the totally 11 available regions at the time of writing, namely northamerica-northeast1 (Montréal), us-central (Iowa), us-east1 (South Carolina), us-east4 (Northern Virginia), southamerica-east1 (São Paulo), europe-west (Belgium), europe-west2 (London), europe-west3 (Frankfurt), asia-northeast1 (Tokyo), asia-south1 (Mumbai), australia-southeast1 (Sydney) [30].

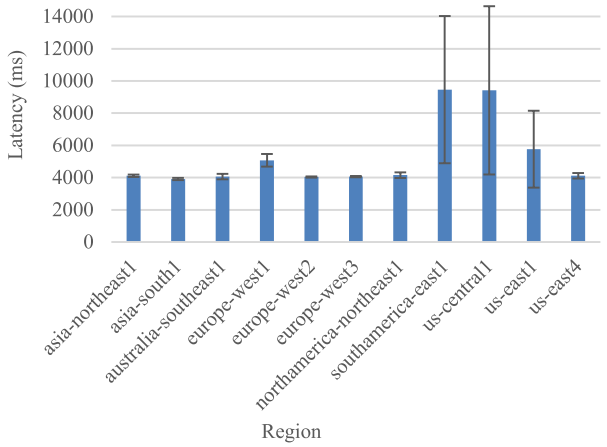

FIGURE 9. Performance comparison of the Java 8 runtime for recursively calculating the 45th `long` Fibonacci number in the 11 available regions (the error bars indicate the standard deviations).

As can be seen from the benchmarking results in Figure 9, we cannot expect even close runtime performance across all the regions of GAE. The regions offering relatively the best GAE environment seem to be *europe-west2*, *europe-west3* and the two Asian ones. Although the Australian and North American ones and *us-east4* also host best-performance runtimes on average, they have slightly bigger variations. Compared to these seven regions in this experiment, GAE runtimes in *southamerica-east1* and *us-central1* show more than two times worse performance in terms of execution latency as well as significant performance fluctuations, followed by *us-east1* and *europe-west1*.

V. THREATS TO VALIDITY

We aim to conduct a fundamental performance investigation into GAE runtime environments for squeezing GAE's free quota in Bonus computing. The heterogeneous experimental results show that it is possible to enjoy more computing power by properly selecting runtime and region of GAE. However, our study still remains some threats to fully understanding and assessing GAE.

Firstly, we ignored time-related performance homogeneity of GAE. It is known that Cloud services may exhibit different quality due to the uncertain resource competitions at different times. Thus, a longer-term monitoring would be valuable to further reveal whether or not GAE performs homogeneously along the time goes by. Unfortunately, given the free quota limits of GAE, the whole-day or longer evaluation is not feasible with our current experimental design. To relieve this threat at least in theory, we follow the common experience/suggestion to expect better performance of GAE within off-peak periods, e.g., during nights in the corresponding regions.

Secondly, the extensive experiments for comparing different language runtimes were conducted in one region only, i.e. *asia-northesast1(Tokyo)* by default in our case. We admit that it is impossible to guarantee the same observations on different GAE runtimes in the other regions, unless the experimental results are reproduced in each of them. However, to answer the question thrown in the title, we have already been able to draw generic conclusions from the current investigation (cf. Section VI). In particular, a single runtime (the Java 8 runtime in this study) is enough to confirm that the factor of geographical location does have impacts on GAE's performance. On the other hand, we have designed student assignments of replicating experiments in different GAE regions, in order to supplement our study.

Thirdly, we only employed Fibonacci number calculation and Fibonacci-like string concatenation as benchmarks in this performance engineering study. Due to GAE's strong restrictions on programming flexibility inside its sandbox (e.g., all code in the Python runtime environment must be pure Python), it is extremely challenging to migrate the de facto benchmarks (such as Bonnie++ for storage, NPB benchmark suite for parallel computing, HardInfo for system as a whole, etc.) to different GAE language runtimes. Nevertheless, it is usually valuable to use different benchmarks as different lens to reveal the nature of Cloud services [10]. Driven by this dilemma, we have decided to start from migrating lightweight open-source benchmarks (e.g., STREAM for memory), in order to gradually extend our investigation. As our effort progresses, the ambitious goal will be delivering a PaaS-friendly multi-language benchmark suite.

VI. CONCLUSIONS AND FUTURE WORK

To address the potential ethical threats of parasitic computing and the possible source limits of volunteer computing, we have proposed Bonus computing and tried to employ free quotas of public Cloud services to satisfy particular class of computing demands. The public PaaS GAE is then selected as a suitable source that provides limited while free Cloud resources for our Bonus computing. After all, given its native support for various programming languages that is closely related to a PaaS's productivity [31], GAE has widely been accepted as a flexible and powerful platform for startups and small businesses from the end users' perspective [32].

Given the multiple options of using GAE standard environment, it is beneficial to understand and compare those

different options to avoid choice overload and help squeeze the free computing power. As an initial step, we followed the methodology DoKnowMe and conducted a fundamental investigation into the free-quota performance of GAE's five available runtimes. Our investigation reveals that:

- The PHP runtime can be excluded from our choice list for its poor computation performance. Recall that, as a continuation of parasitic computing and volunteer computing, the bonus computing paradigm is also in favor of distributable and compute-intensive tasks. However, PHP has widely been used as a web development scripting language for dynamic webpage generation rather than for CPU-intensive computation. Furthermore, compared to the other runtimes, PHP seems to have strict response time limit by default. For example, without customer configurations, PHP programs can barely sleep longer than 30 seconds during their execution on GAE.
- Although the Python runtime has been considered to be better developed with various advantages, we also treat its relatively poor computation speed as a significant weakness in bonus computing. Given the identified ten-minute time limit of each continuous execution session, we expect individual bonus computing tasks to be fine-grained and able to be completed as quickly as possible.
- Java 8 is the most competitive runtime in terms of numerical computation on GAE. However, if including memory as a concern, it seems that the default JVM setting makes the Java runtimes lag far behind the other language runtimes in terms of the memory usage efficiency. In other words, GAE does not optimize environment variables of the Java runtime on behalf of customers, and thus Java applications can run out of memory much earlier than the other language runtimes do. On the other hand, JIT also causes regular performance deterioration since it start with a "warm up" to improve computation efficiency and consistency.
- In addition to the difference in language runtimes, GAE also performs differently across its 11 geographical regions. It is not surprising that Google must have deployed heterogeneous hardware resources to construct its datacenters all around the world. From the service provider's perspective, GAE runtime environments need to be largely improved in the regions *southamerica-east1* and *us-central1*. Before the provider's improvement, practitioners had better avoid selecting these regions when employing the GAE service.

Overall, although the Go runtime could have performance deterioration due to GC (cf. Section IV-D), we select Go to be a well-trade-off and easy-to-use GAE runtime environment for our Bonus computing, and selecting the Java 8 runtime to be an alternative that requires optimizing its environment variables manually. Correspondingly, our future work will be unfolded along two directions. On one hand, we will focus on the Java 8 and Go runtimes to conduct more comprehensive experiments (e.g., multi-thread application benchmarking) to deeper understand GAE. On the other hand, we will start implementing Bonus computing prototypes in the Go runtime environment of GAE.

REFERENCES

[1] M. N. Durrani and J. A. Shamsi, "Volunteer computing: Requirements, challenges, and solutions," *J. Netw. Comput. Appl.*, vol. 39, pp. 369–380, Mar. 2014.

[2] A.-L. Barabási, V. W. Freeh, H. Jeong, and J. B. Brockman, "Parasitic computing," *Nature*, vol. 412, pp. 897–984, Aug. 2001.

[3] Z. Li, Y. Chen, M. A. Rodríguez, and L. Deng, "Bonus computing: An evolution from and a supplement to volunteer computing," in *Proc. 27th Int. Conf. Inf. Syst. Develop. (ISD)*, Aug. 2018, pp. 1–12. [Online]. Available: https://aisel.aisnet.org/isd2014/proceedings2018/ISDevelopment/3/

[4] Google. (Oct. 2018). *The App Engine Standard Environment*. [Online]. Available: https://cloud.google.com/appengine/docs/standard/

[5] S. S. Iyengar and M. R. Lepper, "When choice is demotivating: Can one desire too much of a good thing?" *J. Pers. Social Psychol.*, vol. 79, no. 6, pp. 995–1006, 2000.

[6] B. Nice. (Mar. 2017). *A Complete List of Computer Programming Languages*. [Online]. Available: https://medium.com/web-development-zone/a-complete-list-of-computer-programming-languages-1d8bc5a891f

[7] M. Sherman. (Jul. 2015). *Why are There so Many Programming Languages?* [Online]. Available: https://stackoverflow.blog/2015/07/29/why-are-there-so-many-programming-languages/

[8] Stack Overflow. (Jul. 2009). *Choosing Java vs Python On Google App Engine*. [Online]. Available: https://stackoverflow.com/questions/1085898/choosing-java-vs-python-on-google-app-engine

[9] Quora. (Oct. 2015). *Is There Performance Difference Running App Engine With GO, PHP, Python or Java? If There is Which Language Performs Best?* [Online]. Available: https://www.quora.com/Is-there-performance-difference-running-App-Engine-with-Go-PHP-Python-or-Java-If-there-is-which-language-performs-best

[10] Z. Li, H. Zhang, L. O'Brien, R. Cai, and S. Flint, "On evaluating commercial Cloud services: A systematic review," *J. Syst. Softw.*, vol. 86, no. 9, pp. 2371–2393, Sep. 2013.

[11] R. Prodan, M. Sperk, and S. Ostermann, "Evaluating high-performance computing on Google App engine," *IEEE Softw.*, vol. 29, no. 2, pp. 52–58, Mar./Apr. 2012.

[12] P. Fisher, R. Pant, and J. Edberg, *Cloud Computing: Assessing Azure, Amazon EC2, Google App Engine and Hadoop for IT Decision Making and Developer Career Growth*. Berkeley, CA, USA: Apress, Feb. 2010.

[13] Z. Li, L. O'Brien, and M. Kihl, "DoKnowMe: Towards a domain knowledge-driven methodology for performance evaluation," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 43, no. 4, pp. 23–32, Mar. 2016.

[14] E. Betz. (Aug. 2010). *Donated Computer Time Discovers New Star*. [Online]. Available: https://www.insidescience.org/news/donated-computer-time-discovers-new-star

[15] GIMPS. (Oct. 2018). *Great Internet Mersenne Prime Search*. [Online]. Available: https://www.mersenne.org/

[16] A. L. Beberg, D. L. Ensign, G. Jayachandran, S. Khaliq, and V. S. Pande, "Foldinghome: Lessons from eight years of volunteer distributed computing," in *Proc. 8th IEEE Int. Workshop High Perform. Comput. Biol. (HiCOMB) Conjunct 23rd Int. Symp. Parallel Distrib. Process. (IPDPS)*. Rome, Italy, May 2009, pp. 1–8.

[17] G. Lawton, "Developing software online with platform-as-a-service technology," *Computer*, vol. 41, no. 6, pp. 13–15, Jun. 2008.

[18] R. Prodan and M. Sperk, "Scientific computing with Google App engine," *Future Gener. Comput. Syst.*, vol. 29, no. 7, pp. 1851–1859, Sep. 2013.

[19] R. Ramasahayam and R. Deters, "Is the Cloud the answer to scalability of ecologies? Using GAE to enable horizontal scalability," in *Proc. 5th IEEE Int. Conf. Digital Ecosyst. Technol. (IEEE DEST)*. Daejeon, Korea, May/Jun. 2011, pp. 317–323.

[20] I. Shabani, A. Kovaçi, and A. Dika, "Possibilities offered by Google App engine for developing distributed applications using datastore," in *Proc. 6th Int. Conf. Comput. Intell., Commun. Syst. Netw. (CICSyN)*, May 2014, pp. 113–118.

[21] M. Sperk, "Scientific computing in the cloud with Google App engine," M.S. thesis, Fac. Math., Comput. Sci. Phys., Univ. Innsbruck, Innsbruck, Austria, Jan. 2011.

[22] J. Hollingsworth and D. J. Powell, "Teaching Web programming using the Google cloud," in *Proc. 48th Annu. Southeast Reg. Conf. (ACM SE)*. Oxford, MS, USA, Apr. 2010, Art. no. 76.
[23] A. Tabot and M. Hamada, "Mobile learning with Google App engine," in *Proc. IEEE 8th Int. Symp. Embedded Multicore/Manycore SoCs (MCSoc)*, Sep. 2014, pp. 63–67.
[24] S. P. T. Krishnan and J. L. U. Gonzalez, *Google App Engine*. Berkeley, CA, USA: Apress, May 2015, ch. 5, pp. 83–122.
[25] A. de Jonge, *Essential App Engine: Building High-Performance Java Apps with Google App Engine* (Developer's Library). Upper Saddle River, NJ, USA: Addison-Wesley, Oct. 2011.
[26] J. D. Y. Correa and J. A. B. Ricaurte, "Web application deveplopment technologies using Google Web toolkit and Google App engine-java," *IEEE Latin Amer. Trans. (Revista IEEE America Latina)*, vol. 12, no. 2, pp. 372–377, Mar. 2014.
[27] D. Sanderson, *Programming Google App Engine with Python: Build and Run Scalable Python Apps on Google's Infrastructure*. Sebastopol, CA, USA: O'Reilly Media, Jul. 2015.
[28] J. D. Blower, "GIS in the Cloud: Implementing a Web map service on Google App engine," in *Proc. 1st Int. Conf. Exhib. Comput. Geospatial Res. Appl. (COM.Geo)*, Washington, DC, USA, Jun. 2010, Art. no. 34.
[29] M. Malawski, M. Kuźniar, P. Wójcik, and M. Bubak, "How to use Google App engine for free computing," *IEEE Internet Comput.*, vol. 17, no. 1, pp. 50–59, Jan. Feb. 2013.
[30] Google App Engine. (Mar. 2018). *The App Engine Locations*. [Online]. Available: https://cloud.google.com/appengine/docs/locations
[31] O. Gass, H. Meth, and A. Maedche, "PaaS characteristics for productive software development: An evaluation framework," *IEEE Internet Comput.*, vol. 18, no. 1, pp. 56–64, Jan. 2014.
[32] A. Sekhon, "PAAS framework implementation of cloud computing with Google application engine—A review," *Int. J. Comput. Sci. Eng. Technol.*, vol. 6, no. 6, pp. 218–222, Jun. 2016.

ZHENG LI received the B.Eng. degree from Zhengzhou University, the M.Sc.Eng. degree from the Beijing University of Chemical Technology, the M.E. by Research degree from the University of New South Wales, and the Ph.D. degree from Australian National University. He was a Postdoctoral Researcher with the Cloud Control Group, Lund University, Sweden. He was also a Visiting Research Fellow with the Software Institute, Nanjing University, China, and with Data61, CSIRO, Australia. Before studying abroad, he had around four-year industrial experience in China after receiving his M.Sc.Eng. degree. He is currently an Assistant Professor in software engineering and computer science with the University of Concepción, Chile. His research interests include big data analytics, cloud computing, performance engineering, and empirical software engineering.

XUE GUO is currently a Research Master Student with the Software Institute, Nanjing University, China. She has been active as a volunteer to serve the research community, including multiple international conferences. In addition to having scientific papers published in the Web service composition domain, she also attended an outreach program toward the collaboration with industry. Her research interests mainly include service-oriented computing and empirical software engineering.