Software Verification in the Google App-Engine Cloud

Dirk Beyer, Georg Dresler, and Philipp Wendler

University of Passau, Germany

Abstract. Software verification often requires a large amount of computing resources. In the last years, cloud services emerged as an inexpensive, flexible, and energy-efficient source of computing power. We have investigated if such cloud resources can be used effectively for verification. We chose the platform-as-a-service offer Google App Engine and ported the open-source verification framework CPAchecker to it. We provide our new verification service as a web front-end to users who wish to solve single verification tasks (tutorial usage), and an API for integrating the service into existing verification infrastructures (massively parallel bulk usage). We experimentally evaluate the effectiveness of this service and show that it can be successfully used to offload verification work to the cloud, considerably sparing local verification resources.

1 Introduction

Software verification usually requires a large amount of computation resources. In practice, it is often not only a single verification task that needs to be solved, but a large quantity of individual tasks. This occurs for example in regression verification, where the correctness of all components of a system has to be re-established after some development work. For illustration, let us consider Linux driver verification: there are approximately 1 200 commits per week affecting on average 4 device drivers. Assuming that each changed driver is verified against only 100 safety properties after each commit, and that only 12 s of run time are necessary per verification task, the weekly verification time would sum up to 67 days. Those tasks are usually independent and can be run in parallel to reduce the time until the answers are available to the developers. Instead of buying and maintaining an expensive cluster of machines for occasional peaks of computational load, we can also move the actual verification execution into a computing cloud, where resources are available on demand. This enables a verification process that is less expensive (only actual usage is paid) and faster (higher degree of parallelism).

Two of the different flavors of computing cloud services are suitable for implementing a cloud-service-based verification system: Infrastructure as a Service (IaaS) and Platform as a Service (PaaS). For IaaS, a large number of virtual machines (VMs) is reserved, and expenses incur only for the actual uptime of the machines. Amazon's Elastic Compute Cloud (EC2) is a popular example for IaaS. The customer is responsible for all setup work for the VM, including setup of

A. Biere and R. Bloem (Eds.): CAV 2014, LNCS 8559, pp. 327–333, 2014.

the operating system and applications, and an infrastructure for load-balancing (starting VMs as necessary) needs to be implemented by the customer. For PaaS, the customer is allowed to run own applications on an application server, and expenses incur only for the actual consumption of resources by the application. Google's App Engine is a popular example for PaaS. The PaaS provider operates the application server and will automatically run the application on as many machine instances as necessary depending on the demand. This provides a high degree of scalability without administrative effort by the customer.

We are interested to evaluate the applicability of the Google App Engine as verification infrastructure. There are a number of requirements that an application has to satisfy in order to be runnable on a PaaS. For example, in a typical PaaS environment, the application has no or only limited direct access to external services such as the file system, and the application needs to be integrated using specific APIs for serving user requests. Due to the traditionally high resource consumption of verification tools and the restricted environment, we wanted to investigate if an effective and efficient verification service can be implemented based on the Google App Engine. The convenient scalability and the eliminated administration effort make it a promising approach.

Related Work. Several approaches exist to distribute a single verification task across multiple machines [7, 11, 12], for which IaaS clouds can be used as a source of a high number of virtual machines. Such techniques usually do not scale perfectly with the number of machines due to the communication effort, and require specialized verification algorithms. We focus instead on distributing many independent tasks, which works with any existing automatic verification technique and does not require communication between the worker machines. This concept is used in other areas of computation for a long time, but was not yet evaluated for automatic software verification. Also the applicability of a restricted environment and a PaaS offer for verification was not yet studied.

The idea of providing a web front-end for verification services which is usable with a browser is not new, and several such services are available from different groups[1,2,3,4]. These are intended to serve for demonstration and evaluation purposes, and not as a possibility to offload high-volume verification load.

2 Background

Google App Engine. The Google App Engine [13] is a PaaS offer to run web applications on Google's infrastructure. It provides services that are designed to be scalable, reliable, and highly available under heavy load or huge amounts of data. Scaling and load balancing are automatically adjusted to the needs of the application. The App Engine allows to run applications in Java, Python, PHP,

[1] Multiple tools from Microsoft and others: `http://rise4fun.com`
[2] Aprove: `http://aprove.informatik.rwth-aachen.de/index_llvm.asp`
[3] Divine: `http://divine.fi.muni.cz/try.cgi`
[4] Interproc: `http://pop-art.inrialpes.fr/interproc/interprocweb.cgi`

or Go. Applications are executed in a sandbox that abstracts the underlying operating system, and provides a secure and isolated environment.

An App-Engine application provides specialized request handlers, which represent the entry points into the application. To serve requests, an application has a pool of zero or more instances allocated. Instances are long-living containers for request handlers that retain local memory and are initialized ahead of incoming requests. The use of a single instance with the smallest configuration of 128 MB of RAM and a 600 MHz CPU for one hour counts as one 'instance hour' and costs 0.05 USD as of May 2014[5]. There are more powerful instances available (up to 1 GB of RAM and a 4.8 GHz CPU) that consume the instance hours at a higher rate. The App Engine also offers a schema-less data store and a task-queue service, which is used to enqueue tasks for execution in the background independent from user interaction.

Restrictions. To provide abstraction from the operating system and to ensure security, the Java run-time environment of the App Engine restricts access to a specific set of classes in the Java standard library[6]. The most important of the forbidden actions are file-system writes, starting external processes, and loading native libraries. Data need to be stored using the data-store service or the Google Cloud Storage. For the other operations, no alternatives are possible except implementing all functionality in Java code. There are also some restrictions on the resource usage of applications[7]. Request handlers are expected to terminate quickly (in under 60 s), but tasks in the task queue are allowed to take up to 10 minutes. The data store takes entities up to a size of 1 MB, which poses a problem with large log or source-code files.

Billing. The pricing model of the App Engine specifies the cost for each resource in detail, and charges incur only for the resources that were actually used. Most resources are freely available up to a resource-specific quota[8]. For example, 28 instance hours can be used free of charge each day.

CPAchecker. CPAchecker [3] is an open-source framework for software verification, which is available online[9] under the Apache 2.0 license. It is based on the concept of Configurable Program Analysis [2], which supports the integration of different verification components. CPAchecker implements a wide range of well-known verification techniques, such as lazy abstraction [10], CEGAR [9], predicate abstraction [4], bounded model checking [6], and explicit-value analysis [5]. It is platform-independent because it is implemented entirely in Java (including its libraries, e.g., the Java-based SMT solver SMTInterpol [8]).

[5] `https://developers.google.com/appengine/pricing`
[6] `https://developers.google.com/appengine/docs/java/jrewhitelist`
[7] `https://developers.google.com/appengine/docs/java/backends/`
[8] `https://developers.google.com/appengine/docs/quotas`
[9] `http://cpachecker.sosy-lab.org`

3 Verification in the Google App-Engine Cloud

Porting CPAchecker. It is in principle possible to use all analyses of CPAchecker in the Google App Engine because it is written in Java. Due to the above-mentioned restrictions, some adoptions were necessary. Most features of CPAchecker that rely on disabled Java APIs are either optional or non-critical, and could thus be turned off with CPAchecker's own configuration mechanism. This includes, for example, extended time and memory measurements, and counterexample checks with the external checker CBMC (which is written in C++ and thus not portable to the App Engine). If an SMT solver is needed, we use SMTInterpol, which is written in Java. The major obstacle for porting CPAchecker to the App Engine was to re-design file-system writes. CPAchecker expected the source-code file on the disk and would usually write several output files with information about the analyzed program and perhaps a counterexample. While the output files are optional, they provide helpful information to the user and thus should be available. Thus, we integrated an abstraction layer for all file-system operations of CPAchecker that re-routes the reading of the input program and the writing of all output files to the datastore service. Apart from minor other adoptions, these were the only changes to CPAchecker. All of our work was integrated into the CPAchecker project and is available as open source from its repository.

API for Bulk Usage. The most important application of our cloud-based verification service is solving a large quantity of verification tasks, as in regression verification. We developed an API for automatically submitting tasks and retrieving results. We integrated a client for this API in CPAchecker's execution infrastructure, such that in terms of user interaction, there is no difference between running the verification tasks locally or using the App Engine. Due to the scalability of the App Engine, the results will be available quickly because many verification tasks can be solved in parallel. Another application of the verification-service API is an integration in situations where verification is needed but resources are limited or a verifier might not be available. For example, using the verification service inside an IDE plug-in would make it easier and faster for developers to verify their code.

Front-End for Tutorial Usage. The second channel of access is provided for users who wish to try out CPAchecker, or experiment with software verification in general: we provide a web-based user interface that is easy to use in a web browser and requires no installation effort from the user. The user uploads or enters a program, selects a specification that the program should satisfy, and chooses a configuration of the verifier. After starting the verification run, the user is kept informed about the current status of the task, and the result is provided after the run is finished. Further output like log files, statistics, and information about counterexamples are presented and available for download. The front-end that we implemented is available online[10].

[10] `http://cpachecker.appspot.com`

4 Experimental Evaluation

To evaluate the effectiveness and efficiency of the Google App-Engine cloud for verification purposes, we run CPAchecker in version `1.3.2-cav14` on verification tasks from the International Competition on Software Verification (SV-COMP'14) [1]. We compare the amount of successfully verified programs and the verification time to a local execution of the verifier. To show the applicability of the cloud service, we use two verification approaches that have different characteristics with regard to performance and resource requirements: an explicit-value analysis [5] and a predicate analysis [4] (using the SMT solver SMTInterpol).

Setup. We limit the wall time for each verification task to 9 minutes. We did not use any limits for CPU time, because this is not supported by the App Engine. In the App Engine, we used the default of the available instances, which provides 128 MB of RAM and 600 MHz of CPU frequency. For a direct comparison, we also limited the size of the Java heap memory to 128 MB for the local executions. For desktop machines, this is a rather low limit as current machines provide much more RAM. Thus, we additionally ran the same analyses with a heap size of 4 096 MB. In both cases, we assigned one CPU core (plus one hyper-threading core) of an Intel Core i7-2600 quad-core CPU with 3.4 GHz. In the App Engine, we reserved 100 instances at the same time. For local execution, we ran 4 tasks in parallel on the same machine. We selected those categories from the SV-COMP repository[11] as benchmark verification tasks that are well-supported by the chosen analyses: ControlFlow, DeviceDrivers64, SequentializedConcurrent, and Simple. We excluded programs whose source code was larger than 1 MB (restriction by the data store). This resulted in 2 458 program files written in C.

Results. Table 1 shows a summary of the results. For each configuration, we list the number of successfully computed answers, and the CPU time that was necessary to compute them. We also show the wall time that elapsed between start and end of the benchmark, i.e., the time the user has to wait for all results. Both times are rounded to two significant digits. In Fig. 1 we show quantile functions for the successful results (i.e., the verifier returned an answer) of all configurations. The results are sorted by their run time, i.e., a data point (x, y) means that the respective configuration has successfully verified x programs in at most y seconds each. The area under a graph represents the sum of the CPU time that is necessary for computing the answers (the lower a graph is, the faster a configuration is). This value can also be seen in the row 'CPU Time' of Table 1. The further to the right a graph stretches, the more answers were returned by a configuration. Dark lines (red, blue) in the plot show executions in the App Engine, the corresponding light lines (orange, cyan) show the local executions.

The plot shows that the App Engine is actually often faster. This is due to the relatively long startup time of a JVM on the local machine (almost 2 s), which is not needed in the App Engine. The table and the graph both show that CPAchecker running in the App Engine is not able to verify as many programs as locally within the same time limit, and needs more CPU time. Impressively,

[11] `https://svn.sosy-lab.org/software/sv-benchmarks/trunk/c/`

Table 1. Summary of results comparing App-Engine execution with local execution

Analysis		Explicit-Value			Predicate		
Location		App Engine	Local		App Engine	Local	
CPU Frequency		600 MHz	3.4 GHz	3.4 GHz	600 MHz	3.4 GHz	3.4 GHz
Heap Size		128 MB	128 MB	4 096 MB	128 MB	128 MB	4 096 MB
Successful:	No. of Results	1 842	1 920	2 021	1 771	1 952	2 012
	CPU Time (s)	16 000	13 000	31 000	41 000	39 000	50 000
Total:	Wall Time (s)	11 000	30 000	53 000	9 900	46 000	58 000
Effective Parallelization		25	4	4	30	4	4

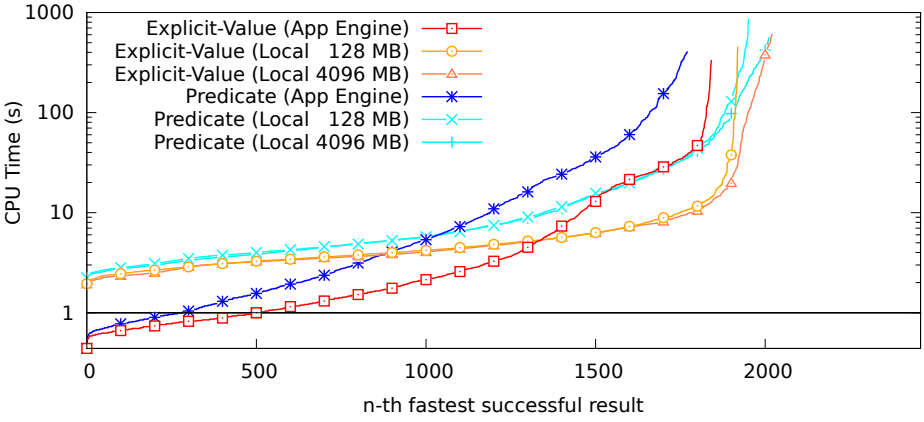

Fig. 1. Quantile functions showing the CPU time for the successful results; symbols at every 100-th data point; linear scale between 0 s and 1 s, logarithmic scale beyond

the difference in the number of results is only about 10 %. Note that we used the rather slow standard instances in the App Engine which provide a much lower CPU speed than our local machine. More powerful instances would be available as well (at a higher price). Furthermore, the row 'Total Wall Time' in Table 1 shows that due to the high scalability of the cloud and the massive parallelism, the total waiting time for the user is much lower (3 hours instead of 8 to 16 hours), even though we ran 4 tasks in parallel locally. The effective parallelization in the App Engine is less than the number of instances (100) due to queue saturation problems which could be fixed with an improved implementation. More details on this issue can be found on the supplementary webpage[12].

Running all 4 916 tasks in the cloud cost 38.09 USD where the explicit-value and predicate analysis consumed 17.78 USD and 20.31 USD, respectively. All experiments that we ran for the preparation of this paper cost only 185.72 USD in total (for obtaining valid results, we had to run the experiments several times). The experiments were done when the prices were still higher, with prices of May 2014 the cost would have been 38 % less.

[12] `http://www.sosy-lab.org/~dbeyer/cpa-appengine`

5 Conclusion

We ported the successful open-source verification framework CPAchecker to the Google App Engine, and have shown that cloud-based verification is an effective way to gain scalability and a high degree of parallelism, allowing users to receive verification results much faster. This new verification service enables a convenient integration of software verification into development environments that do not support the execution of a verification engine locally. It also provides a convenient way for tutorial-like experiments with a verifier without any installation effort.

References

1. Beyer, D.: Status report on software verification (competition summary SV-COMP 2014). In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 373–388. Springer, Heidelberg (2014)
2. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 504–518. Springer, Heidelberg (2007)
3. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
4. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Bloem, R., Sharygina, N. (eds.) FMCAD 2010, pp. 189–197. FMCAD (2010)
5. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Cortellessa, V., Varró, D. (eds.) FASE 2013. LNCS, vol. 7793, pp. 146–162. Springer, Heidelberg (2013)
6. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS/ETAPS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
7. Breuer, P.T., Pickin, S.: Open source verification under a cloud. ECEASST 33 (2010)
8. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: An interpolating SMT solver. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 248–254. Springer, Heidelberg (2012)
9. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
10. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Launchbury, J., Mitchell, J. C. (eds.) POPL 2002, pp. 58–70. ACM (2002)
11. Holzmann, G.J., Joshi, R., Groce, A.: Swarm verification. In: Inverardi, P., Ireland, A., Visser, W. (eds.) ASE 2008, pp. 1–6. IEEE (2008)
12. Lerda, F., Sisto, R.: Distributed-memory model checking with SPIN. In: Dams, D., Gerth, R., Leue, S., Massink, M. (eds.) SPIN 1999. LNCS, vol. 1680, pp. 22–39. Springer, Heidelberg (1999)
13. Sanderson, D.: Programming Google App Engine. O'Reilly (2012)