A Case for Static Analyzers in the Cloud (Position paper)

Michael Barnett[1] Mehdi Bouaziz[2] Manuel Fähndrich[1] Francesco Logozzo[1]

[1] Microsoft Research, Redmond, WA (USA)
[2] École normale supérieure, Paris (France)

Abstract. We describe our ongoing effort of moving a desktop static analyzer, Clousot, into a cloud-based one, Cloudot. A cloud-based static analyzer runs as a service. Clients issue analysis requests through the local network or over the internet. The analysis takes advantage of the large computation resources offered by the cloud: the underlying infrastructure ensures scaling and virtually unlimited storage. Cloud-based analyzers may relax performance-precision trade-offs usually associated with desktop-based analyzers. More cores enable more precise and responsive analyses. More storage enables full caching of the analysis results, shareable among different clients, and queryable off-line. To realize these advantages, cloud-based analyzers need to be architected differently than desktop ones.

1 Desktop-based static analyzers

Traditionally, the design of static analyzers focuses on desktop users. It is therefore strongly influenced by the performance trade-offs made in order to run smoothly in such a setting.

A (classical) desktop static analyzer is a batch process. It takes as input a program or a library, either in source or in binary form, it analyzes it, and it reports possible errors to the user. The analysis phase, the most important phase of the whole process, is usually performed sequentially on the local machine. Once the analysis is done, the intermediate computation is discarded, so that a new analysis of the same program, or that of a slightly modified version, would start from scratch. The analysis process is quite expensive, and on realistic code, it can vary from a few minutes to several hours. A re-analysis from scratch significantly reduces the usability of an analyzer when applied to large code bases. Furthermore, the analysis effort is not shared among a team of software engineers working on the same code base, even though most of the code and analysis options will be identical. Each run of the analyzer is isolated, no information nor computation is shared.

A way to solve those problems is to run the analysis on a centralized server: desktop users send a request to the server. The server is responsible for processing the analysis requests. It can store the results for future reuse. It keeps a database with all the inputs and the respective outcomes: when a client requests the

analysis of a program already in the database, the server can simply replay the outcome, without analyzing it from scratch.

A problem with the centralized server solution is scalability: in the case of medium or large teams of developers, it may be the case that too many analysis requests are issued, causing lag in the response, or overloading the server. Adding more computing power to the server (more cores, memory) may solve the problem; more clients can be served in parallel, assuming each analysis requires one core. A centralized server has the drawback that there is no elasticity in allocating resources—most of the server resources may be unutilized, waiting for peaks of requests. Furthermore, each organization would have to maintain its own server hardware, the service itself, software and hardware updates, *etc.*—a significant cost. On top of this, no sharing can be achieved among different organizations, *e.g.*, in the case of projects jointly developed by multiple organizations.

2 Static analyzers as Services in the Cloud

An even better solution than a centralized server is to let the analyzer run on top of a cloud infrastructure, as SaaS (Software as a Service). The cloud infrastructure provides extremely large computational resources—large numbers of cores and huge amounts of storage. The service can be configured to be elastic with respect to incoming analysis requests: the more requests, the more allocated resources. When requests subside, the analyzer releases the extra resources.

Hosting a cloud based analysis service means that there is only one version of the static analyzer. This simplifies support and bug fixing. Tool developers do not need to support different versions of the analyzer, in different configurations, on different platforms. Furthermore, every bug fix in the analyzer is immediately and transparently available to all the users. On the other hand, deploying and testing newer version of the analysis service does require careful staging and maintenance of at least the production and the next-gen version of the service.

Different service requests, from different users, share the results of the analysis. Suppose that the underlying static analyzer has functionality to identify previously analyzed code and to reuse the analysis results (*e.g.*, the inferred abstract states). Now, for each service request, the analyzer can save the intermediate steps of the computation—assuming the cloud provides virtually unlimited amounts of storage. While serving a request, if the analyzer detects that a piece of code has already been analyzed in a previous request, it can simply reuse that result. Therefore, the cloud enables an aggressive and transparent sharing of the analysis among projects and teams.

3 Perspectives

The cloud offers new and exciting research challenges and opportunities. It provides a (relatively) cheap massive distributed environment. The design of the static analyzer should take into account the large number of available cores. In

particular, we expect the analysis to be faster and more precise. It can be faster because the computation can be parallelized on many nodes. The analysis can also be more precise because the extra resources enable the analysis to use more CPU and memory intensive algorithms than are practical on the desktop.

But how do we structure the static analyzer to fully exploit the cloud infrastructure? For instance, how can we split and parallelize the computation for a global program analysis to make it scalable to very large programs? In Sect. 6 we discuss our current design for Cloudot, our version of the abstract interpretation-based .NET static analyzer to run in the cloud. We designed new distributed and highly asynchronous fixpoint algorithms for static analysis. Chaotic asynchronous iterations [2] are the way to go. Designed in the 70's, theses methods for solving fixpoint systems of monotonic equations on multiprocessor machines are proven to converge to the same result as iterative methods. However, we must explore what happens when the analysis functions are not monotonic, *e.g.*, this is the (common) case when using infinite abstract domains with widening or finite domains with k-limiting. For non-monotonic functions, the analysis may never reach a fixpoint, *e.g.*, it may end up alternating between two incomparable abstract values, leading to two different warnings, which is not an acceptable situation for an end-user. How to deal with those problems is an open research problem.

We expect data collection and mining to be one of the main advantages of cloud-based static analyzers. We can store all the analyzed programs, with all the analysis options and the intermediate analysis results. A first immediate benefit is the ability to collect data on the effective usage of the tool: how often and how many clients use the tool, which options or abstract domains are most/least used, which warnings get fixed, which ones are simply suppressed or ignored, *etc.* By exploring the warnings, we can infer the most common precision weaknesses of the analyzer, and then either refine the analyzer or design new abstract domains tailored to the warnings. In summary, a static analyzer in the cloud will enable data-driven design of static analyses and abstract domains. In our opinion, this will be a giant step enabling a better and deeper understanding of the usage and performance of static analysis tools in the wild.

We can leverage the analysis results on previous versions of a program to perform semantic-guided warning suppression. The analyzer can learn from false warnings to reduce the noise in future versions of the same code base. False warnings are marked by the user, for instance using a suppress warnings mechanism. Ideally, the analyzer will be able to adapt to the coding style of the code base, learning code patterns used in such a code base, and hence automatically reducing the false warnings ratio.

The static analyzer can provide new, semantic metrics on the code to help manage the quality of large projects. It can report how the ratio of warnings per line of code changes from version to version, the evolution over time of the warning ratio for a particular piece of code, which module in the code base has most/least warnings, *etc.* It can combine the data with testing and concrete bug reports to figure out the less stable parts program components.

The user can query the stored data, in order to use it, *e.g.*, for code reviews. She can ask complex semantic queries on the program such as which check-in introduced a warning, which caller passed a negative argument to this method, what happens if this variable is in this range, *etc.* The cloud can cache all such queries, so as to optimize their execution and share their result.

4 Clients

The simplest client for the cloud static analyzer simply uploads a program (the source or a compiled binary). Then it issues the analysis request to the cloud service. When the analysis is done, the client reads back the results and shows them to the user, as a text file, or squiggles in the editor. As the bulk of the analyzer is in the cloud, users do not need to install, configure, or update the analyzer. A tiny client suffices.

A more refined client may exploit the massive parallelism of the cloud. It can request the parallel analysis of the same program but with different analysis options. The options can specify different abstract domains (more or less precise) for the heap and numerical values but also different treatments of method calls (fully modular, summary based, fully inline), join points (merge or not), *etc.* The user will first see the results from the most imprecise (usually the first to be completed) analysis requests, and then these results will be refined over time with more precise (but more time intensive) ones. The main advantage is to obtain very quick but coarse analysis feedback, and improve the precision over time.

The client can be integrated in the project build system. When a new version of the code base is checked in, the build system issues a request to the cloud analyzer and the results are incorporated in the build log.

A hybrid client can partition the computation between the local desktop and the remote cloud. In an interactive setting, the local client only analyzes the code visible on the screen, while the rest of the project is analyzed in the cloud. As a consequence, the user will get immediate feedback for "visible" code, whereas the cloud analyzes the "invisible" code. As the user scrolls to other parts, the newly visible code analysis may already be in the cache and immediately displayed.

As an alternative, the client can also be a rich Web application. The programmer edits the code in a browser application. The code is automatically saved on the cloud and analyzed in the background by the cloud static analyzer. The user will see not only the warnings in real-time but also possible fixes for them [6].

5 Other Considerations

Static analysis can be easily modularized on annotated code, where annotations typically take the form of pre- and post-conditions on methods and object invariants on types. For code under development, it is best if these so-called contracts are directly written in the code itself, making them available during analysis and readable to programmers. However, often, programmers use 3rd

party components that lack contracts and a system of out-of-band contracts to annotate these components is necessary. In a cloud setting, these out-of-band contracts can be more easily distributed, so that the analyzer always has the most up-to-date version of the contracts. Furthermore, a centralized place for these out-of-band contracts makes it possible to crowd-source them—a big advantage, as writing and maintaining them is very time consuming. We envision a client tool to visualize and update the out-of-band contracts from anywhere.

6 Cloudot

We are implementing the vision outlined above in Cloudot—Clousot in the cloud. Our starting point is the code contract static checker cccheck/Clousot [5].

Clousot is a static analyzer for .NET based on abstract interpretation [3]. Clousot analyzes methods in isolation. For each method `m` in the program, it assumes its precondition, and it asserts the postcondition. For each method `n` called from `m`, it asserts the precondition of `n` and it assumes its postcondition. The output is an instrumented method `m'`. Clousot has three phases. First, it analyzes the instrumented `m'` to infer invariants for each program point. Second, it uses this information to prove assertions, either user-provided (*e.g.*, preconditions) or language-induced (*e.g.*, division by zero). Third, it generates verified repairs, a precondition [4] and a postcondition to be propagated to the callers, and an object invariant to be used in other type members [1]. Clousot uses a SQL database (local or remote) to cache the results of the analysis. It uses the hash of the instrumented `m'` as a key into the database. If it finds an entry with such a key, then it simply reads the associated data, *i.e.*, the outputs of the phases two and three above.

Our first step to turn Clousot into Cloudot was to make it able to run as a standalone service. Services should be fully functional: they take a request, process it, and answer it. The answer should only depend on the input, not the history of the serviced requests. Cloudot accepts as input a binary and a set of options, describing among other things the abstract domains for the analysis and the reference assemblies containing the contracts. The output is a list of warnings and code repairs (including inferred contracts). We modified Clousot to run as a service, essentially turning all the static variables into instance variables. Furthermore, we worked out the (complex) details to make the service run remotely. Eventually, this gives us a first level of parallelism: we serve each request independently from all the others, *i.e.*, we run each analysis request on a different core. Internally, we cache the assembly analysis by using a shared database. In practice this first version of Cloudot corresponds to the scenario of a centralized analysis server sketched in Sect. 1.

Next, we wanted to make Cloudot internally parallel, so as to achieve massive parallelism: ideally, we would assign a core to each method analysis, and perform a global fixpoint computation to propagate information among methods. Clousot is a (rather complex) sequential analyzer. Converting it to make it parallel would be a strenuous engineering task. Therefore, we chose a different

solution. When Cloudot gets a request, it slices the binary into minimal analyzable units (MAUs). MAUs are smaller, self-contained binaries containing at the minimum a single method along with all metadata and contracts of program elements referenced by that method. Unlike executable slices, MAUs do not need to include other methods' body. Cloudot pushes MAUs into a shared queue. On the other side of the queue, workers (essentially Clousot services) pull work from the queue, perform the analysis, write the results into a database, and notify the completion of the MAU analysis. Cloudot checks if the analysis of a MAU has reached a fixpoint, *i.e.*, its output is stable. If it is not the case, Cloudot loads the dependencies of the MAU (essentially the callers and the callees of methods in the MAU, and the other methods in the type) and it pushes them again in the queue for analysis. Cloudot iterates until all the MAUs are stable. We are still working on assuring that the computation of the global fixpoint always converges.

7 Conclusions

We advocate the development of static analysis tools in the cloud. The cloud provides huge computational resources that can be used to improve the performance and precision of analysis tools. The designer of a cloud-based analyzer is free to relax most of the usual precision-performance trade-offs adopted for desktop-based static analyzers. On the other hand, the analyzer must be re-architected to exploit the model of computation of the cloud infrastructure. We need to devise new algorithms, for instance for asynchronous fixpoint computation of non-monotonic functions, for the sharing of computation, and for load balancing between local and remote computation. We have started the exploration of these issues in our ongoing effort on Cloudot.

Acknowledgements The authors wish to acknowledge Tom Ball and the anonymous reviewers for their helpful comments on the manuscript.

References

1. M. Bouaziz, F. Logozzo, and M. Fähndrich. Inference of necessary field conditions with abstract interpretation. In *APLAS*. Springer, 2012.
2. P. Cousot. Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice. Technical report, Laboratoire IMAG, University of Grenoble, 1977.
3. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*. ACM, January 1977.
4. P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo. Automatic inference of necessary preconditions. In *VMCAI*. Springer, 2013.
5. M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS*. Springer, 2010.
6. F. Logozzo and T. Ball. Modular and verified automatic program repair. In *OOPSLA*. ACM, 2012.