SCALABLE SEMANTIC VIRTUAL MACHINE FRAMEWORK FOR LANGUAGE-AGNOSTIC STATIC ANALYSIS

Maxim Menshchikov

Saint Petersburg State University

E-mail: maximmenshchikov@gmail.com

The more static program analysis spreads in the industry, the clearer it becomes that its real-world usage requires something more than just optimized algorithms to make execution fast. Distribution of computations to cluster nodes is one of the viable ways for speeding up the process. The research demonstrates one of the approaches for organizing distributed static analysis – "Semantic Virtual Machines", which is based upon symbolic execution of language-agnostic intermediate codes of programs. These codes belong to the developed language for program representation with semantic objects behind. As objects are fully serializable, they can be saved to and later retrieved from a distributed database. That opens up a possibility for making an analyzer spread to many cluster nodes. Semantic Hypervisor is the core of the system which acts as a master node for managing runnable virtual machines and also a relay between them and a semantics storage. Main efforts are put into the development of intercommunication mechanisms which can be effective in most of the scenarios, as well as into integration of the approach in existing static analyzer. The study shows the architecture, describes its advantages and disadvantages, suggests solutions for the biggest problems observed during research. The empirical study shows improved performance compared to a single node run, and the result is almost linearly scalable due to the high locality of data. The main result is the created scalable and unified language-agnostic architecture for static analysis with semantics data in a distributed storage behind of it. The significance of the research is in high achieved performance level alongside with a clean architecture.

Keywords: static analysis, distribution, scalability, semantic virtual machine, semantic hypervisor

1. Introduction

Static analysis has an imminent problem: less precision and more abstractions lead to the loss of detection quality, higher precision leads to an increased execution time. Real-world applications tend to increase source code size with the time, the state space size grows even faster. Static analysis can benefit from distributed computations, but the lack of effectiveness makes it quite questionable whether it should be used or not. The paper suggests one of the approaches which shows good results in our testing. The novelty of the approach is that intermediate code is distributed rather than the source code of an application, and a good level of parallelism is achieved for intraprocedural and in some ways interprocedural analysis types.

About the project. Our analyzer is a work-in-progress tool aimed at finding bugs in programs via abstract interpretation and proving their absence using model checking, which is done in a language-agnostic manner using syntax unification and “semantic virtual machine” intermediate representation, which is described in a paper.

2. Preliminary conversions

Before doing the core part of the analysis, source codes have to be read. As multiple languages are supported, generalized abstract syntax trees are retrieved and converted to language-agnostic intermediate codes. That is done in MapReduce [2] fashion according to an algorithm described in [1], except that intermediate codes are generated. The key to distributiveness is that all objects are serializable and can be stored in a NoSQL database like MongoDB, being merged at a later stage.

To distribute files among nodes, we use a greedy algorithm in which the suggested input size is estimated for each node, after that files are allocated to nodes, taking the biggest file at each step. This algorithm is suboptimal, but measuring code complexity is even costlier, while still providing an inaccurate measure of full analysis time.

2.1. Virtual Machine language

Key properties of our intermediate codes are defined as follows. Commands are simple to read and process, and expression grammar isn’t defined by language itself. We consider the expression language another entity which is supported throughout the static analyzer. As in [1], the semantics is based on resources, expressions, and fragments. Fragments, which are essentially groups of expressions with similar properties, make up an execution flow but are not preserved during the transformation to virtual machine codes. Each variable or a function has the corresponding resource assigned. Expressions help apply actions to operands like resources and other expressions. Precise modeling of memory accesses is made using immutable constant references, constant values, and mutable variables, which are used only for the propagation of return values during function inlining, while the first two object types help simulate low-level load/store semantics.

Our command set includes the instruction for declarative resource preloading – declare (resource). Usually resource declaration can be omitted, however, there are a few exceptions. First of all is dynamic arrays, which size is defined at exact code point. Second is a case of preloading resources from a distributed database before its use, resulting in an improved performance. Other instructions are:

- assign, init, branch [loop], end branch, invoke, return, ... – more common commands.
- check [false expr], constraint [true expr] – constraint system management commands.

The example of C → IR transformation is presented in Table 1.

Table 1. Various code representations

Regular code in C	**Function-separated intermediate codes**	**Inlined intermediate codes**

int **f**(int *x*)
{
  if (*x* == 0)
  return *1*;
  return *0*;
}
printf(*«%d»*, *f(4)*);
	f: **enter function** (*Variable(x)*)
	  branch *x* == 0
	  return *1*
	  end branch
	  return *0*
	  exit function
	declare *f*
	init *ConstVal(cv1) = 4*
	assign *Variable(result) = 0*
	invoke *f* **with arguments** *cv1* **store to** *Variable(result)*
	init *ConstVal(cv2) = «%d»*
	invoke *printf* **with arguments** *cv2, result*
		init ConstVal(cv1) = 4
		check cv1 == 0
		assign Variable(result) = 0
		init ConstVal(cv2) = «%d»
		invoke printf **WITH ARGUMENTS** cv2, result

3. Semantic virtual machines

When the preparation stage is done, the semantics is collected for the whole program and intermediate codes are generated for each procedure. In most of the cases, this is more than enough for performing Model Checking or Abstract Interpretation routines. These algorithms are out of the scope of this study, however, the framework for doing such processing is quite generic.

The framework is based upon processing of virtual machine codes with a semantic storage behind. The hypervisor is the core of the framework, which is directly connected to a semantic database. Having brief knowledge of the target program, it manages entry point selection and virtual machine startup. The virtual machine is more like a code interpreter than a physical machine emulator in a traditional sense. It executes the codes and applies static analysis algorithms. It contacts hypervisor to get some semantic data when needed. This process is called object lookup, and the core idea is that objects can be obtained either from local high-performant storage, or from a global storage, or from node-specific local storages. If the object isn't found after all, it is created with deducted type and marked as non-existent. In proper programs, such objects always have existent counterparts.

Hypervisor runs virtual machines on some entry points in generalization (intraprocedural) or instantiation (interprocedural) modes. The typically required semantics is preloaded to a virtual machine instance – e.g. semantics referenced in headers is definitely required, as well as one introduced by the source file. Semantics introduced by other files is loaded upon request (lazy semantics loading). This logic makes data highly local with little penalties when loading other parts. The excessive data is unloaded to save memory (automatic semantics unloading). It is essential to simplify the code as much as possible during generalization or IR translation as it dramatically improves performance. The other possible optimization is to allocate virtual machines on a per-module basis, especially taking network topology into account (per-module VM allocation).

4. Evaluation

The developed approach had been integrated into a static analyzer. For testing, we used our yet-to-be-published Verification Example Framework with many defects of various kinds, and results are outlined in Table 2. Previously the algorithm without intermediate code representation had been used, but we found it hard enough to establish fast and generalized enough language-agnostic analysis. Without virtual machines, too many aspects had to be taken into account due to the redundancy of human-readable source code forms. Yet the usage of assembler-like presentation with more atomic store operations would have made analysis miss important facts about a program. The semantic virtual machine run is definitely taking more time than a regular run would, but with the right degree of parallelism, it is fully compensated. The higher precision rate also justifies its usage.

Table 2. Test results

Approach	**Performance gain (%)**	**Precision (% of found issues)**
Semantic Virtual Machine	100	100
Distributed Semantic VM (4 nodes)	369	100
Distributed Semantic VM (8 nodes)	720	100
VM with assembler-like input	114	63
Generic semantics-driven analysis	176	80

4.1. Problems observed during evaluation

During the evaluation, we observed a few issues which are mostly related to static analysis field problems rather than intercommunication mechanisms. First of all, the process of line/column keeping within error reports is imminently ineffective as each expression is ought to allocate (file ID, location) pairs. For that, we used LLVM/Clang-like solution in which only raw position within files is preserved and the diagnostics engine has to cache files and lines before showing reports. Second, we found that low-level-like IR programs with immutable objects require excessive processing of loops. A single node can identify variables by their in-memory addresses, but loop unrolling or function inlining requires a “refresh” of IR representation, replacing old variables and constant references or values with newly allocated counterparts. Persistent storage depends on object ID but essentially has the same issue. Reallocation isn’t always needed as ID can be changed in-place when the previous usage had been committed to SMT program or abstract interpretation lattices. Clearly, this is the biggest issue of a semantic database-driven language. The last problem is that performance gain is decreasing when the code base becomes more cross-dependent. For intraprocedural analysis it isn’t a big problem, especially with resume-based verification as the processing of models is done early, but interprocedural kinds of analysis still do block in the process. The extensive use of global resources adds a lot to the problem, however, we suppose that such architecture of a target program is a problem by itself and analyzer’s slowdown is only a side-effect.

5. Discussion

The resulting architecture is almost linearly scalable for some kinds of analysis and most of the applications not involving the intensive data exchange. The ephemerality of the word “almost” is highly questionable. A significant footprint may be noticed if unstructured sources are provided. This issue is negligible due to the way safety-critical software is developed – the best practices prohibit relying on global variables, and, with the help of internal command for cache prewarming, such projects might cause only a negligible slowdown. The interprocedural analysis suggests the use of dependency analysis scheduling, which is a part of additional research not covered by the paper.

The architecture is slower than other tested designs in a sequential mode: programs in intermediate codes are not generated for free. Still, it makes the analysis language-agnostic, indirectly improving code quality by reusing all of the verification sequences for all languages instead of one, so we find the usage of the approach fairly judged. We would still want to further examine interprocedural analysis in the next research.

6. Related work

Most studies focus on analyzing distributed systems rather than developing a static analyzer.

The main pattern used in the paper is based upon MapReduce [2], the well-known distributed model allowing to reduce the problem to multiple chunks in a failsafe manner and then glue up results together. We don’t focus on technical details but rather describe the measures we had to take in order to perform such an analysis.

In the study [3] the authors adopt the actor model to a whole program analysis, particularly testing it on call graph analysis. Our approach is also aiming at providing a framework rather than solving a single issue. The DiViNe model checker [4] provides an efficient way to distribute analysis to multiple nodes using MPI, however, unlike our research, it is aiming at using LLVM [5] operational codes rather than own development – not without a reason as it helps quickly get to support new

languages supported by LLVM-based compilers. Our language can heavily exploit the database behind in order to preserve memory and improve the ease of distributing between nodes, and we see it an advantage over LLVM codes. The research [6] modifies a static analyzer for scalability via OpenMPI. It is based upon LLVM IR, and results demonstrated by authors show its efficiency only on large projects. Unlike that project, the analyzer of ours is built with HPC in mind, although it was not a primary goal at the current stage.

Intermediate code representations aren't uncommon in the static analysis field. The most widespread intermediate representation is LLVM [5] which is used in many compilers including Clang. Many analyzers use LLVM IR because it is easy to start with. However, our opinion (supported by authors of [6]) is that this kind of IR requires significant efforts to apply to static analysis and often is a reason for inefficiency. Saturn project [7] introduces Calypso – a generic-purpose logic programming language. Based on research reports, it was created due to similar reasons.

7. Conclusion and future work

The main result of the research is a developed model for distributing static analysis load to multiple nodes. The unique intermediate code representation language with a semantic database behind had been created with a purpose of simplifying the analysis stage. As source programs are translated to this language, they are sent directly to virtual machines for execution using the suggested greedy file distribution algorithm, making analysis stage language-agnostic in described terms. Hypervisor's role is in controlling and orchestrating of virtual machines, in turn, the latter can request non-local semantics from the hypervisor. The distributed object lookup algorithm had been designed to cope with large projects. Tests show that while presented model suggests a non-zero overhead due to translation to IR, the analysis significantly benefits from load distribution and a good semantics preservation, which results in improved performance with few nodes compared to a single node run (although among single node runs the model is definitely not the fastest) and better precision levels.

The overall evaluation had proven that the model is quite flexible, so the future work will be concentrated on bringing more analysis types onboard rather than changing the model. Certain aspects like source distribution and semantics reducing can be further improved by employing more HPC techniques.

References

[1] Menshchikov M.A., Lepikhin T.A. Applying MapReduce to static analysis //Proceedings of XLVII International Scientific Conference on Control Processes and Stability (CPS'17). V. 4 (20), P. 433-444.

[2] Dean J., Ghemawat S. MapReduce: simplified data processing on large clusters // Communications of the ACM. 2008. V. 51. No. 1. P. 107-113.

[3] Garbervetsky D., Zoppi E., Livshits B. Toward full elasticity in distributed static analysis: the case of callgraph analysis // Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM, 2017. P. 442-453.

[4] Barnat J. et al. Divine: Parallel distributed model checker // Parallel and Distributed Methods in Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop on. IEEE, 2010. P. 4-7.

[5] Lattner C., Adve V. LLVM: A compilation framework for lifelong program analysis & transformation // Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization. IEEE Computer Society, 2004. 75 p.

[6] Abdullin A., Stepanov D., Akhin M. Distributed Analysis of the BMC Kind: Making It Fit the Tornado Supercomputer // International Conference on Tools and Methods for Program Analysis. Springer, Cham, 2017. P. 1-10.

[7] Aiken A. et al. An overview of the Saturn project // Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. ACM, 2007. P. 43-48.