Shibo Xing

Dr. Xulong Tang

CS2210

April 10th 2020

*Review for Quantitative Overhead Analysis for Python*

The paper identified several categories of python’s overheads as an interpretive/dynamic language under the comparison of static languages like C. Due to the inferior run-time performance of the dynamic language, most companies would have to rewrite their apps or models in static language for more robust performance once they complete a prototype written in a dynamic language. The benefits of the rich features from the dynamic languages often come with run-time overheads. And thus, analyzing python Just-In-Time (JIT) interpreter’s overheads is crucial to the development of dynamic languages of its type.

The paper attributes the Python’s slow performance to three reasons which are additional language features, dynamic language features and JIT interpreter overheads. As in the first category, Python is based on C but has a set of bound checking that doesn’t exists in C. It also has much richer control flow features. In terms of the dynamic language features, python has to do all type checking in run time. Because of dynamic typing, Python allows re-declaration of the variables types, which leads to a more complicated name resolution. The paper also deems the third category of the overheads, which consist of JIT dispatches of bytecode interpretation and function calls to C, as a main contributor of Python’s slow performance.

For breaking down the Python overheads ratios, the paper has taken a approaches at different system levels which range from tallying the number of CPU cycles at the assembly level to measuring execution time of different types of Python and JIT operations. The discovery of function calls to C as a part of the overhead is a big part of this paper. They realize the operations related to stack frames take up the majority of the C function call overheads. This discovery is very critical since the new V8 engine for JavaScript has similar issues and might be an issue for other dynamic languages.

The paper also analyzes measures to reduce the run-time of JIT, of which the garbage collection run-time seems to be a major part. In the generational garbage collection model, the sizing of the nursery (youngest generation variables) with a purpose of decreasing cache misses result in reduction of JIT run-time in most cases if not all. Not only the nursery-cache ratio, the absolute size of the cache is also an major factor of JIT performance. They mention 3% to 14% of run-time could be improved by this optimization on garbage collection. Also proven in this section is that V8 engine shares these properties.

The biggest discovery of this paper is that C function calls contribute to a main part of dynamic languages’ overhead. The authors mention that this result is found only after they have taken some optimization measures such as using inlining to reduce Python function setup time and caching variable lookups for reducing the name solution time. My thought regarding this paper is that since the authors have identified how specific optimization measures have reduce their corresponding overheads, they could also further explain how the application-based nursery or cache sizing can help reduce the C function call overheads. The authors seem to skim over the connection between their optimization and C function call overheads.