Review: A Brief History of Just-In-Time

Today we are going to review a paper by John Aycock written about the Just-In-Time techniques used since the 1960s. Before jumping into the details underneath, there are some concepts I would like to go through:

- Translation: programming languages usually require translations to be executable. Usually, there are two ways to do so: compilation and interpretation.
- Compilation: involves at least two languages. One is source language, usually at a higher level, and the target language. Codes won't execute during compilation. After compilation, the original codes get closer to native/lower level.
- Interpretation: instead of converting all codes entirely at once, an interpreter reads the code block by block and executes corresponded actions.

There are advantages and disadvantages on both of these techniques. For example, compiled code runs faster, especially when compiled to be directly executable by the hardware underneath, and as compilation time does not count toward the runtime performance, lots of optimization and analysis could be done at this stage. However, after compilation, as programs are getting closer toward native code, codes become more and more platform specific and their size grows larger and larger. Interpretation, on the other hand, is small to carry on, since higher level languages contain more information using the same amount of space, compared with native code. Also, interpreted programs are more portable as the program itself mostly does not contain platform specific sections. In order to take both of these approaches' advantages, just-in-time(JIT) techniques kick in.

As we talk more about JIT, I would like to be more specific about the definition: fitting existing code into templates and self-modified code are out of discussions since there is no actual translation happening in the process.

The first sign of JIT was from the University of Michigan Executive System for the IBM 7090 around 1975. At that time people discussed the possibility that assembler and loader could do translation during program execution. Later on, Mitchell found that compiled code could be extracted from the interpreter at runtime by storing the action done during execution and used later. However, if the codes haven't been executed before, interpreters will be re-invoked. This may potentially lead to pause as unexecuted codes will only be executed after interpretation.

JIT is a tradeoff between execution efficiency and space. Based on the data from Knuth's study in 1971, most of the programs are executing a small portion of their codes most of the time. To make use of this discovery, two approaches appeared: mixed code and throw-away compiling.

- Mixed Code: the program consists of native code and interpreted code, where the native code parts are frequently used while interpreted codes are seldom to be executed.
- Throw-away compiling: unlike static compiled, some codes will be compiled dynamically when needed, and if the machine is under memory pressure, compiled code will be abandoned to free up space. However, once the machine has more space then code might be compiled again. Example here is BASIC

Notice that for mixed code mode, using pre-compiled libraries within interpreted programs doesn't count: as it does not involve the translation process of the program itself. Mixed code also has other huge disadvantages here: maintainer now needs to keep identical behavior across the compiler and the interpreter. Throw-away compiling, however, is basically targeting saving space.

Also, there is an extension over the mixed mode: like Java, which has a series of intermediate byte code, or say virtual machine instructions, and the program will be entirely compiled into this state before execution.

Here also comes the topic of "what kind of code should we compile?" Fortran language maintained a "frequency-of-execution" counter, and then tried compiling and optimizing these "hot" code blocks. Self language then found that the object-oriented style usually brought short methods and it is better to optimize method callers by inlining these frequently called short methods. Franz in his doctoral work introduced a concept "slim binary", which contains a series of machine-independent representations of program modules. Codes are generated when a module is loaded. Even though this approach was not faster than traditional ones, it was believed that as the performance of computers increased, this method will win eventually. Scheme introduced more sophisticated metrics e.g. use code profiles to reorder code blocks and improve hardware prediction in the background, so called "continuous optimization". Erlang, however, allowed users to explicitly annotate codes that needed to call JIT compiler. OCaml brought "opcodes", which injected translated codes directly into existing threads to reduce the overhead of dispatching instructions, and left rooms for future optimization.

We may also find that simulation is doing similar tasks to translation. Simulators are trying to run a binary that is specifically targeted and optimized for a certain platform on other platforms. Line by line translation may be fine at first, but better performance is needed to make themselves useful. So, except the technique we have mentioned above, simulators have added a new technique: "hot path detection". Some structures, like loops, may be more likely to be executed multiple times thus could be translated. Also, if hardware supported profiling codes, hot paths could be detected and translated on the fly. Other considerations, for instance, are 1) complicated codes may be excluded from translation to save time and space for more generic cases 2) bail-out mechanism when some code's performance is worse after translation.

A lot of effort was invested at the time Java was introduced, and people found that the performance of JVM was astonishingly bad. Cramer, the founder of Java, observed that 68% of the time was spent on interpreting the code, and this is also the upper bound of JIT optimization in JVM. During this progress, instead of totally relying on improving JVM's performance, it is found that optimizing Java code was also important. However, the traditional compilation optimization was expensive if applied to Java, so later researchers were looking for a balance between the efficiency of the compile algorithm and the speed of java bytecode execution. A different idea may be annotating bytecodes before runtime for more efficient JIT process (targeting native code, etc.), and running a separate thread, preferably on a different CPU core, for "continuous compilation".

Review: A Brief History of Just-In-Time

To sum up, a JIT system can be classified using these characteristics from the author's point of view: 1)invocation: either JIT is called by users or not. 2)executability: either JIT is targeted specifically for a single pair of source/target language or not, if multiple targets supported, JIT could choose which language to go with. 3)concurrency: the compilation runs in a different thread or CPU core, and will not block the normal program runs. The trends are implicitly executed JIT, and executability varies among different systems, but not really just a JIT system issue: it requires more collaboration between different systems. Concurrency was at the beginning state back in 2000.

I was amazed after finishing the paper, knowing the work of JIT had begun in the 1900s. The ideas people have for developing various dynamic languages and optimization done under limited information context are also interesting to learn. I am also motivated to learn how JIT evolved in the modern era, and it turns out this is still a popular topic today. PHP, a scripting language used by facebook, implemented JIT optimization at 8.0 and got rid of the reputation of poor performance as in version 5.0. Android, a widely used smartphone operating system, optimized their application VM dalvik by implementing JIT. Although our hardware performance has increased from time to time, JIT is still playing an important role, releasing the potential power our machine owned. Learning the history of JIT helps a lot when trying to understand the idea behind modern JIT systems.