VMKit - Adaptive Optimization System for LLVM JIT Compiler

Gayana Ranganatha Chandrasekara Pilana Withanage,Kuganesan Srijeyanthan, David Daharewa Gureya
{gayana.withanage, kuganesan.srijeyanthan, david.gureya } @tecnico.ulisboa.pt
Group 2- Virtual Execution Environments: 79529, 79531, 79533
Instituto Superior Técnico, Lisbon, Portugal.
March 2014

Abstract

This article is presented to propose an improvement for an existing problem regarding the performance of LLVM JIT (Just in Time) compiler, which is an inherent slowness during the start-up of virtual machines (VM), Ex: VMKit [1]. Our proposed solution is to use a Trace based JIT compiler as an adaptive optimization for the current LLVM JIT. Moreover our implementation will have a mixed mode execution including a byte-code interpreter and LLVM JIT together. In the end, we will benchmark our implementation so that we can compare the difference in performance of the current LLVM JIT and the improved LLVM JIT with VMKit framework.
Key words*: VMKit, LLVM, JIT, VM*

1. Overview

Implementing a virtual machine (VM) is a painful task which demands huge effort and knowledge regarding the hosting infrastructure and architecture. When moving from one architecture to the other it is again a time consuming task which demands reimplementation of the main modules of VM. To relax this inherent overhead of building a VM, VMKit which work as a substrate has been developed so that it provides about 95 percent of the code required in developing a new VM. VMKit provides basic components required to create a VM such as JIT (Just In Time) compiler, Garbage Collector (GC) and a Thread Manager.
J3 and N3 are VMs that have been developed using VMKit which prove the significant reduction in development time of a completely new VM [1]. Furthermore, the core of VMKit depends on LLVM compiler infrastructure which provides the required key components such as JIT compiler and GC. In this article our point of interest is the LLVM JIT compiler because it is a known fact that the startup time consumption during the VM execution, developed using this core module is considerably higher. Therefore we propose an improvement for the behavior of LLVM JIT compiler, in order to make the startup and the run time to be faster than the current implementation.

2. Virtualization technology studied

- ***Name***: VMKit
- ***URL***: http://vmkit.llvm.org/
- ***VM-type***: High Level Language VM
- ***Common usage***: VMKit eases the development of new managed runtime environments (MREs) and the process of experimenting with new mechanisms inside MREs. J3(JVM) and N3(CLI) are built on top of VMKit.
- ***Motivation***: While VMKit aims to provide a fully functional intermediate layer for VM, its still under heavy development and misses some vital components with some open projects proposed. One of the current drawbacks of the LLVM JIT is the lack of an adaptive compilation system which will be our contribution to this project. What is missing is a system that can keep track of and dynamically look-up the hotness of methods and re-compile with more expensive optimizations as the methods are executed over and over. This should improve program startup time and execution time and will bring great benefits to overall performance.

3. Internal Mechanisms Study -JIT compilation with LLVM

The LLVM Project is a collection of modular and reusable compiler and tool chain technologies. All the libraries including VMKit have been implemented in C++. As VMKit is a substrate for VMs, the components integrated in order to develop VMKit should have a general purpose interfaces and functionalities. Therefore the compiler should have support for a general purpose instruction set to allow VMs to implement intermediate representations

of arbitrary functions. LLVM JIT is exactly a suitable compiler for this task because of its inherent behavior without imposing object model or type system.
Once the VM generates the intermediate representation with the help of interfaces provided by LLVM, that VM delegates the compilation to LLVM JIT to generate the native code. LLVM JIT optimizes all methods to the same degree without considering the frequency of the number of time each method get called. This behavior is called the compilation without adaptive optimization. As a result, the initialization time get higher and the runtime also shows some slowness. Our proposed method would be an extension for LLVM Execution engine. once JIT compiler is triggered, it reads the byte code and creates Intermediate Representation code (LLVM IR). All the generated LLVM IR will then be optimized by the LLVM optimizer by removing dead code, constant propagation etc. Finally LLVM IR will produce machine code optimized for a specific CPU.

4. Project Approaches

We have presented trace-based JIT compilation as an adaptive optimization method for LLVM which is the technique used by virtual machines to optimize the execution of a program at runtime. Our implementation uses a hybrid mode of execution approach: the VMKit starts running in a fast-starting byte-code interpreter. As the program runs, the VMKit identifies hot (frequently executed) byte-code sequences, records them, and compiles them to fast native code. This method will be called as sequence of instruction, a trace. We have made this design decision which is based on the assumptions that program are spend most of their time in hot loops. Further, trace-based JIT can greatly speed up programs if they spend most of their time in loops where they take similar code paths.

5. Detail Sections

Loops are not compiled the first time they are called. VMKit maintains a loop count,which is incremented every time the loop executed. VM interprets a loop until its call count exceeds a predefined trace_JIT threshold value. So, most frequent loops(hot) are compiled as soon as VM started, loops which are executing less than threshold value will not be compiled by JIT. The threshold value could be selected to obtain significant balance between start-up times and run-time performance.
After particular loop is compiled, its call count will be reset to zero. Subsequent calls to the loop continue to increment its count. When the call count reaches its trace_JIT threshold again it will be compiled by applying more optimization methods. This scenario will be occurred on some methods during an aggressive call by VM. So, high frequent loops will be more optimized than the normal less frequent routine.

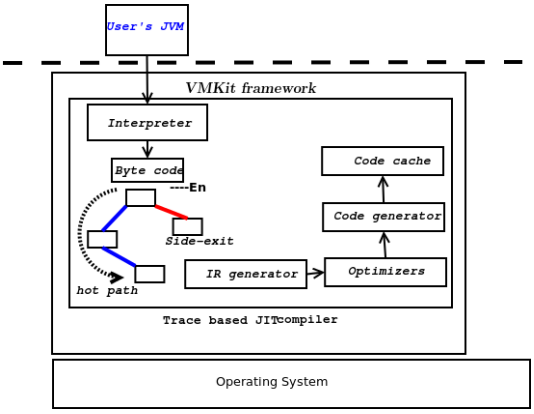

Figure 1. Trace based JIT for VMKit

Fig 1 explains how trace based JIT will lay down in VMKit. Even though VMKit composed with other components , diagram shows only JIT components which has derived from LLVM. As number of loop execution is counting by JIT when the counter reaches a threshold, the interpreter will be starting trace process. According to above diagram tracing process will be started at place -En and when tracing process reaches back to the address where the tracing started, the interpreter stops tracing and the compiler compiles the trace to native code. The path in blue may or may not be an actual hot path since interpreter is not doing an perfect path profiling. Guard control will be applied when instructions are deviated from actual hot path. During the interpretation , if a Guard is triggered , the native code use side-exit and back to normal interpretation mechanism. There are some other advanced way to deal with frequent side-exit which would make side-exit path to separate branch of hot-path [2].

6. Conclusion

VMKit provides a substrate for the development of VMs within a short period of time but still it has left interesting research areas in optimizing the execution of the VMs. Therefore our proposed solution for the LLVM JIT improvement would be an exciting research area to identify and exercise Virtualization concepts in practice. At the checkpoint level, we hope to have a thorough understanding of the current system architecture and a prototype version of our implementation which will provide a better insight of the system behaviour. Finally, at the successful completion of this project we are hoping to contribute our research efforts to the VMKit community

References

[1] VMKit: a Substrate for Managed Runtime Environments. http://llvm.org/pubs/2010-03-VEE-VMKit.pdf.

[2] Michael Bebenita, Mason Chang, Karthik Manivannan, Gregor Wagner, Marcelo Cintra, Bernd Mathiske, Andreas Gal, Christian Wimmer, and Michael Franz. Trace based compilation in interpreter-less execution environments. *In Principles and Practice of Programming In Java 2010 (PPPJ)*, 2010.