A CONCEPT PAPER FOR A BETTER JIT TRANSLATION

GROUP 120

April 18, 2017

Contents

0.1 INTRODUCTION

The advances in software and information technology today put great demands on the hardware and system software of devices. Devices like smart phones have larger capabilities than an average computer in the y2k era [1]. This has lead platform and system engineers to push the boundaries when it comes to increasing the performance of these systems. This process has led them to revisit some of the dormant ideas in computer science and try to apply them from a newer perspective and in a different computing landscape. One such idea is just-in-time (JIT) compilation. JIT compilers translate byte codes during run time rather than prior to execution, to the native hardware instruction set of the target machine [2]. JITs are also known as dynamic translation.

0.2 BACKGROUND

Though this idea was first proposed in the early seventies, it has seen a renaissance with interpreted languages like Java and dynamic languages like JavaScript and Python being adopted for large scale applications. Just-in-time compilation attempts to bridge the gap between the two approaches to program translation, compilation (ahead of time compilation (AOT)) and interpretation [3]. Generally, compiled programs run faster as they are translated to machine code. However, they occupy a larger memory footprint as the compiled machine code is typically larger than the high level program implementation and also take a longer time to optimize the code. Interpreted code on the other hand takes up a smaller memory footprint as it is represented at a higher level and hence can carry more semantic information making it more portable. However, they need access to the runtime of the system as they need to gather much more information during the runtime to successfully execute the programs [4]. JIT compilation combines the speed of compiled code with the flexibility of interpretation, with the overhead of an interpreter and the additional overhead of compiling (not just interpreting). JIT compilation is a form of dynamic compilation, and allows adaptive optimization such as dynamic recompilation, thus in theory JIT compilation can yield faster execution than static compilation. Interpretation and JIT compilation are particularly suited for dynamic programming languages [5], as the runtime system can handle late-bound data types and enforce security guarantees. The existing JIT for is very simple-minded, and does little more than translate each byte code into the corresponding machine code. Either improve the translation by using one of the many JIT libraries now available, or adjust the Oberon compiler and the specification of the byte code machine to free it of restrictive assumptions and produce a better pure-JIT implementation.

0.3 PROBLEM STATEMENT

The current JIT compilation causes a slight delay to a noticeable delay in initial execution of an application, due to the time taken to load and compile the byte code. Sometimes this delay is called "startup time delay". In general, the more optimization this JIT performs, the better the code it will generate, but the initial delay will also increase. A JIT compiler therefore has to make a trade-off between the compilation time and the quality of the code it hopes to generate. However, it seems that much of the startup time is sometimes due to IO-bound operations rather than JIT compilation.

0.4 OBJECTIVES

Come up with information for a better JIT Translation that can help in;

- Compiling byte code (not high level code)
- Perform AOT optimizations faster
- Perform runtime optimizations
- Executing machine code is faster than interpreting byte code

- Combine speed of compiled code w/ flexibility of interpretation

0.5 SCOPE

The concept will limit its focus on smartphones tablets android applications and its application in JavaScript in relation to TraceMonkey for Firefox.

0.6 ANTICIPATED OUTCOMES

A better version of JIT compilation should be able to be used in some programs, or for certain capacities, particularly dynamic capacities such as regular expressions for example, a text editor may compile a regular expression provided at runtime to machine code to allow faster matching and this cannot be done ahead of time, as the pattern is only provided at runtime. Several modern run time environments rely on JIT compilation for high-speed code execution, including most implementations of Java together with Microsofts NET Framework. Similarly, many regular expression libraries feature JIT compilation of regular expressions, either to byte code or to machine code.

One possible optimization, used by Sun's Hotspot Java Virtual Machine, is to combine interpretation and JIT compilation. The application code is initially interpreted, but the JVM monitors which sequences of byte code are frequently executed and translates them to machine code for direct execution on the hardware. For byte code which is executed only a few times, this saves the compilation time and reduces the initial latency; for frequently executed byte code, JIT compilation is used to run at high speed, after an initial phase of slow interpretation. Additionally, since a program spends most time executing a minority of its code, the reduced compilation time is significant. Finally, during the initial code interpretation, execution statistics can be collected before compilation, which helps to perform better optimization[3].

A common implementation of JIT compilation is to also first have AOT compilation to byte code (virtual machine code), known as byte code compilation, and then have JIT compilation to machine code (dynamic compilation), rather than interpretation of the byte code. This improves the runtime performance compared to interpretation, at the cost of lag due to compilation. JIT compilers translate continuously, as with interpreters, but caching of compiled code minimizes lag on future execution of the same code during a given run. Since only part of the program is compiled, there is significantly less lag than if the entire program were compiled prior to execution. [3]

0.7 References

[1] $SnapdragonS4, S3, S2, S1ProcessorSpecsandDetails|Qualcomm, Qualcomm, 2017.[Online].Available$
$http://bit.ly/TIDHMK.[Accessed: 16-Apr-2017]$.

[2] Aho, A., Lam, M., Sethi, R., and Ullman,6 J. Compilers: principles, techniques, and tools, vol. 1009. Pearson/Addison Wesley, 2007.

[3]$Just-in-timecompilation, En.wikipedia.org, 2017.[Online].Available: https://en.wikipedia.org/wiki/Just-in-time_compilation.[Accessed: 16-Apr-2017]$.

[4] $2017.[Online].Available: http://www.cs.columbia.edu/\ aho/cs6998/reports/12-12-17_Ramanan_JIT.pd.[Accessed: 16-Apr-2017]$.

[5]$2017.[Online].Available: http://www.cs.columbia.edu/\ aho/cs6998/Lectures/14-09-22_Croce_JIT.pd.[Accessed: 16-Apr-2017]$.