JIT through the ages

Evolution of just-in-time compilation from theoretical performance improvements to smartphone runtime and browser optimizations

Neeraja Ramanan

Abstract

This paper is a study on just-in-time compilation and traces its evolution from being a theoretical performance optimization to a technology that provides concrete speed-ups for constrained applications and in dynamic programming languages. Also highlighted are the increase in sophistication of the techniques used to deal with the complexities that arise in these problem domains and the inherent trade-offs.

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 [16][18]. However, this increase in capability has also lead to a mind-boggling spectrum of applications that these devices find. Further, with the tremendous increase in the number of developers, the quality of software being written is also varied.

In such a scenario, platform and system engineers are pushing the boundaries when it comes to increasing the performance of these systems. This process has lead 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 to the native hardware instruction set of the target machine [1]. 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. This paper studies this trend and discusses the evolution of this concept for modern day computing.

The rest of this paper is organized as follows. Section 2 gives a brief overview about just-in-time compilation and talks about the trade-offs involved, while Section 3 describes some of the earlier implementations of JIT compilation. Section 4 describes a generic JIT Compiler, based on the Java run-time environment. Section 5 describes the Android JIT, while providing some background on the Android application framework and also describes some similar JIT implementations for embedded computers. Section 6 describes a just-in-time compilers for dynamic languages like JavaScript and Python and discusses the features that make them applicable for today's internet systems.

2 JIT compilation

Just-in-time compilation attempts to bridge the gap between the two approaches to program translation: compilation and interpretation. 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. Further, they 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. Thus, it's more portable. However, interpreters need access to the runtime of the system as they need to gather much more information during the runtime to successfully execute the programs. This is why interpreted programs take longer to run and have a more complex runtime.

2.1 Overview

In the just-in-time compilation process, starting with the interpreter, some features of a static compiler are built into the system. Typically, a JIT compiler will isolate some sections of the code at run-time which are accessed more often and then compiles them to native code, aggressively optimizing those sections in the process. The sections of code that are to be statically compiled can be identified in many ways, and

this is briefly described in Section 3. These sections of code are commonly called *hot-paths*.

2.2 Hot-path detection

As described above, hot-paths are the most commonly accessed sections of code. Hot-paths could be identified at various granularities. Here, granularity means the level of detection of each commonly access code part. This could mean at the method level or at the level of a string of instructions or even at the individual instruction level. Most JIT compilers are written for statically typed languages like Java, working on servers and desktop. Thus, they performed method-based identification, where the hot-paths are identified at the method level granularity. While this technique works well in simplifying the design of the JIT and provides high levels of speed-up, studies have shown that there's an additional overhead in terms of memory and power consumption [6]. This is because at this coarse granularity (coarse, compared to a string of instructions), there a different sections of the code which are compiled even though they are not hot sections. This includes exceptions and other such code. To avoid this, a more complex method of identifying hot-paths, known as trace-based just in time compilation has been proposed. Here hot-paths, which are also called traces, are selected based on various criteria. Generally, the trace head or start of a trace is selected as the beginning of loop or a jump statement. A trace-based JIT is explained in detail in Section 4.

2.3 A few considerations

In general, the design of a JIT system involves many trade-offs. The most major consideration to take into account is the fact that having both a compiler and an interpreter at run-time could prove expensive. This was the reason this idea did not catch on in the early stages [4]. Further, constant switching between interpreted and compiled code could prove for interrupted execution. Thus, a JIT compiler writer must take care to ensure seamless transitions between the two modes of execution. Different systems do this differently. A few JIT compilers manage this by having an additional data structure called the translation cache that provides for quick reference to the compiled code.

3 Chronology

In the purest sense, one of the first theoretical ideas for a JIT compiler can be dated back to McCarthy's 1960 Lisp paper [17] where he talks about compiling the source to machine code. However, implementations of the idea surfaced about a decade later for various languages like FORTRAN, Smalltalk, Self, Erlang, O'Caml and even ML and the ubiquitous C [4].

Some of the early ideas for JIT compilation can be summarized in terms of mixed-code and throw-away code compilation. Another interesting paradigm shift is to view JIT compilers in terms of simulators.

3.1 Mixed code and throw-away code compilation

Work by Dawson [9] and Dakin and Poole [8] are the earliest papers that talk about just-in-time compilation as we know it. Published in 1973 in the same journal, both papers talk about how performance of interpreted code can be improved by compiling it down to machine code. In the mixed code approach in [8], Dakin and Poole propose that in order to achieve the right balance between the poor space utilization of direct compilation and the slower running times of interpreted code, a common approach with data structure that keeps track of procedure calls in both cases must be used. Similarly, in [9], Dawson notes the three classes of instructions: very rarely used, occasionally used and often used, and addresses how we would select which of these instructions should be compiled. He states that the cost of compilation is relatively smaller than that of storing the compiled code and thus, once the memory for storing the compiled code reaches its limit, it can be flushed for the next compiled section of the code.

3.2 Simulation and binary translation

Four generations of early simulators were identified. The first generation consisted of plain interpreters, while the second generation dynamically translated the source instructions into target instructions one at a time. The third generation translated entire blocks of the source code dynamically while the fourth generation improved on the third by isolating a few key paths and translating them. Third generation simulators are similar in principle to method-based JITs while the fourth generation is similar to trace based JITs as discussed in section 4.3. The main considerations of both fourth generation simulators as well as trace-based JITs include, profiling execution, detecting hot-paths, generating and optimizing code and incorporating exit mechanisms. These features are explained in detail for trace-based JITs in the sections below

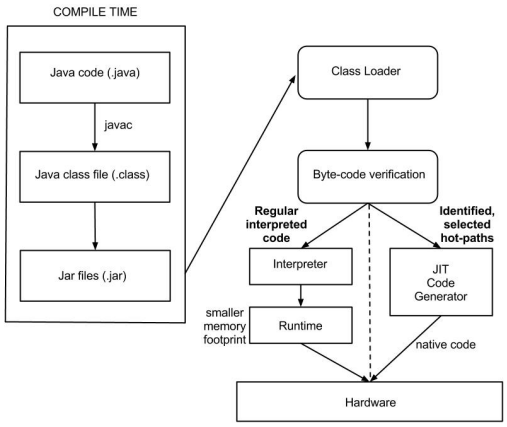

Figure 1: Generic Java-based JIT Compiler

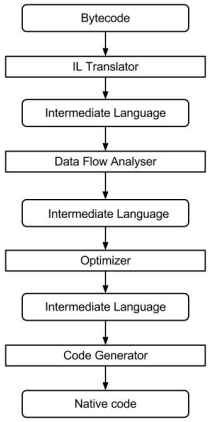

Figure 2: JIT Compiler Internals

4 A Generic Trace-based JIT Compiler

This section explains the working of a simple just-in-time compiler. For the purpose of this section and most of the rest of the paper, unless stated, we would be talking about a Java-based JIT compiler. The choice to work with a Java JIT is because Java is one of the interpreted languages that benefits significantly by the use of a just-in-time compiler. Further, as Java is one of the most widely used languages that is being used today [20], studies in optimization of the Java runtime has greater relevance to speeding up most commonly available systems.

4.1 Run-time environment with a JIT compiler

Figure 1 above provides a general schematic of a Java-based just-in-time compiler. The major components include the Java compiler and the class loader as well as the byte-code verifier in addition to the interpreter and a native code generator. The .java files are first compiled down to .class files and then they are bundled as jar files. This is all done at compile time.

At runtime, the class loader first loads all these class files on to the Java virtual machine. Then the byte-code verifier performs type checking to ensure that typing information is maintained constantly. In addition to these steps, depending on the exact implementation of the JIT, in some earlier stage, the potential hot-paths are detected and marked accordingly. Then, during runtime, the system keeps track of the number of times these potential hot-paths are run. When execution count of that particular hot-path hits a particular threshold, it is then compiled to native code. JIT compilers also differ in the manner in which the compiled binary is maintained. Most times, they are stored in a cache-like data structure that is designed to store system configurations.

4.2 Inside a JIT compiler

Figure 2 depicts the internals for a general just-in-time compiler block that is seen as a black-box named the JIT code generator, in Figure 1. As we can see from the figure, the code is aggressively optimized and checked multiple times to ensure that there is no change in the semantics with respect to the original interpreted code. Starting with the byte code, it is first translated to a carefully chosen intermediate language. The intermediate representation must have the properties that allow it to be translated to a tree-like structure that allows for better optimizations [2][12][14]. Then data and/or control flow analysis is performed on this IR to ensure that there is consistency between the compiled version and the original code. The most common representation that is used is the static single assignment form [7]. The SSA form is most amenable to optimizations like constant propagation, code motion and elimination of partial redundancies. After all the optimizations are performed, the code is then translated out of the SSA form and back to the original intermediate representation. This is then fed into a code generator to produce machine code.

The data structures, optimizations and maintenance of the hot-paths depends on each individual implementation of the JIT and how closely the system is integrated. We will discuss this in slightly more detail for a few systems in the later sections. However, most of the JIT compilers follow the schematic described above.

4.3 Trace selection and compilation

The optimizations that can be done on a JIT are greatly dependent on the granularity of the hot-path detection. The finer the granularity of the hot-path, greater the optimizations can be done. However this increases the cost of transitioning between the compiled and interpreted modes of execution and necessitates the two units to be integrated more tightly.

Commonly, hot-path detection is done at either the method level or at the trace (or a string of instructions which start with a loop head) level. In method based JITs, as the name suggests, the potential hot-paths are marked at the beginning of each method implementation. However, what is most prevalent and effective is the trace-based JIT compiler, which compiles the sections of the code that are most likely to be called often. These could include certain obvious choices like targets of backward branches [14]. The trace is ended when it forms a cycle in the buffer, executes another backward branch, calls a native method or throws an exception [15]. These potential traces are profiled with some additional meta-data to keep track of their execution count.

At each stage when the potential trace head is reached, the counter is incremented. When this count reaches a predefined threshold value, the trace is then compiled as described in the previous subsection. This compiled trace is then stored in a translation cache like structure. Most modern JIT systems enable chaining of multiple traces for greater flexibility. This allows the execution to transfer a little less frequently between the JIT and interpreter.

The above section described the generic JIT compiler that is seen in most systems today. For the rest of this paper, we will talk about what special corner case optimizations are handled for systems like smartphones and fast web-browsers.

5 JIT on Smartphones and Tablets

Handheld devices today are changing the entire computing landscape. This phenomenon has made computing accessible to almost any one. The organizations that can take credit for making this possible include the hardware manufacturers and the open source developers. The hardware manufacturers can be credited for the System on Chip (SoC) design that makes it possible to have multiple components on a single piece of silicon that functions as a single unit. The open source developers create various applications for these devices. These applications provide

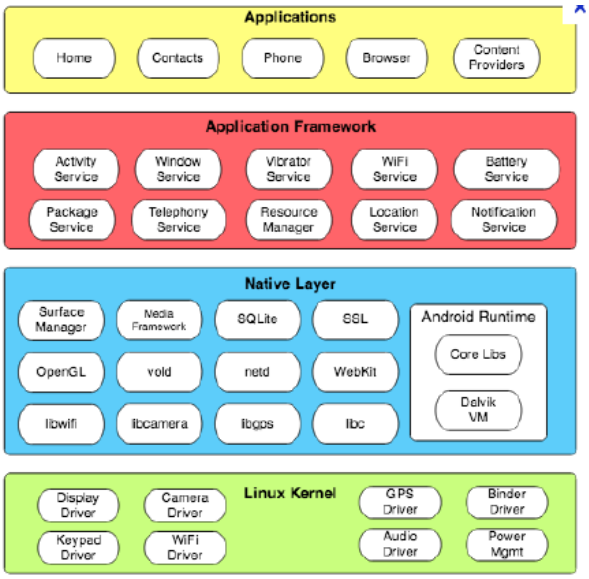

Figure 3: The Android Application Framework

information and services to the user on-the-go, making life more connected. Many open-source developers have applications for these devices that are available for free or for a small nominal amount. These include applications for productivity, news and information, entertainment and games.

While these developers work on frameworks on the runtime that are provided by software development kits (SDKs), the engineers themselves have implemented many optimizations both at the hardware level as well as at the operating system level. In particular, the optimizations at the operating system level are very complex and worth looking into. The three major operating systems include Apple's iOS that runs the iPhones, iPads family, Google's Android that runs various devices that are manufactured in accordance to the open-handset alliance, and the Windows 8 touchscreen OS by Microsoft. In this paper we will talk about the Android operating system in particular and how it uses the JIT compilation techniques to improve performance.

5.1 Android Application Framework and Dalvik

From Figure 3, we see that the Android operating system runs on the Linux kernel. While the kernel itself and its many features are abstracted away from the user, the individual applications are sandboxed on top of the Dalvik virtual machine. This VM was developed by Google instead of the regular Java VM. There were many reasons for this, the foremost being that Dalvik is more sensitive to the constraints that are imposed on embedded devices like smartphones,

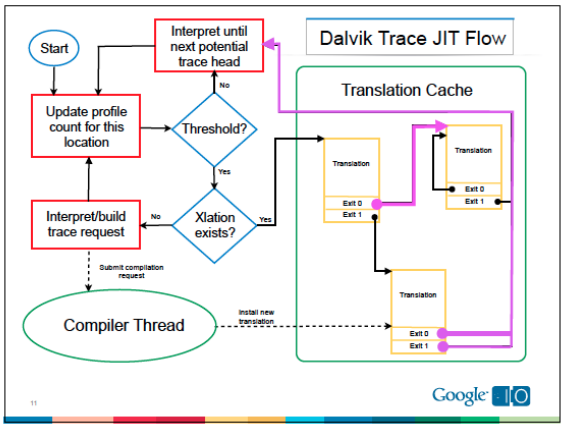

Figure 4: The Dalvik Trace JIT Flow

like lower frequency, smaller RAM sizes as well as battery power [10]. This, along with some core libraries that are abstracted from the kernel drivers, constitutes the main runtime of Android. The applications themselves, written in Java, are run on top of a framework which further abstracts some features and provides for appropriate package managers for each of the services. In this manner, the kernel and the lower layers are abstracted from the developer and the user, providing for both ease of use and development

The Dalvik VM is a very lightweight implementation as each of the different services that are run on an Android powered device are all run on an individual instance of the Dalvik VM. Moreover, the initial system server process, which includes the activity manager, libc and other such components, is also bundled and run on an instance of Dalvik. The component that provides for forking out an instance of Dalvik upon request is called the Zygote and it is crucial to the Android system itself.

Thus, it is clear that any optimizations made to Dalvik will help speed up the overall runtime. Further, many of the applications that are commonly run on tablets and smartphones are games. Most games are extremely compute intensive (not exclusively games but other applications as well), running the same sections of code repeatedly. Thus it is natural to see an implementation of the JIT compiler in Dalvik. The next subsection discusses the Dalvik JIT in more detail.

5.2 Dalvik JIT

The Dalvik JIT is similar to the generic JIT that was explained in Section 4. As the applications are written in Java and run on top of a virtual machine that replaces the JavaVM, the schematic is very similar to Figure 1. The only significant change is seen during compile time. Instead of storing each of the class files into a separate jar file, there is a tool called the dx tool which compiles multiple class files into a single dex file. The dex file format provides for about 5% improvement in storage as compared to the Jar file over uncompressed data. This is significant in terms of the memory savings.

The JIT itself is a generic trace-based JIT whose flow is depicted in Figure 4. The potential trace heads are identified in the front-end of the compiler at the parsing stage after the conversion to bytecode. The opcodes of the dex byte code instructions are checked. The front-end analyzes each method from a high level and marks out sections which may not be optimized and does other such inspections of the source code. When the traces are compiled, as in the generic trace-based JIT in Section 4, they are stored in the translation cache. This cache is maintained during run-time when the traces are compiled. There is provision to chain multiple traces, which decreases the bouncing between the compiler and interpreter. The translation cache is designed in such a way that it integrates the compiler and interpreter tightly, acting as a buffer between the two.

The trace is aggressively optimized before it is compiled. This is done by converting it into the SSA notation. Some of the optimizations included are dead store elimination, variable folding and inlining of getters and setters. On the whole, the Dalvik JIT by design itself is highly efficient and is optimized to be lightweight and take up very little overhead on a constrained system.

5.3 Similar work

Gal et al. proposed an embedded Java JIT compiler for resource constrained devices [14]. Similar to the Dalvik Trace JIT, this JIT compiler also optimized the traces in the SSA form. The HotpathJIT proposed to merge the trees when they were overlapping in terms of the traces that they compiled. Further they supported optimization of invariant code motion as well. The HotpathVM was a more complex system that was proposed rather than the Dalvik JIT which is a production quality compiler.

The above section concludes our discussion on the JIT compilers in smartphone/tablet or other handheld devices. We talked about how they helped to optimize the runtime and improve the speed and memory demands on them.

6 JIT on the Internet

With the increase in the data that is available on the Internet, there is a need to greatly increase the

efficiency and speed at which webpages and servers are processed and requested. Most server-side scripting languages and front-end design languages are dynamic which by nature makes them slower. This has lead to engineers coming up with innovative solutions to improve performance, while maintaining the ease of use for developers. For instance, Facebook has come up with its source translator that compiles PHP down to C++. Called the HipHop, this translator uses g++ to run the compiled C++ code [11].

In such a landscape, it is natural that the developers look towards JIT compilers to increase performance as some of the functions that are implemented while querying webpages or rendering them are highly regular. Some of the most interesting implementations of JITs in terms of working with webpages and servers include the TraceMonkey for JavaScript in the Firefox and the JIT in the PyPy implementation of the Python interpreter. These topics will be covered briefly in the following subsections.

6.1 JavaScript JITs and TraceMonkey for Firefox

There have been many implementations of JIT compilers for JavaScript, similar to the paper on trace-based JIT Type Specialization for Dynamic Languages by Gal et al. [13]. The most popular is the TraceMonkey JIT that is found in the Firefox web browsers of versions 3.5 onwards.

This paper proposed an inexpensive and efficient method for performing type specialization by means of generating trace trees. The interesting name of TraceMonkey was proposed because the flow of execution jumps across the tree depending on the program counter in the interpreter. The trace selection and optimization is similar to the JIT compiler discussed in Section 4. The major difference between TraceMonkey and the other implementations is that additional information has to be incorporated in the Trace trees, namely the type information. This was implemented in the SpiderMonkey JavaScript VM in the Firefox browser and the authors observed an approximate speed up of 10x overall.

6.2 Some other developments on JavaScript JITs

Many different implementations of JavaScript JITs are available currently, the most popular of them being the dynamic optimization system called Dynamo [5]. Some other examples of JavaScript engines that have the interpreters generate native code include Apple's SquirrelFish [3]and Google's V8 JavaScript engine.

6.3 PyPy

Server-side scripting is a big bottleneck when it comes to data analysis of the billions of bytes of data. For instance, most Facebook's data is estimated at 100 peta bytes stored in a single Hadoop store. Accessing this data must be done quickly and efficiently without disturbing the configuration of the system overall. They are mostly done using shell scripts or scripting languages like Perl, Python and Awk. Most scripting languages, like Python are dynamic, which while making them very easy to use, make them highly inefficient.

Thus, there have been many attempts to optimize Python and other such languages for speed-up, including implementing JIT compilers for them. One such implementation is PyPy which is an alternative implementation of Python 2.7.2 that is fast and memory efficient [19]. The JIT compiler in PyPy is not built-in as a separate module but is instead generated by a JIT compiler generator. The authors exploit the fact that their interpreter is written in a high-level language to include this module. They believe that this will prevent their JIT compiler from going out of sync with their interpreter at any stage. Overall, they achieved an average speed-up of around 5.5 times over the regular implementation of Python. As this is on-going work, it will be extremely interesting to see the speed-ups that they will be able to get after including additional optimizations to their interpreter.

This concludes our Section on how JIT compilation helps to improve performance of systems that are used to power the internet as we know it, both at the front-end rendering as well as the server-side scripting.

7 Conclusion

This paper was intended to be an introduction into some of the recent work in just-in-time compilation and how this is implemented in some of the technology that affects our everyday lives. With greater need for faster and more reliable computing, just-in-time compilation along with many other enduring ideas for optimization, has seen a renaissance nearly three decades after it was first proposed. It will be interesting to see how just-in-time compilation will adapt further, with some of the more challenging problems that are going to rise in computing

References

[1] Aho, A., Lam, M., Sethi, R., and Ullman,

J. *Compilers: principles, techniques, and tools*, vol. 1009. Pearson/Addison Wesley, 2007.

[2] Aho, A. V., Ganapathi, M., and Tjiang, S. W. K. Code generation using tree matching and dynamic programming. *ACM Trans. Program. Lang. Syst. 11*, 4 (Oct. 1989), 491–516.

[3] Apple Inc. Introducing squirrelfish extreme - http://bit.ly/UzRMu5.

[4] Aycock, J. A brief history of just-in-time. *ACM Comput. Surv. 35*, 2 (June 2003), 97–113.

[5] Bala, V., Duesterwald, E., and Banerjia, S. Dynamo: a transparent dynamic optimization system. *SIGPLAN Not. 46*, 4 (May 2011), 41–52.

[6] Buzbee, B., and Cheng, B. A jit compiler for android's dalvik vm - http://bit.ly/UxO02j.

[7] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13*, 4 (Oct. 1991), 451–490.

[8] Dakin, R., and Poole, P. A mixed code approach. *The Computer Journal 16*, 3 (1973), 219–222.

[9] Dawson, J. Combining interpretive code with machine code. *The Computer Journal 16*, 3 (1973), 216–219.

[10] Ehringer, D. The dalvik virtual machine architecture - http://bitly.com/ewB19V.

[11] Facebook. Hiphop for php - https://github.com/facebook/hiphop-php/wiki.

[12] Fraser, C. W., Henry, R. R., and Proebsting, T. A. Burg: fast optimal instruction selection and tree parsing. *SIGPLAN Not. 27*, 4 (Apr. 1992), 68–76.

[13] Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M. R., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E. W., Reitmaier, R., Bebenita, M., Chang, M., and Franz, M. Trace-based just-in-time type specialization for dynamic languages. *SIGPLAN Not. 44*, 6 (June 2009), 465–478.

[14] Gal, A., Probst, C. W., and Franz, M. Hotpathvm: an effective jit compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual execution environments* (New York, NY, USA, 2006), VEE '06, ACM, pp. 144–153.

[15] Inoue, H., Hayashizaki, H., Wu, P., and Nakatani, T. A trace-based java jit compiler retrofitted from a method-based compiler. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2011), CGO '11, IEEE Computer Society, pp. 246–256.

[16] Intel. Pentium-iii processor family - http://intel.ly/UfjeMD.

[17] McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM 3*, 4 (Apr. 1960), 184–195.

[18] Qualcomm Incorporated. Snapdragon processors - http://bit.ly/TIDHMK.

[19] Rigo et al. Pypy - http://pypy.org/.

[20] Tiobe Software. Tiobe programming community index for december 2012 - http://bit.ly/cuwptY.