A Hybrid Just-In-Time Compiler for Android

Comparing JIT Types and the Result of Cooperation

Guillermo A. Pérez
National Tsing Hua University
Hsinchu, Taiwan
gaperez64@
sslab.cs.nthu.edu.tw

Chung-Min Kao
National Tsing Hua University
Hsinchu, Taiwan
klozesk@
sslab.cs.nthu.edu.tw

Yeh-Ching Chung
National Tsing Hua University
Hsinchu, Taiwan
ychung@cs.nthu.edu.tw

Wei-Chung Hsu
National Chiao Tung University
Hsinchu, Taiwan
hsu@cs.nctu.edu.tw

ABSTRACT

The Dalvik virtual machine is the main application platform running on Google's Android operating system for mobile devices and tablets. It is a Java Virtual Machine running a basic trace-based JIT compiler, unlike web browser JavaScript engines that usually run a combination of both method and trace-based JIT types. We developed a method-based JIT compiler based on the Low Level Virtual Machine framework that delivers performance improvement comparable to that of an Ahead-Of-Time compiler. We compared our method-based JIT against Dalvik's own trace-based JIT using common benchmarks available in the Android Market. Our results show that our method-based JIT is better than a basic trace-based JIT, and that, by sharing profiling and compilation information among each other, a smart combination of both JIT techniques can achieve a great performance gain.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Interpreters, Code generation, Optimization, Parsing, Retargetable compilers*

General Terms

Performance, Design, Experimentation, Languages

Keywords

Method-Based, Trace-Based, JIT, Android, Dalvik VM

1. INTRODUCTION

Interpreted and Dynamic languages have been on the rise for several years now. These programming languages are not compiled and translated into native instructions like classical programming languages are. Instead, a program written using them is read line by line and executed by an intermediate engine or virtual machine. The process of fetching each instruction and the execution of it can be much slower than how a native program is usually ran. This problem has traditionally been dealt with by adding a Just-In-Time (JIT) Compiler to the virtual machine [25] [4]. A JIT Compiler is a compiler that takes a code sequence or code block and translates it into native code during run time in order to improve its execution time.

The Java language is an example of an interpreted language which runs on top of Oracle's Java^{TM}Virtual Machine (JVM). Java allows developers to write a program once and run it on any platform that implements a JVM [30]. Developers might choose to write computationally intensive code using a different programming language (C and assembler are the most common choices) and communicate with the JVM using Java's Native Interface (JNI) [28]. This approach, however, requires that the developer provide the target user with a library compiled for the user's hardware. The use of JNI to interface with the virtual machine is also a problem because of the overhead required to change from the interpreted to the native environment and back [20]. It is also possible to compile an application written in Java to a native binary file using Java to native compilers [9] [32] but the resulting binary and the libraries needed to execute it need to be compiled for the user's platform as well, making this and the previous approaches less portable than the JVM and Google's Dalvik virtual machine. Because of these reasons most Java applications are written using mostly Java and are therefore benefited by a VM's JIT compiler. As of Java 1.3, the standard HotSpot became the default Sun Java Virtual Machine [29]. The HotSpot virtual machine included a method-based (MB) JIT compiler to analyze the program's performance for hot spots (frequently executed instructions) and compile them.

Recently, trace-based (TB) JIT compilers have become more popular than it's MB counterpart because of its ability to define traces that range from one method to another and therefore extend the scope of what is being considered for compilation and optimization. TB JIT compilers can also take dynamically typed languages and compile them to optimized native code that handles the variables' types being observed during run time [10]. As is explained in section 4.1.1, Java is not actually a dynamically typed language and therefore such an optimization does not completely apply to the Dalvik VM. Nevertheless, short instruction sequence compilation can result in smaller memory footprint and still improve performance by selecting the few most executed instructions for JIT compilation. Because of these reasons, and also because of the speed at which a simple TB JIT can start delivering performance improvements, Google chose to implement a TB JIT compiler for their Android operating system for mobile devices [6]. However, with the advantage of method structures as basic block delimiters and considering that Android's TB JIT, currently does not have the ability to create traces containing code from more than one method, it is not really clear and no one has really studied if a MB JIT would have the same effect or achieve even better performance.

Both JIT compiler types have their own strengths and outperform the other in terms of memory usage or delivered code quality depending on the situation and the platform's resources. Simple TB JIT compilers, such as Dalvik's, might seem better suited for resource-constrained devices such as cellphones because of their ability to avoid compiling cold code and their fast compilation times. However, more robust TB JIT compilers or MB JIT compilers, although slower and even though they might have a bigger memory footprint, deliver faster, more efficient, code. This paper compares the performance of the code generated by a limited TB JIT compiler and a MB JIT compiler running on a mobile device and suggests the usage of a fast, incomplex JIT compiler combined with a slow JIT compiler capable of delivering better optimized code.

In this work we propose a MB JIT compiler based on the Low Level Virtual Machine (LLVM) [22] compiler framework and API to deal with methods that can be further optimized by considering them as a whole block instead of focusing merely on the hot traces they contain, like the current TB compiler does. Our resulting JIT framework has been shown to improve the execution speed of commonly used benchmarks to test the speed of Java Virtual Machines by up to 4 times compared to Android's mainstream, unmodified Dalvik VM. We show that a MB JIT compiler can provide similar results to those a static compiler could while having the advantage of run-time profiling information and taking a relatively small toll on resource usage compared to the performance gain. Finally, we propose a sharing environment in which information resulted from the TB JIT profiling and subsequent compilation of code can be useful for our MB compiler and vice versa in order for each compiler to complement each other's weaknesses and harness their own advantages.

2. OVERVIEW

2.1 Method-Based JIT Compiler flow

Every time an application is launched in Android a new copy of the Dalvik VM is spawned from the system Dalvik zygote. Each DVM acts as an application sandbox with its own Garbage Collection and Compiler Threads. The new DVM opens the application file (called a dex file) and loads into memory the contents of it. In order to speed up the execution of interpreted instructions, a TB JIT compiler checks for hot traces while the program is being run. Any instruction executed a certain amount of times can trigger a trace selection phase inside the interpreter and then a trace compile order will be emitted for the compiler to transform the trace into native code. Our work adds a second threshold to the process. The new threshold applies only for invocation opcodes and is tuned to allow the TB JIT to run as the base JIT compiler to allow for fast optimization of the application by means of short traces being compiled while the MB compiler delivers faster, heavily optimized compiled methods afterwards. When the threshold for invocation instructions is reached, there is no need for a trace selection phase so the compile order is emitted quicker than a trace compile order. Figure 1 depicts the first profiling phase for our MB JIT and how it appends to the existing JIT profiling technique. This thread profiling phase is simpler than Dalvik's JIT because there is no need to build a trace before submitting a method compile order.

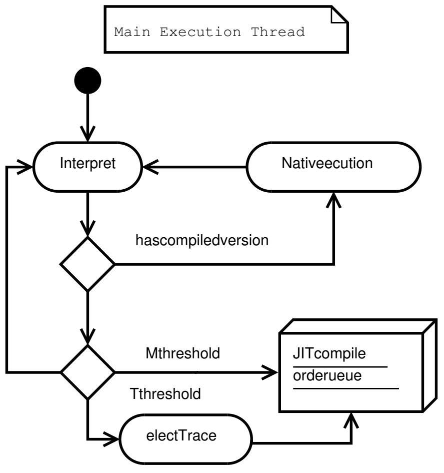

Figure 1: Main Thread State Diagram

In recent times, the tendency has been to create software that can be executed in parallel on multi-core processors. This is often allowed by processes spawning new child processes or by processes being further divided internally as threads. Java virtual machines provide a mechanism to create, handle and kill additional threads apart from the main execution thread [28]. Threads can be created and started using the `Runnable` interface or the `Thread` class and the virtual machine is in charge of creating a new thread and executing the code in the class' implementation of the run method [27]. In the case of the Dalvik VM, this threads are provided via POSIX [17] threads and the corresponding API. The invocation of the `run()` method is not profiled by Dalvik's TB JIT because it is directly invoked from the virtual machine. We propose an additional Class-level profiling phase to be able to select when the `run()` should be considered for MB JIT compilation. Having Class-level profiling information might enable us to consider related methods from the same Class or related Classes for JIT compilation. This kind of semantic information is not being utilized, at the moment, by the JIT compiler framework.

The Compiler Thread works by emptying the compile order queue and processing every trace one at a time. Trace orders are handled by Google's Dalvik VM TB JIT compiler. At this moment, the only methods considered for inlining by the TB JIT compiler are simple methods that fit the profile

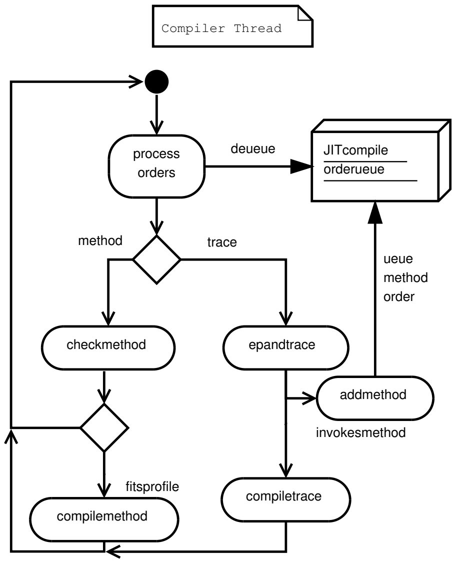

Figure 2: Compiler Thread State Diagram

of `getters` or `setters`. When presented with invocations to other methods then inlining is not realized. In our proposed framework we consider the case in which a method is not inlined counter productive for the JIT compiler's goal. Therefore we add a call to our MB compiler to avoid having a compiled trace call an interpreted method and then return to the native environment because of the overhead that this would mean (interfacing native code with interpreted usually implies the use of JNI or another bridge) [20]. This is the third way a method can be scheduled for MB compilation. We chose to append our MB JIT compiler to the same compiler thread because this, when the profiler is tuned to favor the TB JIT as the base JIT, allows for traces to be compiled first. In this manner, the TB compiler thread does not compete with the MB compiler thread for CPU time, because they are handled by one single compiler thread, and together deliver quicker native versions of traces while the slower method compilations are left for later. The assumptions and derivations we followed for the tuning of our profiler are explained in section 3.2. Figure 2 shows all the states in which the compiler thread can find itself. It also shows the method level profiling phase of our proposed JIT.

A second advantage of our decision to use the same compiler thread is that Dalvik's original security scheme is left unmodified. Every application still runs inside its own virtual machine and every virtual machine has its own compiler thread that has access to its own native compiled code. The only difference is that, for method work orders, after some transformations are applied to the DEX code, the MB JIT is called instead of the original TB JIT.

2.2 The Compiler

The compiler itself uses LLVM's framework to translate optimized DEX instructions directly into LLVM bitcode. Although a trace is not provided for the compiler to work on, the object oriented structures created by the DVM to hold classes, objects and such provide access to the DEX code without having to reload the dex file from the application. The reasons for having chosen LLVM as the compiler's backbone and the benefits that this meant, and ones yet to exploit, are explained in section 4.2.

Using a third party developed compiler such as LLVM's allows us to focus on the comparison and cooperation between JIT types. The idea of building a JVM using third party components had already been fathomed by Geoffray et al. [11] to reduce the expertise needed to implement any of its main components. The JIT compiler is just another one of a JVM's components and its job is to take an intermediate representation (IR), be it DEX code or LLVM bitcode, and generate native code. The compiler thread (its state diagram shown in figure 2) is in charge of calling the right JIT compiler for each work order. The Dalvik VM later sets the jump point from interpreted code to the newly compiled one.

3. PROFILING APPROACH

3.1 Existing Profiling Tools

Google provides a tool called traceview to profile the execution of applications on the Dalvik Virtual Machine. Wang's previous work on an Ahead-Of-Time-Compiler (AOTC), Icing [35], bases its profiling model on information provided by traceview available after the application has been ran once. The information is then considered at static time to determine which methods are good candidates for static compilation previous to repackaging the Android application. The situation for a JIT is a bit different since run time information is readily available and more valuable than any sort that could be obtained ahead-of-time. In most cases user behavior will affect which methods and which code sections are executed more often, this is something that can only be determined during run-time. Further differences with similar compilers and their approaches are discussed in section 6.

Dalvik's TB JIT compiler keeps track of the amount of times most instructions from a program have been called. This information is stored in a hash table and is updated any time a candidate trace entry point is about to be interpreted. The choice of a hash table is definitely based on the need for the interpreter to be able to continue its job without much overhead being added by the TB JIT. Once the amount of times an instruction has been called passes the threshold set for the TB JIT to kick in then the entry in the hash table is reset and the Interpreter goes through a series of steps to decide where the trace should stop, taking into account a parametrized maximum length for any given trace. The trace is later dispatched as a work order for the Compiler Thread to take care of the JIT compilation and attachment of the compiled code where the Interpreter is able to call it. The compiled code, however, is not called until the execution threshold is once again passed; at this moment the interpreter checks if the code has already been compiled, and if it has, then jumps to the native compiled code. At this point we notice that, not only are trace compile orders dispatched earlier, but they will also be called earlier (given the fact that the TB JIT threshold, $thresh_T$, is low enough). With a different tuning, the profiler would easily be able to make the MB JIT the base JIT because the MB compile orders take less time to build and they do not depend on the invocation threshold being passed twice for the native code to be installed.

3.2 Profiling for our Method-Based JIT

We took advantage of the existing profiling mechanism and for every method called we check if a specific MB threshold had been reached. Once the threshold is passed then a work order is submitted to the Compiler Thread for the method to be compiled and made available for future calls. The method is marked so that, in the future, it will not be considered for MB JIT orders and rather be taken into consideration only for trace compilations. This way we avoid re-evaluating rejected methods. Since the native version of the method will replace the invocation to a interpreted method for an invocation to a native one, this only affect in the case that the method was considered not fit for the MB JIT. The modifications we added to this profiling stage are executed in constant time and account for a minimum amount of time in the whole compilation process. The interpreter only needs to check a hash table and update the counter stored in it.

In order to obtain an initial setting for the profiling model thresholds we followed a simple derivation. We defined a `Method` as a finite sequence of instructions: $Method = \{i_0, i_1, i_2, ...\}$; and a `Trace` as a subsequence of instructions $Trace \subseteq Method$ so if we set our MB JIT threshold, $thresh_M$, to any value such that $thresh_M > 1$, the hot traces of instructions inside a method's instruction sequence are sure to be dispatched as trace compile orders if there is a flow control structure inside the method such that the TB JIT's threshold is reached. Actually, although loops, recursive method calls and repetitive event inputs are all examples of situations that might cause an instruction to be ran several times, only loops do not implicitly include the invocation of a method (which would be profiled for the MB JIT as well). With this setting then, an instruction can be called a certain amount of times before the compiler's behavior degenerates into compiled traces being outdated by complete native methods. Consider $exec(i)$ to be the expected amount of times an instruction $i \in Method = \{i_0, i_1, i_2, ...\}$ is to be executed per method call. We can now say that the degenerate case occurs when we have the following conditions:

- $\exists i \in Method \mid exec(i) \geq thresh_T / thresh_M$
- calls to $Method$ exceed $thresh_M$

This means that an optimal tuning should consider low MB thresholds compared to the numerator, otherwise the degenerate case might occur too often. The usual TB threshold for Dalvik (around 40) proved to be a good setting in combination with a relatively small MB one (around 2 or 5). If the TB threshold is increased too much then the speed boosts will be delivered too late for some benchmarks and probably for short-lived applications as well. Of course, the TB maximum trace length and the MB minimum instruction count for consideration also play a role in avoiding attempts to compile a method that has already have most of its instructions or opcodes translated by the TB JIT. However, we depend on the above formulation for the case in which a method is worth being compiled completely even if it contains a loop that might trigger the TB beforehand.

A compiled trace is also called earlier because the process of building the trace itself and then compiling it takes two passes over a $length_{Trace} = |Trace|$ instructions long trace. Our previous derivations imply that $length_{Method} = |Method| \geq length_{Trace}$. For most cases, at the very least in our tests, this is enough time for the trace to be ready the next time $thresh_T$ is reached. The compilation of a method also takes two passes over the instruction sequence but we devised a simple optimization (described in section 4.3) so that we can avoid having to wait for the threshold to be met again until the native version of the method is put in place.

When the compiler thread is faced with a method compilation work order it has to determine whether a compiled version of it will help increase performance or not. First of all, the method should contain no instructions that will indicate a need to jump from native to interpreted environment. In the case of DEX instructions, invocations of non-inlineable methods would end up being compiled to a call to the interpreter to execute the method. Instructions requiring access to fields in order to retrieve or save values can be provided because the JIT sends a reference to the method's owner, be it an Object or a Class, and therefore do not have to be banned. Second, the method instruction count should amount or surpass a threshold set to avoid small methods, that usually only set, get or return a value, from being compiled. The described types of methods would only result in native code islands and would probably only increase the amount of environment jumps between interpreted and native code and would not be benefited from MB JIT compilation, they should instead be inlined.

3.3 Trace-Based vs. Method-Based JIT

Android's TB JIT for the Dalvik Virtual Machine focuses on the hot parts of a method's code and compiles them into native code that is linked to the first instruction from the original trace. Methods with loops and other flow control constructs that cause instructions to be executed repeatedly, will trigger this behavior and cause a trace compilation work order to be emitted. However, if an invocation to a method is included in a hot trace or if a method is being invoked too often, it is probably better to compile all of the method into native code instead of calling an interpreted method and then shift to native code once inside the method and possibly back into interpreted code (if the trace does not extend until the end of the method) before returning. The situation looks even more complicated when a trace containing a method invocation is compiled and the method's code includes a compiled trace of its own. Consequently, we decided that a trace compilation work order including an invocation instruction should also trigger a method compilation order so as to avoid unnecessary environment jumps.

Most literature indicates that loops can be exploited so that observed variable types are used for further optimized code generation [10]. Short traces that are executed for a long period of time over and over again constitute the perfect target for Dalvik's standard JIT. The difference with most dynamic programming languages that are sped up by TB JIT compilers and DEX code is that, although being indeed a type-less registered based intermediate representation, most DEX code opcodes reveal hints as to what the registers' types could be. Translating the registers to LLVM bitcode, which has to indicate variables' types explicitly, is hard but not impossible if we handle registers as 32 or 64 bit integers when only their type size is known. When a later executed instruction reveals the real type of a register it can be casted using the `bitcast` instruction [22]. When we are done with the translation, we are left with an SSA representation of the method, with information regarding all of the variables' types and many options for optimizations.

4. THE METHOD-BASED COMPILER

We based our JIT compiler on LLVM, translating Dalvik's DEX code into LLVM bitcode to be able to use its method-based JIT. LLVM provides libraries and APIs for the translation of any source language into bitcode IR, LLVM's lingua franca, and manipulate the resulting bitcode in many forms.

4.1 DEX code to LLVM bitcode

The Dalvik VM is a register based Virtual Machine. Registers have no type and can be assigned many times through the execution of any given method. LLVM bitcode, on the other hand, has to comply with SSA form and has to load and store values to and from memory if the same variable is to be assigned more than once. Every method compile work order is handled by translating every DEX instruction in the original method to a sequence of IR instructions equivalent to it. The translation is dealt with in two passes: *1*) search for backward branching targets and opcode filtering; and *2*) code translation.

The first pass deals with labeling of backward branch targets and making sure that the method contains no banned opcodes (examples of undesired opcodes for MB JIT compilation are mentioned in section 3.2). The first pass also makes sure the amount of instructions that will be translated exceeds the instruction count threshold for the method to actually benefit from being JIT compiled. Finally, if the method contains only desired instructions then the second pass will translate all DEX instructions into LLVM IR, validate and compile it to memory.

4.1.1 Typing the Typeless

Although Java is a statically typed language, Google's dx tool [12] translates Java bytecode into a typeless register based represensation, DEX code. This presents a problem when translating into LLVM bitcode because all operations on LLVM bitcode have to be executed on operands with strictly the same type. Most DEX opcodes present hints as to what type of data a register holds, but sometimes it is necessary to infer the data from assignment operations or such. To overcome this problem and to avoid having many versions of the same register name, we translate every DEX register into variables representing the register name and their type, i.e. `v5_INTEGER`, and since the variables, in most cases, are references to memory in order to preserve the SSA constraint, any register could represent a number of different variables in its LLVM bitcode translation and have many memory addresses related to it.

4.2 LLVM Advantages

LLVM provides an API to run several transformations through code represented using its bitcode. Once the translation from DEX code to bitcode is done, we can optimize the resulting code using these transformations and optimization passes to get different levels of optimized code. LLVM also provides many ways of running the code. One of them, conveniently for us, is a JIT compiler that returns the address of the memory location of the compiled method. Aside from the obvious reason of convenience, we chose LLVM to build our compiler because of Google's own decision to include it into Android with the Renderscript [34] execution engine. Although our compiler uses the mainstream execution engine to create the JIT compiled code, it can be slimmed down to have a smaller footprint on the operating system, just like libbcc (Renderscript's execution engine) was. The other option would be to modify libbcc to allow for it to compile methods for Dalvik. The latter would not be too hard a modification to our work since our translation to bitcode would only have to call a different library for the last phase, native code emission. The benefits of using libbcc include caching of compiled code (faster applications without having to recompile methods) and the use of a common LLVM execution engine throughout Andoid, which roughly translates to less space on disk.

ODEX Instruction	Original Instruction
`iget-quick offset`	`iget id`
`iput-quick offset`	`iput id`
`invoke-virtual-quick offset`	`invoke-virtual id`
`invoke-super-quick offset`	`invoke-super id`

Table 1: Optimized opcodes

In our experiments, code compiled using LLVM's JIT compiler performs better than code compiled by Dalvik's TB JIT compiler in part because of it's use of ARM's floating point architecture VFP [24]. The standard version of Google's Dalvik VM and its JIT compiler don't use vfp instructions by default and require device manufacturers to modify Android to have the JIT compiler emit different specialized instructions specific for their platforms. LLVM takes care of generating specialized code depending on the platform it was set up for.

4.3 Optimizations

4.3.1 Quick Field Access

Usually native code called from Java has to go through the Java Native Interface in order to have access to fields inside objects or classes even if the address to the container has been passed as an argument to the method. We avoid the situation in which we have to call back into the interpreted environment at all costs, so instead we take advantage of Dalvik's optimized instructions, which have the offset of every field to speed up the retrieval of them. Just as the interpreter uses the optimized opcodes to get fields by adding the base address with the offset of any field, we use LLVM's API to manage pointers to addresses and are able to load fields in constant time. Examples of instructions that are optimized by `dexopt` [33] are shown in table 1.

4.3.2 Installing the Compiled Method

Dalvik's JIT delegates the job of linking the compiled code to the interpreted trace to the main execution thread. After the execution count for an instruction exceeds the $thresh_T$ then the translation cache is checked to verify that a native translation of the trace exists. It is then called instead of the interpreter continuing normally. In our case we are changing a whole method from interpreted to native, the easiest way is to change the method's attributes to tell the interpreter that it requires a call to native code and allow for a Java Native Interface (JNI) call. However, changing the method's attributes would allow for bugs during the execution if there are more than one thread trying to call the method (i.e. one thread calls the native method but the address has not yet been updated) so we modified the `Method` structure inside Dalvik to be able to have additional information and additional checks so that no bugs are introduced. This also allows for faster calls to native code, avoiding some checks and parameter conversion routines that JNI does before calling native code. The result is that our native code pinning process is quicker than how the TB JIT does it and allows for additional information to be available to the native code, compared to what would normally be for a JNI compliant native method, while making sure that regular security checks are still applied to external native code being called by the Java Native Interface (effectively giving special treatment to internal native code).

4.3.3 Invocation to System Libraries

Finally, our choice of LLVM as our MB JIT compiler framework has the advantage that it seamlessly links external function calls to the standard library (i.e. sine or cosine) without any additional library having to be added, resulting in less memory wasted. This allows for static invocations into standard Java libraries to be translated to static native calls. DEX code tries to cope with this by using an opcode (`execute-inline`) to inline common static system routine calls, so we translate both static invocations and inlined invocations to the `java\lang\*` library into external method calls that LLVM deals with.

5. RESULTS

The following tests, mentioned in this section, were ran on a Galaxy Nexus [14] phone running our modified image of Google's latest version of the Android - Ice Cream Sandwich operating system [13]. This platform has a 1.2 GHz TI OMAP 4460 ARM Cortex-A9 dual-core processor and a 304 MHz PowerVR SGX540 (Underclocked from 384MHz) GPU, as wel as 1 GB of memory. Note that the platform does have hardware-based floating point operation support (ARM's Floating Point Architecture - VFP [24]).

5.1 CaffeineMark 3.0 Benchmark

To test our own implemented MB JIT compiler and to provide information we could use to compare it to the TB JIT and existing ahead-of-time compilers for Java, we targeted two of the most commonly used benchmarks for Java virtual machines. The calculation of the number π (pi) up to a specific number of digits and a series of mathematical tests whose execution is counted versus a given amount of time are the main tasks carried out by the BenchmarkPi and CaffeineMark 3.0 Android applications. Both have been used by Google to test their JIT compiler and by Wang et al. [35] to prove the potential of a MB AOTC. The mathematical tests executed by the CaffeineMark benchmark include the Sieve of Erathosthenes, the execution of many code branches, loops and floating point calculations.

Our tests suggest that methods compiled using our MB JIT compiler can be almost twice as fast as methods that have only been optimized by Dalvik's standard JIT. Figure 3 shows how the `Sieve`, `Loop`, `Float` and `Logic` tests from the CaffeineBenchmark3.0 are faster when ran on our proposed hybrid JIT. `Float`, `Loop` and `Logic` tests being the ones that enjoy the most benefits from being compiled using the MB JIT. The figure also shows that the MB JIT by itself appears to be better then the proposed hybrid JIT framework (`Loop` and `Logic` tests). This is due to the fact that the TB and MB JIT compilers by themselves create less compile orders and

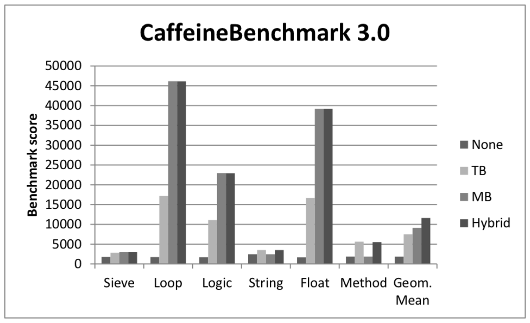

Figure 3: CaffeineMark 3.0 benchmark results (the higher the better)

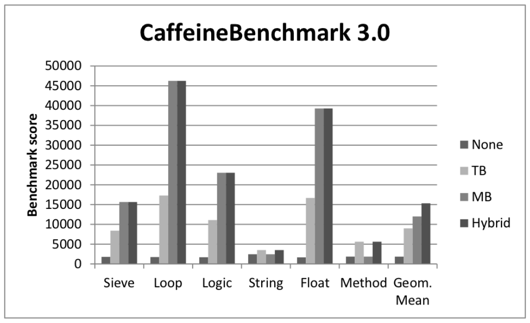

Figure 4: CaffeineMark 3.0, second run

so the native versions of the traces or methods are delivered and consequently used earlier, resulting in the native code providing earlier perceivable performance gain. However, the proposed hybrid JIT, because of our profiling choices, delivers performance gain almost as good or sometimes even better than that of the TB JIT on initial runs and much better on following ones.

Since our modifications to Dalvik's profiling system create an extra flow of work into the compiler thread, it is to be expected that some of the compilation orders are not attended at the very start. The profiler settings can be tuned to allow for compile orders to be dispatched earlier but this would rather be a problem if the compiler thread becomes overwhelmed and neither type of JIT would deliver on time. This behavior is not unique to our proposed hybrid JIT, Android's standard TB JIT also exhibits better performance results during the second run (especially for the first tests that were run by the benchmark, i.e. `Sieve`). Figure 4 shows how some of the tests in the CaffeineMark3.0 benchmark are much quicker during a second execution of the application since the compiler thread has dealt with all the compile orders in queue. Code as much as twice as fast as the one delivered by the TB JIT is observed for the `Sieve` test and, although not as easily visualized on the graph, for the `Logic` test after executing it more than once.

In both figure 3 and figure 4 it is also evident that the `Method` and `String` tests were rejected by one of the MB JIT profiling phases. Their score is almost as good as the one corresponding to the TB and the virtual machine with no JIT compiler (the first compared to the hybrid JIT and the latter compared with the MB JIT alone). Although the MB compilation of these tests has been rejected, the additional

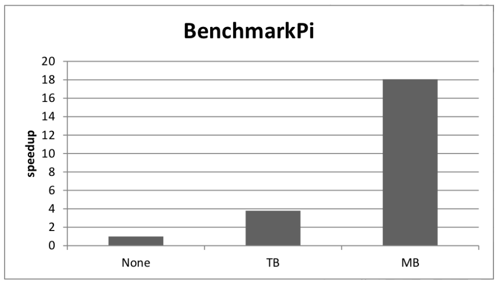

Figure 5: Pi Benchmark results

lag introduced by our proposed profiling techniques is almost non-existent, even though the first run of this application generates many compile orders (both traces and methods) for the compiler thread to handle all at once. Although we do not expect our present profiling tuning to be perfect for all situations, we observe that there at least exists such a configuration for applications behaving similar to the ones we tested.

5.2 BenchmarkPi

Another well-known benchmark used by Google to benchmark their JIT is the BenchmarkPi [31]. The benchmarks computes the number π up to a specific amount of digits and times the computation. Our results show that performance improvement achieved by our hybrid JIT framework can obtain similar results to those of an Ahead-Of-Time-Compiler such as Wang et al.'s Icing [35]. Figure 5 shows how fast the benchmark can be ran once both TB and MB JIT compilers have done their work compared to having only a TB JIT. Our results show that a MB JIT outperforms Android's TB JIT by almost a factor of 4 in such computation-intensive applications. MB JIT compilers are expected to outperform simple TB JIT compilers on such computations but the huge difference in performance is mainly due to LLVM's advantages previously mentioned in section 4.2.

5.3 Memory Overhead

Our proposed hybrid JIT compiler framework transforms complete methods into a faster native version, but in doing so requires more memory than a TB JIT compiler which would only translate a subset of the method's instructions. This compilation of both hot code and cold code in methods is also the reason why compiling methods takes longer than compiling traces, which consist only of the hot code. The memory overhead in which we incur by compiling whole methods is proportional to the amount of instructions each compiled method contains. Figure 6 shows both the TB JIT and hybrid JIT compilers and their memory usage behavior throughout one of our tests. The memory usage of the hybrid JIT is made up of its native heap and compiled methods, both shown on the graph. As is expected, compiling methods is costly and therefore has to be done smartly and sparingly. This reaffirms the importance of our framework's profiler having to be tuned in order to provide good performance without having to exhaust the system's resources. It also confirms that the decision to keep the default TB JIT compiler as the base JIT compiler for a hybrid framework is the best idea for a resource constrained device.

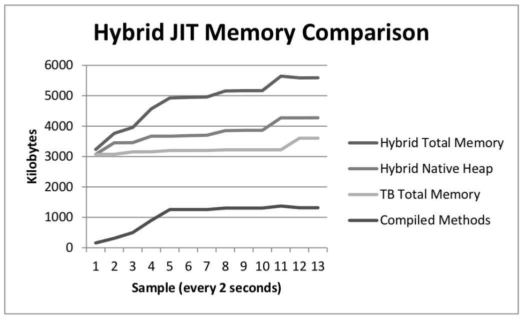

Figure 6: Hybrid versus trace-based JIT memory overhead (Hybrid Total Memory is the sum of its Native Heap and Compiled Methods)

6. RELATED WORK

During Google I/O 2010 [6], Google announced that the TB JIT they now provide for Android was only the initial step towards improving their application platform. The compiler's code reveals hints that plans for extensions towards making the trace analysis include complete methods have been considered, and the outline for a basic MB compiler is also laid out but not fully implemented. Their proposed MB compiler scheme includes an SSA transformation stage, register allocation as well as translation phases to two intermediate representations before generating native instructions (all of which LLVM provides to our proposed framework without the need for more than one intermediate representation). Google also fathomed the idea of both JIT types coexisting in Android but there have been no further comments or developments on this project.

Other authors have tried to compare mature MB compilers with their own TB JIT compilers [18] but haven't done so in a resource constrained environment. Nor have they measured a limited TB JIT's capabilities versus a MB JIT. The cooperation between a limited TB compiler, like the one provided by Google for the DVM, and our proposed MB JIT delivers a speed boost quickly after starting an application and replaces slow, hot methods with fast and optimized native ones after they have been invoked a few times. It does so without having to compile all the DEX code and whithout replacing Android's TB JIT as the base JIT compiler because it is more mature in the Android operating system. It is also worth mentioning that the work by Inoue et al. [18] does talk about a TB compiler and its advantages over a regular MB one but refers to a compiler that has the ability to analyze traces that span more than one method and therefore is not comparable to our study.

6.1 Disassemblers

There are many tools available for the disassembly of DEX code or complete Android packaged applications into text files containing DEX source code. Most of them read the dex file format and obtain all the strings from the constant pool and the opcodes for each instruction in order to construct a stream of human readable dex code. Smali and Baksmali [19] as well as Android's DexDump [12] tool are some of the most popular. Baksmali can even be used to reconstruct an application from modified DEX source files. However, all of these tools go through an unnecessary step by transforming DEX opcodes into readable text, which our compiler does not need, and therefore are not suited for a fast DEX IR to LLVM IR transformation like the one we implemented.

DEX disassembler programs are able to output human readable code after parsing DEX code. Disassembler applications are similar to our DEX to LLVM IR translator because they also decode DEX code and transform it into another format, plain text. Understanding of the DEX code format is necessary to be able to read the instructions correctly from the DEX file, be able to group them using a common format and finally translate them (a similar process is used by the DexDump tool).

6.2 Method-Based Compilers

The Icing Ahead-Of-Time-Compiler [35] provides a fairly good attempt at profiling the Dalvik VM's behavior and determining which methods are better off being completely native instead of just having traces compiled. Our profiling model is based on parts of their work, such as the avoidance of excessive environment jumping and attempting to access object fields directly from object pointers. However, the impasses prompted by the use of the Java Native Interface are no longer present in a run time compiler and cooperation with the TB JIT compiler makes it possible for easier method inlining and a more accurate decision on which methods can be considered hot.

6.2.1 Zero and Shark

Zero and Shark [5] are an implementation of a Java Virtual Machine using zero architecture dependent code. The project aimed at delivering a more portable version of the Virtual Machine that could be used by Linux on other architectures besides x86. This zero-machine code OpenJDK port is profiled by a MB JIT that works on top of the LLVM framework. Shark, the JIT compiler, has dealt with all the issues involved with plugging a JIT into a Java VM. Most issues include dealing with the Garbage Collector and the way references are deemed unused. Our work takes some ideas from their implementation, but the Dalvik VM internals differ a lot from regular Java virtual machines, and so most of their insights are not of much use for the Android application platform.

6.3 Combined JIT Frameworks

Most attempts at combining both trace and MB JIT compilers have been on the field of JavaScript engines. During the past few years most web browser companies have developed a fairly good combination of a quick MB JIT compiler to avoid having to interpret, sometimes at all, and a slower TB compiler that applies aggressive optimizations and is supported by run time information to replace initially compiled code with several versions of heavily optimized traces. Our approach is quite different since we propose the MB JIT compiler to be the one with heavy optimization settings and leave early speed boosts up to Android's TB JIT compiler, the latter providing a few optimization considerations but not as complete as static compilers and others have to offer.

The work by Lee et al. [23] provides good insight regarding whether JavaScript engines should put so much effort into compiling all the script to avoid interpretation. Benchmark software exhibit many opportunities for hot traces to be compiled into extremely efficient native code but real web pages' JavaScript code does not necessarily contain so many instructions being called again and again. Their work concludes that a good interpreter matched with a JIT compiler that is used only when the situation really calls for it, is often better than trying to compile everything. Google has already adopted this idea with their quick interpreter being optimized only when necessary by their TB JIT. Additionally, our hybrid framework builds on top of the same idea by compiling methods only when certain conditions are met.

6.3.1 Chrome's V8

Google's JavaScript execution engine V8 [15] translates most of the script into native code as soon as it is downloaded. The JavaScript is translated using a MB compiler initially. It then uses a TB JIT to optimize the code being executed so that loops, and run time observed types can be used to compile faster versions of portions of the code.

6.3.2 JaegerMonkey

Firefox's mature JavaScript execution engine, SpiderMonkey, has been upgraded to TraceMonkey and more recently to JaegerMonkey [26] to allow the code to enjoy an initial quick boost from the MB JIT. The TB JIT then applies late improvements to hot loops and other traces with additional run time information.

7. CONCLUSIONS

First and most importantly, this work proves that a limited TB JIT compiler is still no match for a robust MB JIT such as the one provided by the LLVM compiler framework. More specifically, most advantages one could achieve by favoring the implementation of a TB JIT over a MB one are minimized by the properties of DEX code, which can be ”lazily” transformed back into a typed representation. Although type specialization and various versions of optimized code might not be good optimization approaches for DEX code, it turns out that the TB JIT works really well for considerably long loops, especially if the trace does not invoke other methods. Other DEX code properties might later be studied to reveal further optimization opportunities available to either JIT compiler type.

7.1 JIT Type Comparison

Dalvik's TB JIT compiler achieves performance boost quickly by compiling instruction sequences that are being called too often, hot traces. When possible, it inlines small methods being invoked by the instructions contained in the trace. In contrast, the MB JIT compiler we designed is capable of compiling most methods, as long as they pass the profiling steps. The compiler is able to apply optimizations regarding the ”liveness” of variables and other optimization techniques that are only possible because the whole method is being taken into consideration. Further transformations similar, if not the same as, GCC's O3 and O2 battery of optimizations [8], are also available to our MB JIT compiler through LLVM's clang [3] and jello [21] projects. Although our MB JIT does incur on more memory usage than a basic TB JIT, and the average compilation time for a method might be up to one-hundred (100) times that of a small trace, when hot methods are chosen correctly (and this can be fine-tuned depending on the platform system), performance can be greatly improved.

7.2 JIT Behavior

From our tests we have also come to the conclusion that both TB and MB JIT compilers become idle after an application has been executing for some time. A benchmark is a good example of this situation, when the same methods are ran several times and not much changes between runs. We believe this idle time can be used for further aggressive optimizations and even some unsound optimizations, as categorized by Guo and Palsberg [16], could be attempted in order to increase the execution speed of the program under the current observed state as long as a side-exit [10] is made available. The previously described behavior could be proposed for devices that are currently connected to a fixed power source, and therefore in a state where further resource usage can be afforded without compromising the battery's life. Google had proposed the introduction of a dynamically adjustable profiling approach [6], but right now Dalvik's JIT's behavior does not adapt much to the hardware's state.

7.3 Relevance

Our proposed hybrid JIT compiler takes ideas from what current web browsers are trying to do to speed up the interpretation of a scripted language. Java and .NET technologies, however, do not output a script but instead transform code into an intermediate representation, which could be stack or register based and that is easily transformed into machine code by the respective implementations of their virtual machines for all the architectures they support. Each intermediate representation has been conceived with specific features that may allow a TB or MB JIT to compile it easily into native code. As we have proven in our work, hybrid JIT compilers can be easily tuned depending on how fast performance gain is needed and code behavior. Although our test platform is a resource constrained device running an Android operating system, different systems run programming languages that are stored as intermediate representations and so these systems could possibly benefit from the inclusion of a similar tool.

8. PROJECT STATUS AND FUTURE WORK

8.1 Tool Status

Many methods have been successfully compiled using our MB JIT but not all methods can be successfully emitted into memory as native instructions by LLVM's current JIT. All previous phases (namely translation, transformations and optimizations) are completed correctly and promptly but the JIT's support for ARM is limited and probably buggy or broken in some parts. To overcome this problem with some methods the profiler indicated had to be processed by the MB JIT, we were forced to execute the last step (compiling from assembler to native code) manually using LLVM's static compiler. The LLVM team is currently working on a new version of the compiler, MCJIT [2], and will deprecate the current JIT framework.

Our framework also supports memory petitions to the Dalvik VM from MB JIT compiled code, and therefore requires that we handle garbage collection in an efficient manner. We currently yield to the garbage collection thread every time we have a request for virtual machine managed memory. This is necessary because the Dalvik VM does not pause threads running on native mode to allow garbage collection. If we did not yield at this moment an interpreted heap memory request might be denied because of yet uncollected garbage, even if there is unused memory. This technique, however, is an overly simplistic solution. Since the benchmarks used to measure the performance of our framework did not result on compilation of memory requests, further testing of our approach to manage native and the virtual machine's memory has to take place before being able to conclude anything relevant on the matter.

8.2 Future Work

There are many other benchmark applications available in Android's application market that can be tested to get more conclusive results on how TB and MB JIT compilers behave. There are even different versions of the CaffeineBenchmark which include image, graphics and other tests which are not included in the version used for our experiments [7]. Moreover, the effect of using our hybrid JIT approach on real applications, apart from benchmarks, is still unknown. In order for further tests to be conducted more DEX opcodes have to be translated into LLVM bitcode.

Although the cooperation between both types of JIT compiler has been shown to improve performance and each one has covered many of the other's weaknesses, these compilers are far from finished. Google's TB JIT for the Dalvik VM still lacks support for traces spanning through more than one method. The implementation of this feature would mean that the profiling model proposed in section 3.2 would have to be reevaluated. The TB JIT could also be extended to profile MB JIT compiled code and attach optimized versions of hot traces inside hot methods as was proposed by Bala et al. [4]. This extension to the TB JIT could work on top of a native version of the method as well as its original DEX code. In this case the TB JIT would also be able to inline native code invocations, like those done through the Java Native Interface, which previous work on the Dynamo project [4] indicate would increase performance.

Another possibility for future comparisons would be splitting the compiler thread into a MB JIT thread and a TB JIT thread. This would probably result in optimized compiled methods being delivered earlier but would definitely make it more difficult to avoid the problem of compiling a trace only to later compile the whole method. Finally, we propose testing different methods of communication between the interpreted and native parts of the code. Google has already designed Renderscript [34] with a parallel thread in charge of the execution of the compiled script and Asghar et al. also describe an alternative way, A-JUMP [1] for parallel applications using multiple programming languages. Such techniques could reduce the amount of time wasted in environment jumps and would allow both environments to communicate more often without having to avoid it for performance's sake [20].

9. REFERENCES

[1] S. Asghar, M. Hafeez, U. A. Malik, A. ur Rehman, and N. Riaz. A-jump, architecture for java universal message passing. In *Proceedings of the 8th International Conference on Frontiers of Information Technology*, FIT '10, pages 34:1–34:6, New York, NY, USA, 2010. ACM.

[2] C. S. D. at the University of Illinois at Urbana-Champaign. The llvm target-independent code generator. `http://llvm.org/docs/CodeGenerator.html#mc`, 16 January, 2012.

[3] C. S. D. at the University of Illinois at Urbana-Champaign. clang: a c language family frontend for llvm. `http://clang.llvm.org/`, 2012.

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *SIGPLAN Not.*, 35(5):1–12, May 2000.

[5] G. Benson. Zero and shark: a zero-assembly port of openjdk. `http://today.java.net/pub/a/today/2009/05/21/zero-and-shark-openjdk-port.html`, 2009.

[6] B. Cheng and B. Buzbee. A jit compiler for android's dalvik vm. `http://dl.google.com/googleio/2010/android-jit-compiler-androids-dalvik-vm.pdf`, 2010.

[7] P. S. Corporation. Caffeinemark 3.0 benchmark. `http://www.benchmarkhq.ru/cm30/`, 1997.

[8] I. Free Software Foundation. Gcc, the gnu c compiler. `http://gcc.gnu.org/`, 05 March, 2012.

[9] I. Free Software Foundation. Gcj - the gnu compiler for the java programming. `http://gcc.gnu.org/java/`, 2012.

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

[11] N. Geoffray, G. Thomas, C. Clément, and B. Folliot. A lazy developer approach: building a jvm with third party software. In *Proceedings of the 6th international symposium on Principles and practice of programming in Java*, PPPJ '08, pages 73–82, New York, NY, USA, 2008. ACM.

[12] Google. Android developer tools. `http://developer.android.com/guide/developing/tools/index.html`, 2007.

[13] Google. Google android - an open handset alliance project. `http://code.google.com/android/`, 2008.

[14] Google. Galaxy nexus. `http://www.google.com/nexus/`, 2011.

[15] Google. V8 javascript engine. `http://code.google.com/p/v8/`, 2011.

[16] S.-y. Guo and J. Palsberg. The essence of compiling with traces. *SIGPLAN Not.*, 46(1):563–574, Jan. 2011.

[17] T. IEEE and T. O. Group. Posix.1-2008: The open group base specifications issue 7. `http://pubs.opengroup.org/onlinepubs/9699919799/`, 2001 - 2008.

[18] H. Inoue, H. Hayashizaki, P. Wu, and T. Nakatani. A trace-based java jit compiler retrofitted from a method-based compiler. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, pages 246 –256, april 2011.

[19] JesusFreke. smali/baksmali: An assembler for android's dex format. `http://code.google.com/p/smali/`, 2009.

[20] D. Kurzyniec and V. Sunderam. Efficient cooperation between java and native codes âĂŞ jni performance benchmark. In *In The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications*, 2001.

[21] C. Lattner, M. Brukman, and B. Gaeke. Jello: a retargetable just-in-time compiler for llvm bytecode, 2002.

[22] C. A. Lattner. Llvm: An infrastructure for multi-stage optimization. Technical report, 2002.

[23] S.-W. Lee and S.-M. Moon. Selective just-in-time compilation for client-side mobile javascript engine. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, CASES '11, pages 5–14, New York, NY, USA, 2011. ACM.

[24] A. Ltd. Arm: Floating point architecture. `http://www.arm.com/products/processors/technologies/vector-floating-point.php`, 2012.

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

[26] Mozilla. Jaegermonkey. `https://wiki.mozilla.org/JaegerMonkey`, 2010.

[27] S. Oaks and H. Wong. *Java Threads*. O'Reilly Media, Inc., 2004.

[28] Oracle. Java se specifications. `http://docs.oracle.com/javase/specs/`, 2012.

[29] Oracle. Openjdk's hotspot vm. `http://openjdk.java.net/groups/hotspot/`, 2012.

[30] Oracle. Oracle. `http://www.java.com/en/about/`, 2012.

[31] V. K. Polychronis. Benchmarkpi - android benchmarking tool. `http://androidbenchmark.com/`, 2009.

[32] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson. Toba: Java for applications - a way ahead of time (wat) compiler. In *In Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, pages 41–53, 1997.

[33] T. A. O. S. Project. Dalvik optimization and verification with dexopt. `http://www.netmite.com/android/mydroid/dalvik/docs/dexopt.html`, 2008.

[34] G. Tim Bray. Renderscript. `http://android-developers.blogspot.com/2011/02/introducing-renderscript.html`, 09 February, 2011.

[35] C.-S. Wang, G. Perez, Y.-C. Chung, W.-C. Hsu, W.-K. Shih, and H.-R. Hsu. A method-based ahead-of-time compiler for android applications. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, CASES '11, pages 15–24, New York, NY, USA, 2011. ACM.