PAYJIT: Space-Optimal JIT Compilation and Its Practical Implementation

Jacob Brock, Chen Ding
University of Rochester, USA
{jbrock,cding}@cs.rochester.edu

Xiaoran Xu
Rice University, USA
xiaoran.xu@rice.edu

Yan Zhang
Futurewei Technologies, USA
yan.zhang@huawei.com

Abstract

Just-in-time (JIT) compilation in dynamic programming languages can improve execution speed in code with hot sections. However, that comes at the cost of increased memory usage for the storage of compiled code and associated bookkeeping data, as well as up-front compilation time for the hot code.

The current standard JIT compilation policy (used in both Android's JIT compiler and the HotSpot Java compiler) is simplistic; any method that has reached a certain hotness, as counted by invocations and loop iterations, is scheduled for compilation. This would be fine if the cost/benefit of compilation was the same for every method of a given hotness, but that is not the case. In fact, compiling a large method will likely result in *less* overall speedup than compiling a smaller, equally hot method. This exposes an opportunity for improvement. We propose the PAYJIT compilation policy for method-based JIT compilers, which scales compilation hotness thresholds with method sizes, and includes two-point tuning, a mechanism for determining those hotness thresholds. In this way, PAYJIT compiles more small methods, it compiles them sooner, and it nearly eliminates the compilation of large methods that contribute little to overall speedup.

Among 10 popular Android apps tested across 16 experiments, the PAYJIT compilation policy decreases compilation time by a maximum of 49.2%, with an average of 18.7%; execution time by a maximum of 10.9%, with an average of 1.34% (for a geometric mean speedup of 1.46%); and code cache size by a maximum of 41.1%, with an average of 15.6%.

CCS Concepts • **Software and its engineering → Just-in-time compilers**; **Dynamic compilers**; *Runtime environments*;

CC'18, February 24–25, 2018, Vienna, Austria

ACM ISBN 978-1-4503-5644-2/18/02…$15.00
https://doi.org/10.1145/3178372.3179523

Keywords JIT Compilation, Method-Based JIT, Code Cache, Dynamic Runtime, Android Runtime

ACM Reference Format:
Jacob Brock, Chen Ding, Xiaoran Xu, and Yan Zhang. 2018. PAYJIT: Space-Optimal JIT Compilation and Its Practical Implementation. In *Proceedings of 27th International Conference on Compiler Construction (CC'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3178372.3179523

1 Introduction

Just-in-time (JIT) compilation in dynamic programming languages can improve execution speed in code with hot sections. However, that comes at the cost of increased memory usage to store JIT compiled code (and for the compiler thread), as well as higher CPU and energy demand for compilation. The Android Runtime system (ART) uses JIT compilation, which can require megabytes for storage of code and related data for a single app. Current state-of-the-art JIT compilers (e.g., HotSpot, Android's JIT compiler) determine which methods merit compilation based on the amount of time spent in them. As a proxy for that time, they use method invocations plus loop iterations. This proxy is called *hotness*, and each method that exceeds a universal hotness threshold is compiled.

There is a serious problem with this approach, which we solve. A single hot method often contains both hot and cold code, and the larger the method is, the more likely this is to be the case. In our experiments, neither hotness nor size is evidently a good proxy for time spent in a method. In fact, the reverse is true; per hotness event, *less* time is spent in larger methods. Thus, compiling such methods results in less time-savings per byte of memory. We solve this problem with *PAYJIT*, a novel, data-driven compilation policy that scales compilation thresholds based on the expected cost and benefit of compiling and storing the method. PAYJIT includes *two-point tuning*, which allows the system designer to optimize compilation thresholds for methods of all sizes by tuning the thresholds at only two sizes. We implemented and tested PAYJIT in the Android Runtime JIT compiler.

1.1 Resource Constraints in Android Devices

Due to the tightening resource constraints that Android needs to operate under, there is a need for memory-efficient and cycle-efficient code generation in ART. We chose to implement PAYJIT in ART due to Android's ubiquity and its

unique resource constraints. The Android operating system has dominated the smartphone market since 2011. In the first quarter of 2017, 86% of all mobile phones purchased ran on Android [25]. Current low-end Android devices have tight memory constraints; many have only 2 GB of RAM, and some have only 1 GB. Moreover, cost-constraints in developing markets mean that there is a market for devices with as little as 512 MB (Google's Android Go project is aiming to provide a very-low memory Android platform for such devices [4]).

The Android OS itself uses much of the available RAM on a device. For example, on our Google Pixel phone running `android-7.1.1_r26`, the OS process already uses 656 MB of RAM immediately after bootup. After about 45 minutes of use, approximately 1.7 GB of RAM were in use. Even on a new low-end device with 2 GB of RAM, each megabyte saved would represent 0.3% of the remaining available memory; on devices with 1 GB of RAM or less, a single megabyte of savings would be much more valuable. Therefore, reducing application memory usage can substantially ease pressure on memory. This will be particularly true in future low memory devices in developing markets. Additionally, most Android devices are energy-constrained, and minimizing CPU demand (by compiling less code, or by compiling hot code sooner) will help extend battery life. While there has been much research on improving JIT compiled code performance, the tight memory constraints of Android devices present a new problem. Our PAYJIT policy decreases both memory and CPU demand.

The main contributions of the work are as follows:

- A characterization of code size and execution frequency for Android applications, a new metric called *time per hotness event* to quantify the cost and benefit of memory use in the code cache, and a policy called PAYJIT that makes optimal use of the code cache based on oracular knowledge.
- A compile-time model to predict the cost and benefit, and a non-linear variable-value hotness threshold based on a technique called two-point tuning.
- An implementation in Android 7 and evaluation on 10 widely used mobile applications in the areas of social networking, news, messaging, and online shopping, with billions of downloads.

The remainder of this paper is organized as follows: Section 2 describes the Android Runtime (ART) system and its JIT compilation policy; Section 3 motivates and introduces *PAYJIT*, a new compilation policy that favors compilation of smaller methods, saving both space and time; Section 4 describes our experiments and analyzes their results; Section 5 outlines prior work in JIT compilation policies and code caching; and finally, Section 6 concludes the paper.

2 The Android Runtime JIT Compiler

This section describes the structure of the ART system with respect to the execution of application code. In both our description of the Android Runtime (ART) system, and in our experiments, we use `android-7.1.1_r26`, a device-specific version of Android Nougat for the Google Pixel.

2.1 History of ART

Until Android 5, Android executed applications from *DEX* bytecode (which is compiled from Java application code) using the Dalvik virtual machine. Prior to Android 2.2 (FroYo), the DEX code was only interpreted. Android 2.2 executed DEX code using a combination of interpretation and trace-based JIT compilation. Android 5 (Lollipop) introduced a new execution model for DEX bytecode with the Android Runtime, exclusively using ahead-of-time compilation whenever possible. In Android 7 (Nougat), ART uses interpretation, method-based just-in-time compilation, and ahead-of-time compilation. Section 2.4 describes the structure of the runtime system in Android 7. For the rest of this paper, when saying ART, we are referring to ART in Android 7 (and more specifically `android-7.1.1_r26`).

2.2 Method-Based JIT Compilation

JIT compilers may be classified as either method-based or trace-based, depending on the granularity of compilation. Method-based JIT compilers compile one method at a time (though they may also use inlining). Trace-based JIT compilers compile hot code paths that may begin and end in the same or different methods. Inoue et al. [14] retrofitted the IBM J9/TR Java method-based JIT [10] compiler with trace-based compilation in order to compare the two techniques. They found that the key advantage of trace-based JIT was its ability to compile larger sections of code. However, the trace-based execution model has higher overhead, so their trace-based JIT compiler only achieved at 92.8%-95.5% of the method-based JIT performance. The ART compiler is method-based.

2.3 Code Cache Structure

After a method is JIT compiled, it is stored in memory, in a structure called the *code cache*. The code cache is initialized with a 64 kB capacity. It contains one 32 kB `mspace`[1] for code, and another 32 kB `mspace` for associated data (stack maps and profiling information). Each of these keeps a 32 kB limit on the amount of space that may be acquired from the system, although less space may be used. When the capacity of either is reached, the code cache is *garbage collected*, removing code that is no longer the entry point for its corresponding method (e.g., because that method has been deoptimized, its class has been unloaded, or it has been recompiled with different

[1]The type `mspace` provides an interface for mallocing space, with limits on total size.

optimization based on profiling information), as well as the data associated with these methods, and the capacities of the code and data memory regions may be increased (in tandem), up to a maximum of 32 MB each. The "size" of the code cache is monitored as the sum of the spaces used by each method (data is monitored separately).

2.4 Compilation Policy

When an app is first run, every one of its DEX methods is executed by the interpreter. The Android Runtime measures the hotness of each method as the number of invocations plus the number of loop iterations. This has generally been accepted as a proxy for time spent in the method (though we argue that this metric is flawed). ART has three different hotness thresholds at which the runtime acts: warm, hot, and OSR (on-stack replacement is explained below). When the warm threshold (5,000) is reached, the method is marked for ahead-of-time compilation next time the device is idle and charging, and profiling data is generated and placed in the code cache. When the hot threshold (10,000) is reached, a new JIT compilation task is created, and placed in a thread pool for background compilation. When the method is compiled, code cache space is allocated for its code, as well as for a stack map for it. The OSR threshold (20,000) may be reached during execution of a method if the method is in a hot loop. In this case, a high priority compile task is created for the method, and run in the background. Once compilation is completed, execution may continue with the compiled code. This mechanism is known as on-stack replacement (OSR).

3 The PAYJIT Compilation Policy

The PAYJIT compilation policy is designed to efficiently utilize both memory space and processor time. This is done by giving each method a different hotness threshold, based on the expected cost of compiling and storing it, and the expected benefit of having a compiled version to run (instead of interpreting its DEX code). This section presents evidence of shortcomings of the state-of-the-art approach to JIT compilation, formulates the new PAYJIT policy which solves them, and describes our implementation of it in ART.

3.1 Motivation for a New Policy

As detailed in Section 4, we performed 16 experiments using ART's default JIT compilation policy, each automating the 10 apps enumerated in Table 1 in succession. In these experiments we see that there are a small number of large methods which occupy a significant fraction of the code cache. These larger methods cost more to compile and store, but do not provide more performance improvement than smaller ones. This section outlines our observations from these experiments and shows the need for a smarter compilation policy.

Figure 1 shows two histograms from the experiments. The blue histogram shows the percent of methods compiled by

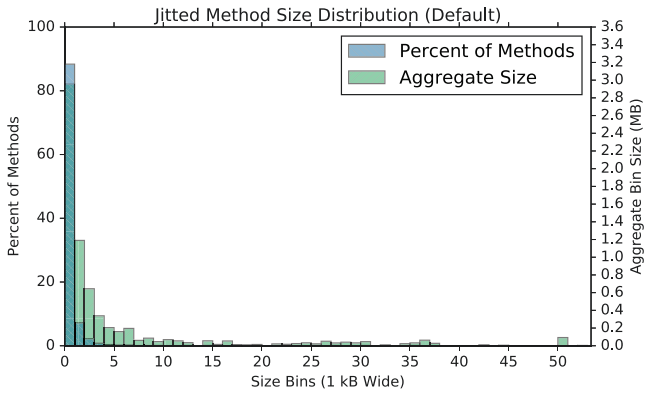

Figure 1. A histogram showing the compiled sizes of JIT compiled methods, on average, under the default compilation policy. Blue bars (lower bars for all but the first if rendered in black and white) show the percent of compiled methods in each size class, and green bars show the total sizes of all methods in each class. For example, in the average run, almost 90% of compiled methods were under 1 kB, and their sizes summed to about 3 MB.

ART's default compilation policy that are between 0 and 1 kB, 1 and 2 kB, etc., after compilation. The green histogram shows the total size of all methods in each size class. That is, the green bars show the impact of each size class on the code cache. The figure shows that significant space is used by the very small proportion of methods that are larger than a few kilobytes. E.g., only 1.4% of methods are over 4 kB, yet they account for 24% of space (in contrast, 88% of methods are less than 1 kB, accounting for 44% of space).

Due to space constraints, we omit individualized histograms for each APK. Note, that `facebook` is responsible for most methods larger than 20 kB in Figure 1, while `wechat` `twitter`, `airbnb` and `pinterest` are each responsible for a few.

This space usage would be justified if the compilation of larger methods resulted in a proportionate performance increase, but that is not evidently so. We instrumented ART to log hotness events[2] for each method, and we used a tracer (detailed in Section 4) to measure the time spent in each method. Using this instrumentation, we ran a short experiment with the `airbnb` apk. Figure 2 shows the trend between exclusive time[3] per hotness event and DEX code size in this experiment. The blue dots show the average exclusive time for each method in 64-Byte bins (we show the data in bins to highlight the negative trend). Counterintuitively, we see that methods with less DEX code typically take more time per hotness event, and larger methods take less. We suspect there are two reasons for this. First, a method can become

[2]Hotness events are events (method invocations and loop iterations) that increase a method's hotness count.
[3]Exclusive time is the amount of time spent in a method, excluding time spent in its descendants.

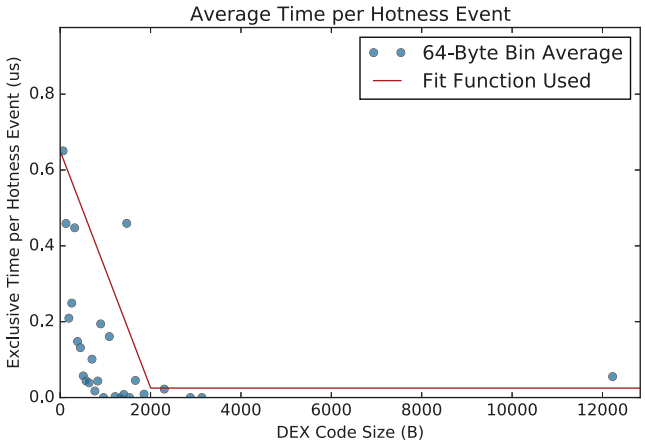

Figure 2. A plot of exclusive time per hotness event vs. DEX code size for a short run of airbnb. Blue points indicate averages of all methods in 64-byte bins (in order to highlight the trend). The red line indicates a fit function, which we use in the evaluation of PAYJIT.

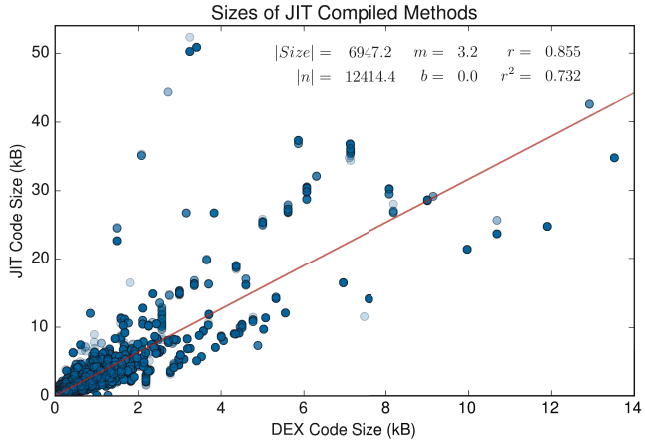

Figure 3. A scatter plot comparing the DEX-code and jitted-code sizes of each method compiled by the default JIT compilation policy, across all 10 APKs and all 16 experiments. Dots are transparent to show the density of data points. The plot shows a strong linear relationship (shown in the red line and the numbers in the upper-right), indicating that a method's DEX-code size can be used as a proxy for the expected code-cache usage in making the compilation decision.

hot even if only a small fraction of its code is heavily used. Second, larger methods are more likely to have large conditional statements, meaning that less of their code is actually executed.

In summary, the default JIT compilation policy compiles a small number of large methods which occupy *more* space, but contribute *less* to overall speedup. We propose a compilation policy that does not compile such methods until they reach a higher hotness threshold that justifies the memory space their compiled code will occupy. Of course, prior to compilation, the compiled size of a method is unknown. Figure 3 shows that there is a correlation ($r = 0.855$) between the size of a method's DEX code (which is available at run-time) and the size of that method after compilation. The PAYJIT policy uses this relationship to predict the compiled size of each method, and favor compilation of smaller methods.

3.2 Motivation for a Cost-Based Policy

In a system with multiple actors and multiple scarce resources, such as an economy or a computer, it is useful to have some means of comparing the value of different resources. For example, if multiple businesspersons need access to both limited storage space for their goods and the limited time of a business consultant, there must be a way to compare the cost of each resource, even though they are very different things. In this scenario, the obvious solution is money; each individual will decide for themself, based on the prices and expected profit from acquiring the resources, what quantity of each to purchase. We call models which use a common unit (money or otherwise) to assign values to different resources *cost-based* models.

A cost-based page replacement policy for main memories called VMIN [21] assigned costs to both memory space and time spent servicing page faults. Knowing the time between the current page access and the next access to that same page, VMIN keeps the page in memory if storage for that duration of time is the least costly option, or removes it after use if a page fault is less costly. In this way, it achieves the lowest possible page fault rate for a given average memory usage. The well-known Working Set page replacement policy (WS) is analogous to VMIN, without requiring knowledge of future page requests [6].[4] WS keeps a page in memory after it is used until the maximum time for which the VMIN model would hold it; at that point the page is marked eligible for eviction. Note that in both VMIN and WS, the eviction decision reduces to a single time threshold γ.

In a dynamic-language runtime system, we can construct a similar cost-benefit structure that boils down to a single hotness threshold. In this case, we construct a simple two-resource model from memory space and processor time. Each method execution consumes processor time (more if it is interpreted, and less if it is run from compiled code), each compilation also requires processor time and memory, and the storage of compiled code consumes space in memory. Ideally, the cost of each resource would reflect its scarcity, i.e., the cost of memory should increase when more of it is in use, and the cost of processor time should increase when there is high processor demand or battery is low. Tracking resource availability, however, would increase the complexity of a

[4] WS was shown that under the locality principle, the working-set policy is very nearly optimal [7].

JIT compilation policy. Therefore, in this work, we keep the model simple by fixing the costs of each resource.

3.3 The PAYJIT Policy for JIT Compilation

The fundamental question for the PAYJIT compilation policy is how many times a method should be interpreted before the runtime invests in compiling it. This is similar to the ski rental problem, where a skier does not know how many times they will ski, but must eventually decide whether to purchase their own equipment. The canonical solution for limiting the worst case is to buy skis just before you spend as much money renting them as you would have spent buying them in the first place. For example, if skis cost $10 to rent and $100 to buy, you can limit your wasted money to $90 by buying skis just before your tenth trip to the ski hill. PAYJIT takes a similar approach: it compiles a method once once the method's hotness (and size) indicate that it would have been cheaper to compile it in the first place than to interpret it as many times as has been done.

We present two equivalent formulations for PAYJIT. In the *efficiency formulation*, PAYJIT determines whether to compile a method based on a expected cost/benefit analysis of doing so. The profit formulation provides a model from which the easily tunable *threshold formulation* is derived. In the threshold formulation, PAYJIT compiles each method only once it has reached a hotness threshold determined by the size of its DEX bytecode (which is used as a proxy for its compiled size). This threshold is the number of hotness events for which the cost of interpreting the method exceeds the cost of compiling, storing, and natively executing it.

3.3.1 Optimal PAYJIT

In order to construct our cost model, we first consider an ideal case where the whole-program hotness of each method is known a priori, and given that information, all methods are either compiled before program execution or never compiled. We present a policy called *Optimal PAYJIT* that uses the least resources for a given amount of time spent executing methods, and spends the least time executing methods for a given resource usage.

Expressing the time per hotness event for an interpreted or compiled method i as t_i^{int} or t_i^{jit}, respectively; the number of hotness events for that method as h_i; the size of that method's DEX code in bytes as d_i; the abstract cost of storing the compiled code for each byte of DEX code[5] as σ; and the cost of compiling that method as γ_i, we can write the cost of executing a program from start to finish as follows:

$$C = \sum_{i \in \text{ uncompiled}} h_i t_i^{int} + \sum_{j \in \text{ compiled}} (h_j t_j^{jit} + \sigma d_j + \gamma_j),$$

Optimal PAYJIT minimizes this cost function by making the least expensive compilation choice for each method. That is, method i should be compiled if

$$h_i t_i^{jit} + \sigma d_i + \gamma_i < h_i t_i^{int}, \tag{1}$$

and not compiled otherwise. In the case where compilation is free (e.g., it is performed in the background with plenty of resources), Inequality 1 points to a particularly important kind of optimality. Whatever cache size results from the choice of which methods to compile, there is no policy that will use the same amount of space but spend less time executing methods. Likewise, whatever time Optimal PAYJIT policy spends executing methods, there is no policy that will use the same amount of time but use less space.

Not having oracular knowledge at runtime, PAYJIT instead uses past information on the methods' usage, and predictions of its post-compiled size, compilation cost, and time per hotness event to calculate the cost function. PAYJIT then takes a conservative approach to compilation; each method is compiled only once it reaches the hotness proving it would be compiled by Optimal PAYJIT.

3.3.2 Threshold Formulation

In this section we formulate a hotness threshold that is unique to each method based on the size of its DEX bytecode. As shown in Figure 2, there is a common trend between DEX bytecode size and interpreted time per hotness event, which we denote as $f(d) = t_{int}$. The correlation between DEX code size and compiled size is implicit in the variable σ. The cost of compiling is different for each method, but it is difficult to predict a priori due to inlining. Therefore, we use the constant γ as a stand-in. Finally, we denote the average speedup from compilation as $S \triangleq \frac{t_{int}}{t_{jit}}$. Rearranging Inequality 1 for h, we get the hotness threshold $T(d)$ as a function of the DEX bytecode size:

$$h > T(d) = \frac{\gamma + \sigma d}{f(d)(1 - 1/S)} \tag{2}$$

3.3.3 Two-Point Tuning

At this point, we have a compilation criterion with two free variables: the compilation cost γ, and the per-byte storage cost σ. While we could conceivably set each of them based on expected resource availability, this would require monitoring that availability, so we leave that for future work. Instead, we propose a technique we call *two-point tuning*. Where the standard JIT compilation policy is tuned with one compilation threshold, PAYJIT can be tuned with two thresholds: one for small methods (e.g., methods approaching zero bytes of DEX code), and another for large methods (e.g., methods with 1,000 bytes of DEX code). Setting $T(0) = T_0$ and $T(d_1) = T_1$, we obtain

$$\gamma = T_0 f(d_0)(1 - 1/S) \tag{3}$$

[5] A linear correlation between DEX code size and compiled code size, as is demonstrated in Figure 3, is assumed.

and

$$\sigma = \frac{T_1 f(d_1)(1 - 1/S) - T_0 f(d_0)(1 - 1/S)}{d_1}, \quad (4)$$

so that

$$T(d) = \frac{d_1 T_0 f(0) + d\,(T_1 f(d_1) - T_0 f(0))}{d_1 f(d)}. \quad (5)$$

Figure 4 shows the shape of the threshold curve we use in our experiments, where $T(0)$ =4,000 and $T(1{,}000)$ =50,000 (as described in Section 4.2). The speedup S drops out of Equation 5 because it is in both the numerator and denominator. This is a side-effect of fixing the thresholds T_0 and T_1 instead of calculating them based on resource availability. The value used for $f(0)$ is 0.65 μs, as shown in Figure 2, because this follows the trend. Two-point tuning can be used to match the resource constraints of the system. For example, in systems with limited CPU, compilation should be more expensive. It is clear in Equation 3 that raising T_0 will achieve this. Likewise, a memory constrained system would correspond with high storage cost. From Equation 4, it can be seen that raising the value of T_1 will do this.

3.4 Implementation

In order to test PAYJIT, we implemented it in ART. This section describes a workaround that was necessary to do so with minimal changes. In ART, the hotness counter for each method, and the compilation thresholds, are stored as `uint16_t`s. With the default JIT compilation policy, this is sufficient, since OSR[6] compilation occurs when the counter reaches 20,000. However, PAYJIT uses thresholds exceeding 2^{16} (as shown in Figure 4), and there are complex side effects of modifying the counter's type (because member offsets are fixed in the class that implements methods).

Instead of giving each method a different threshold, and modifying the counter and threshold data types, we scaled the frequency at which each counter was incremented. This allowed us to use a single threshold, T', for all methods.

We calculate the scaled threshold, T, from Equation 5, and each increase in hotness count is made in proportion to the method's scaled threshold. For example, if the scaled threshold is 2× the default one, and the hotness count is supposed to increase by 10, PAYJIT only increases the hotness by half of that (5). Alternatively, if the hotness count is supposed to increase by 3, since the hotness cannot be increased by 1.5, it is increased by 1 or 2 (randomly[7]). In this way, the method's hotness count only reaches T' after T hotness events (T may be either greater than or less than T'). This mechanism has the added benefit of affecting the warm and OSR thresholds in the same way, so they also do not need to be modified. E.g., if $T = 2T'$, the warm and OSR thresholds are also doubled. The warm threshold should not affect performance much, since it only indicates which methods should be compiled while the device is idle and charging, but the change in the OSR threshold does affect compilation during our experiments. All thresholds affect the code cache size, since profiling information is generated and stored in the code cache when the warm threshold is crossed, and code is generated when the other thresholds are crossed.

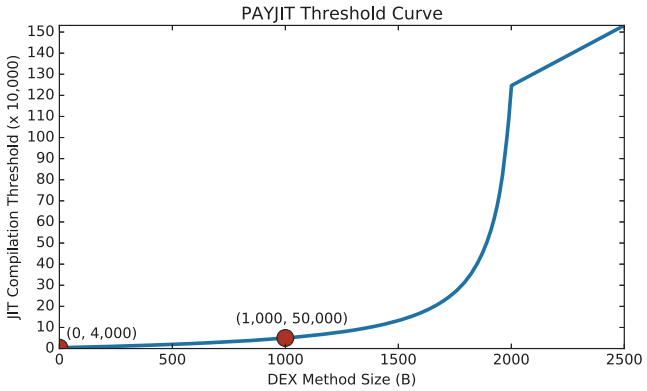

Figure 4. Weighted thresholds used in the PAYJIT policy, as calculated from DEX code size. The curve is determined by any two points; we chose a minimum compilation threshold of 4,000 (for DEX code sizes approaching zero), and a threshold of 50,000 for DEX code size of 1,000 Bytes.

4 Experiments

4.1 Workloads

We tested PAYJIT with ten different APKs: `airbnb`, `aliexpress`, `amazon`, `ebay`, `facebook`, `instagram`, `magicseaweed`, `pinterest`, `twitter`, and `wechat`, as shown in Table 1. These apps were chosen because they are all widely used, they all have user interfaces that are simple enough to automate for a short time (2-4 minutes), and they all have a significant number of JIT compiled methods (performance-sensitive apps, such as games, often ship with pre-compiled code). All but `magicseaweed` are mass-market apps. `magicseaweed` was chosen because we expected a lower-budget app to have fewer in-house optimizations and possibly less code overall.

PAYJIT is based on the correlation between DEX size and exclusive time per hotness event. We found that the relation was similar across applications, that is, all programs were similar to `airbnb`, which is shown in Figure 2. Therefore, we trained using only `airbnb` and tested on all ten applications. For all experiments, the thresholds T_0 and T_1 were pinned at

[6] OSR is one of the three hotness thresholds described in Section 2.4.

[7] The number is chosen randomly with the C standard library function `std::rand()`. An informal experiment on the Pixel shows that it has an overhead of 0.15 μs per invocation, and it is invoked approximately 22.5 million times in a 2.5 minute experiment. This adds up to a total of ~3.4 seconds spent in the `std::rand()` function. This is reasonable for our experimental implementation of PAYJIT, but a production implementation will need to use a faster implementation of `rand()`, or track method thresholds differently.

Table 1. APKs used in experiments, along with the dates they were available.

APK	Version	Date
airbnb	17.03.2	Jan. 20, 2017
aliexpress	5.3.4	June 19, 2017
amazon	12.0.1.100	May 26, 2017
ebay	5.11.1.1	June 19, 2017
facebook	100.0.0.20.70	Oct. 18, 2016
instagram	10.25.0-60813718	June 9, 2017
magicseaweed	3.3	Mar. 31, 2017
pinterest	6.23.0-623023	June 23, 2017
twitter	7.1.0-7120079	June 19, 2017
wechat	6.5.7 (1041)	April 20, 2017

4,000 and 50,000, respectively (Figure 4). These thresholds were tuned in preliminary experiments to maximize code cache space reduction with little or no slowdown.

4.2 Experimental Setup

Tests were run on a Google Pixel phone, using the APKs listed in Table 1. The Pixel contains a Qualcomm Snapdragon 821 64-bit quad-core processor, which implements the ARM big.LITTLE architecture. Two cores have frequency scaling from 0.31 GHz - 1.59 GHz, and the other two scale from 0.31 GHz - 2.15 GHz. Certain architectural details of the processors, such as cache structure, are unavailable.

The execution time is sensitive to thread-core configuration/binding and to frequency scaling. In order for our experimental results to be repeatable and reproducible, we disabled the two "little" cores and pinned the two big cores to 1.5 GHz prior to each experiment.

In order to maximize repeatability, we used the *monkeyrunner*[8] user interface automation tool, a component of Android Studio, to script interactions with each test workload [9]. In one APK (`airbnb`), it was necessary to manually log into the app prior to starting the monkeyrunner script, but in all others, the launch and running of the app were entirely automated. In order to handle experimental noise due to network and server variability, we performed a total of 16 experiments. In each, we flashed the default build of the operating system to the device, installed all 10 APKs, logged into the `airbnb` app, and initiated the series of ten monkeyrunner scripts, which were run in alphabetical order by APK name (starting with `airbnb`). Since `airbnb` needed to be logged into manually, we began profiling only after login (along with the start of the monkeyrunner script). Each monkeyrunner script launched an app, and automated user interface inputs for between 185 and 217 seconds, depending on what was possible to do in the app (the time was consistent across runs of the same app). Example user inputs include scrolling through a feed and selecting some items to open and view, and typing a message. We carried out this procedure using the default JIT compilation policy and PAYJIT, in alternating experiments.

[8]The monkeyrunner tool is designed for automated application/device testing

4.3 Profile Collection

Method profiles were collected with Android Studio's *activity manager*, using the sampling profiler, at an interval of 1,000 μs. The sampling profiler pauses the virtual machine at each sampling interval, and analyzes the stack trace, obtaining an estimate of the amount of time spent in each method (excluding time spent in children of that method). We only count the *thread time*, or the time during which the method's thread was scheduled by the Java run-time environment (this is akin to "user time"). We do not measure app responsiveness. Additionally, we instrumented the Android build to log both the pre-compiled (DEX code) and compiled (jitted code) sizes of each compiled method, and the compilation time for each method (including its inlined methods).

PAYJIT was trained using only the profile of `airbnb`, although the profile of all applications was measured and examined in the study, as mentioned in Section 4.1.

4.4 Sources of Noise

There were several unavoidable sources of noise in our experiments. These include variability in quality of service from the WiFi network, as well as from the servers the apps were communicating with. Additionally, the data served differed from one run to the next. For example, `facebook`'s news feed had different posts from one experiment to the next; many posts contained images, and some contained videos that played automatically. Since it was impossible to control the data being served to any of the apps, we performed 16 experiments and took averages (Figures 6 and 7 and sections 4.5 and 4.5 show 95% confidence error bars).

4.5 Results

In this section, we show the results of 16 pairs of experiments[9]. Figure 5 shows histograms of the percent of methods compiled by the PAYJIT policy which occupy between 0 and 1 kB, 1 and 2 kB, etc., as well as the sum of the sizes of those methods (like Figure 1). The figure demonstrates that PAYJIT eliminates the high-cost compilation and storage of larger methods; the largest compiled method was 12.0 kB (as compared to 52.4 kB with the default policy).

[9]There were actually more than 16; in several instances the total time for a single APK was significantly (~10×) less than the average. This occurred in runs using both the default and PAYJIT policies, and for multiple APKs. This was due to unavoidable factors external to the compilation policy (e.g., content not loading due to poor QoS, and unexpected pop-up windows preventing monkeyrunner from interacting with the app as scripted). These data points are not relevant to the research question. We omit every experiment in which this occurred for any APK.

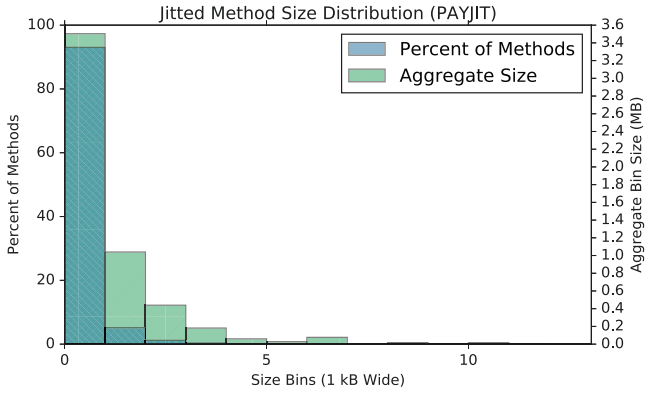

Figure 5. A histogram showing the compiled sizes of JIT compiled methods, on average, under the PAYJIT compilation policy. Blue bars (lower bars if rendered in black and white) show the percent of compiled methods in each size class, and green bars show the total size of all methods in each class. For example, in the average run, ~95% of compiled methods were under 1 kB, and their sizes summed to about 3.5 MB.

Figure 6 compares final code cache sizes between the default policy and PAYJIT. Code and data are shown separately; between the two, and across all APKs, PAYJIT reduces the total code cache size by an average of 2.02 MB, or 17.0%.

Figure 6a shows the average quantity of code in each APK's code cache at the end of an experiment. For every APK except `twitter`, PAYJIT decreases the average code size. Across all experiments and all programs, the code size is decreased by an average of 84.3 kB (13.9%). The maximum average decrease is in `facebook`, where PAYJIT reduces the code cache size by an average of 307.3 kB. The maximum average *percentage* decrease occurs in `wechat`, in which PAYJIT reduces code cache size by an average of 37.5%.

Figure 6b shows the results for data (profiling info and stack maps). Average data size is decreased for all APKs. Across all experiments and programs, the data is decreased by 122.4 kB (20.1%) on average. Both the maximum total decrease and the maximum percentage decrease occur in `wechat`, whose code cache data footprint is decreased by an average of 377.6 kB, or 44.2%.

Figure 7 shows the average total size of all code-cache garbage collections that occurred during the execution of each APK. In 6 out of 10 cases, the quantity of compiled code collected is decreased with PAYJIT, and in 7 of them, data collection is decreased. Over all experiments, PAYJIT reduces code collection from the code cache (Figure 7a) by 402.5 kB, and data collection (Figure 7b) by 1.09 MB. The quantity of code and data collected is important because this represents memory space that is in use at some point during the program run, though not at the end. Additionally, garbage collection of code indicates that the code was only useful for a limited time, not for the duration of the program.

Table 2. Improvements due to PAYJIT over the default JIT compilation policy. The bottom row shows aggregate savings for code cache and build time reduction, and geometric mean for thread time speedup. APK names are abbreviated.

apk	Code Cache Reduction (%)			Code Cache GC Reduction (%)			Thread Time Speedup	Compile Time Reduction (%)
	Code	Data	Total	Code	Data	Total		
air	4.3	10.8	7.7	20.1	7.3	12.1	1.016	6.6
ali	6.1	9.0	7.6	10.0	11.4	10.9	0.950	7.7
azn	2.5	7.2	4.9	-14.0	-11.2	-12.3	1.001	-0.4
eba	7.9	11.1	9.5	-56.4	-19.0	-31.8	0.993	5.9
fbk	23.8	27.9	25.9	64.2	66.7	66.0	1.070	49.2
ins	27.7	32.4	30.2	24.7	38.8	34.8	1.023	42.5
msw	13.6	14.2	13.9	-71.1	2.3	-11.6	1.122	9.0
pin	7.8	17.1	12.1	14.8	13.0	13.6	1.001	18.0
twi	-0.2	6.8	3.0	-12.3	-8.5	-9.9	0.995	3.0
wct	37.5	44.2	41.1	41.5	56.7	51.6	0.984	45.7
agg	13.9	20.1	17.0	30.1	39.7	36.6	1.015	26.9

Figure 8a shows the amount of user time spent executing the main process for each APK (not including spawned processes such as services), per minute of execution[10]. Only in one case (`facebook`) do the error bars not overlap, indicating a high degree of certainty that there is speedup. In all other cases, there is insufficient difference to conclude there is speedup or slowdown (though there are several where speedup or slowdown appear likely). Across all APKs, the geometric mean speedup is 1.46%.

Figure 8b illustrates the total CPU time of the compiler thread. Comparing these numbers to those shown in Figure 6 demonstrates that there is some correlation between total compilation time and total compilation size, even though it is difficult to predict an individual method's compilation time from its size (due to inlining). `airbnb` appears to be an exception in this trend, but that is because some code (~250 kB) had already been compiled prior to the start of profiling.

Table 2 shows the code cache reductions, speedups, and compilation time reductions in Figures 6 to 8.

5 Prior Work

5.1 Code Caches

Robinson et al. [22] propose, simulate, and implement several code cache replacement policies in the Java HotSpot VM using profiling and online heuristics to predict cold methods and remove them when space is needed. Zhang and Krintz [27] propose periodically marking JIT compiled methods for eviction if they have not been used in the last profiling interval. Hazelwood and Smith [13] propose a generational

[10] We display thread time per minute of execution so that comparison between apps is meaningful.

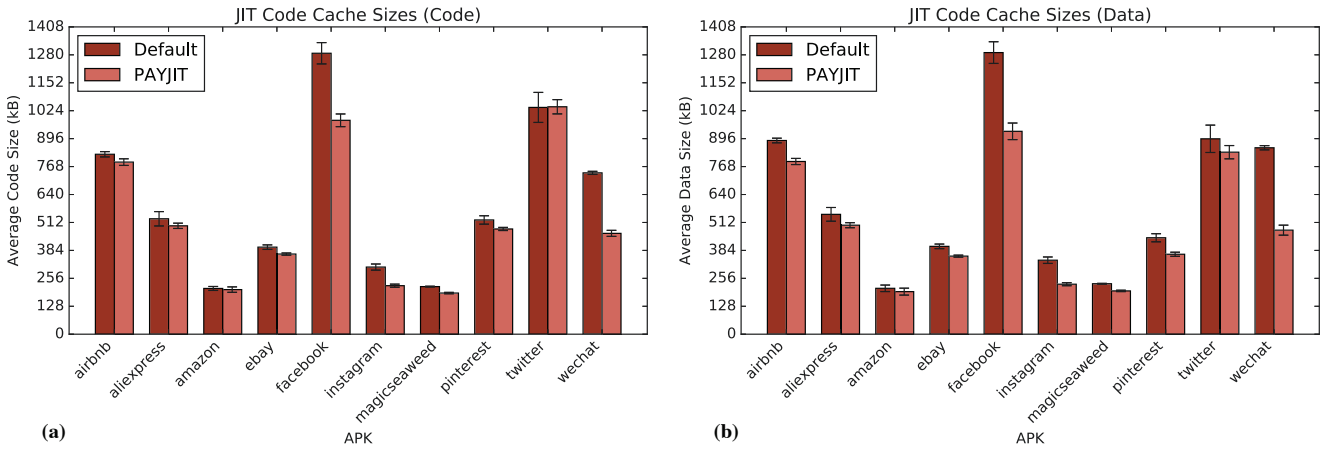

Figure 6. Average end-of-experiment code (a) and data (b) in code cache for each APK over all experiments using the default and PAYJIT compilation policies. Error bars show 95% confidence intervals.

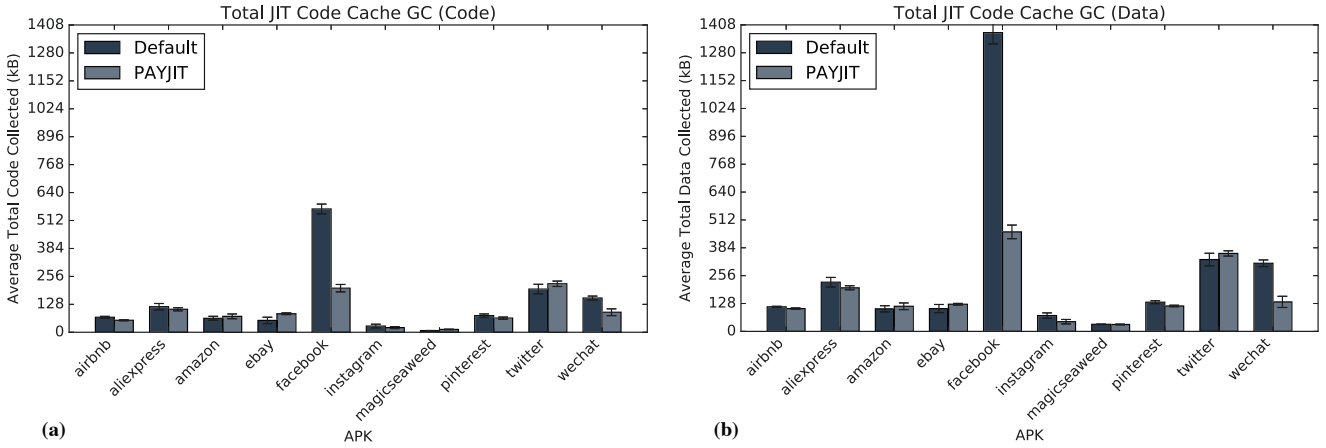

Figure 7. Average total size of collected code (a) and data (b) for each APK over all experiments using both compilation policies. This indicates memory that was in use during the experiment, but not at the end. Error bars show 95% confidence intervals.

approach to code caching, with three FIFO queues (nursery, probation, and persistent) storing "superblocks". Hartmann et al. [12] propose a *segmented code cache* for dynamic multi-tiered compilation systems. The cache is divided into three parts: code that is known to last the entire lifetime of the runtime environment, shorter-lifetime code that is inexpensive to generate, and highly optimized code, which is typically larger and more expensive to generate. Dividing the code cache in this way reduces fragmentation from recompilation, and cache sweep (GC) times.

5.2 JIT Compilation Policy

Several groups have studied tiered JIT compilers, which compile many methods with low optimization (e.g., at a low threshold), and then recompile the ones that become the hottest with high optimization. This technique can achieve both fast startup and high steady state performance [2, 15, 20, 26]. Arnold et al. [2] makes recompilation (at higher optimization level) decisions based on compilation cost and expected speedup, but ignoring code-cache impact. Kulkarni [18] tested the effects of various compilation policy parameters on the HotSpot JVM, in single-core and multi-core machines, including level of compiler multithreading, compilation threshold values, and compiling from a priority queue. This study finds that the choice of compilation threshold had a significant effect on both program execution time and compilation time. Jantz and Kulkarni [16] extend this work, testing the performance of 57 benchmarks in the HotSpot JVM with varying compilation thresholds and numbers of compiler threads, and including tiered compilation. They find

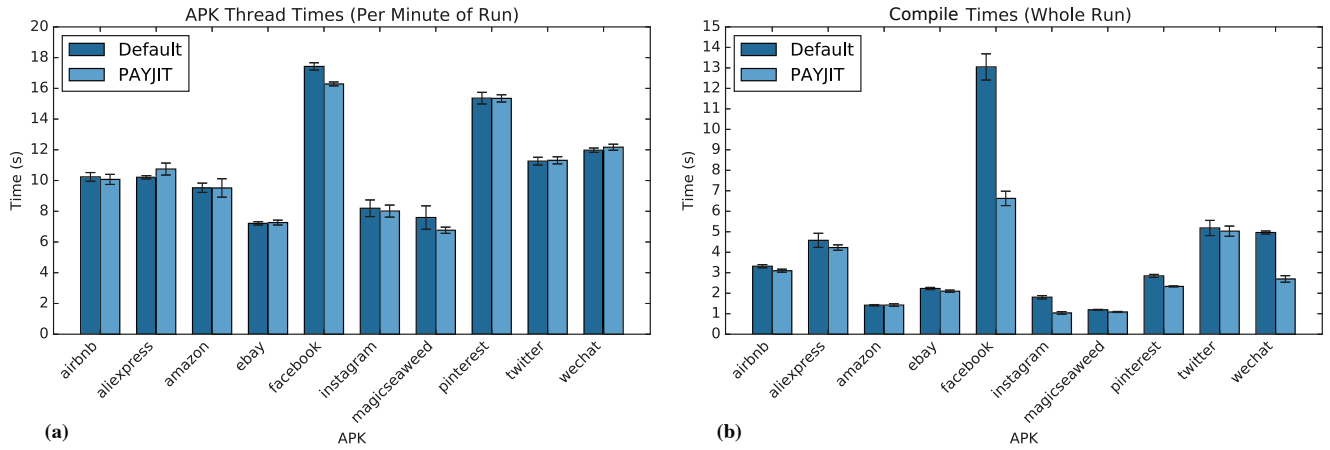

Figure 8. Average thread time per minute of execution (a), and average total CPU time of the compiler thread (b), for each APK over all experiments using the default and PAYJIT compilation policies. Error bars show 95% confidence intervals.

that the right degree of compilation aggressiveness depends on both the program and the available resources. Several publications also address compilation order [5, 8, 18, 24]. While PAYJIT uses method size to predict the space-utility of compiling a method, Fu et al. [8] use method size and loop count to predict compilation cost (*absent inlining*), which is then used to prioritize methods in the HotSpot server JIT compiler's compile queue. Kulkarni et al. [17] propose guaranteeing a minimum utilization to the compilation thread.

A number of JIT policies are adaptive. Arnold et al. [3] use a cross-run repository to select the best policy for each input. A similar approach is taken by Mao and Shen [19], who adopt more sophisticated learning methods with confidence measures to guard against misuse. A cross-run repository would be useful for PAYJIT if the application behavior, i.e., the time per hotness event (show for example in Figure 2), changes based on the program input. Since the time pattern is based on method size, not method identity, it is possible to make cross-application predictions. We have found that the pattern does not vary significantly for our workloads (see Section 4.1). Previous work has studied desktop and server applications, where PAYJIT may benefit from cross-run adaptation especially when PAYJIT could be run with more computing and memory resources than are available on a mobile device. Gu and Verbrugge [11] adjust the compilation policy based on program phase. We have not examined how the time-per-hotness-event pattern changes during an execution. This is complementary to PAYJIT and may be used in conjunction with it, especially if phase tracking can be done efficiently on a mobile device. Agosta et al. [1] propose heuristics for identifying hot code statically with byte code analysis, and dynamically with profiling.

Targeting performance, but not code cache size, Schilling [23] identifies a weak positive correlation between method size and interpreted execution time in SPEC JVM98 benchmarks, and advocates compiling larger methods in case they are extraordinarily time consuming. PAYJIT, in contrast, is based on an observed negative trend between method size and interpreted execution time *per hotness event*, and targets code cache size. The difference in the observed trends may be due to differences between the workloads, the metrics (tight loops in larger methods causing hotness count to increase out of proportion with execution time), or both. To our knowledge, PAYJIT is the first compilation policy to formally optimize the code cache size vs. performance trade-off.

6 Conclusions and Future Work

This work identified a shortcoming in the standard compilation policy in method-based JIT compilers; the hotness of a method is not a good proxy for the value of compiling it. We proposed the PAYJIT compilation policy to preferentially compile smaller methods, because per-byte their compilation contributes more to the speedup of the program as a whole. By identifying the trend between DEX bytecode size and time spent in methods for just one APK (`airbnb`), we were able to decrease code in the code cache by an average of 13.1% (for a total savings of 842.8 kB of code), and data by an average of 18.1% (totaling 1.2 MB). Additionally, PAYJIT cut compilation time by 18.7% and main-thread times by 1.3% on average. Future work can sample method time per hotness and resource availability online to determine thresholds heuristically. This may be done either for the purpose of hitting a resource usage target (e.g., code cache size), or a performance target (e.g. decreasing thresholds for more aggressive compilation). Cost models for systems with multiple agents (e.g. users, processes) and multiple resources can provide decision making tools for resource allocation.

References

[1] G. Agosta, S. Crespi Reghizzi, P. Palumbo, and M. Sykora. 2006. Selective Compilation via Fast Code Analysis and Bytecode Tracing. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC '06)*. ACM, New York, NY, USA, 906–911. https://doi.org/10.1145/1141277.1141488

[2] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. 2000. Adaptive Optimization in the JalapeÑO JVM. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*. ACM, New York, NY, USA, 47–65. https://doi.org/10.1145/353171.353175

[3] Matthew Arnold, Adam Welc, and V. T. Rajan. 2005. Improving virtual machine performance using a cross-run profile repository. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 297–311. https://doi.org/10.1145/1094811.1094835

[4] John Callaham. 2017. Android Go: Everything you need to know. (2017). http://www.androidauthority.com/android-go-773037

[5] Simone Campanoni, Martino Sykora, Giovanni Agosta, and Stefano Crespi Reghizzi. 2009. Dynamic Look Ahead Compilation: A Technique to Hide JIT Compilation Latencies in Multicore Environment. In *Proceedings of the 18th International Conference on Compiler Construction: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009 (CC '09)*. Springer-Verlag, Berlin, Heidelberg, 220–235. https://doi.org/10.1007/978-3-642-00722-4_16

[6] Peter J. Denning. 1968. The working set model for program behaviour. *Commun. ACM* 11, 5 (1968), 323–333.

[7] Peter J. Denning. 1970. Virtual Memory. *Comput. Surveys* 2, 3 (1970), 153–189. https://doi.org/10.1145/356571.356573

[8] Jie Fu, Guojie Jin, Longbing Zhang, and Jian Wang. 2016. CAOS: Combined Analysis with Online Sifting for Dynamic Compilation Systems. In *Proceedings of the ACM International Conference on Computing Frontiers (CF '16)*. ACM, New York, NY, USA, 110–118. https://doi.org/10.1145/2903150.2903151

[9] Google. 2017. monkeyrunner | Android Studio. (2017). developer.android.com/studio/test/monkeyrunner

[10] Nikola Grcevski, Allan Kielstra, Kevin Stoodley, Mark G Stoodley, and Vijay Sundaresan. 2004. Java Just-in-Time Compiler and Virtual Machine Improvements for Server and Middleware Applications.. In *Proceedings of the USENIX Virtual Machine Research and Technology Symposium (VM '04)*. 151–162. https://www.usenix.org/legacy/publications/library/proceedings/vm04/tech/full_papers/grcevski/grcevski_html/

[11] Dayong Gu and Clark Verbrugge. 2008. Phase-based adaptive recompilation in a JVM. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '08)*. 24–34. https://doi.org/10.1145/1356058.1356062

[12] Tobias Hartmann, Albert Noll, and Thomas Gross. 2014. Efficient Code Management for Dynamic Multi-tiered Compilation Systems. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '14)*. ACM, New York, NY, USA, 51–62. https://doi.org/10.1145/2647508.2647513

[13] Kim Hazelwood and Michael D. Smith. 2006. Managing Bounded Code Caches in Dynamic Binary Optimization Systems. *ACM Trans. Archit. Code Optim.* 3, 3 (Sept. 2006), 263–294. https://doi.org/10.1145/1162690.1162692

[14] Hiroshi Inoue, Hiroshige Hayashizaki, Peng Wu, and Toshio Nakatani. 2011. 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 (CGO '11)*. IEEE Computer Society, Washington, DC, USA, 246–256. http://dl.acm.org/citation.cfm?id=2190025.2190071

[15] Hiroshi Inoue, Hiroshige Hayashizaki, Peng Wu, and Toshio Nakatani. 2012. Adaptive Multi-level Compilation in a Trace-based Java JIT Compiler. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. ACM, New York, NY, USA, 179–194. https://doi.org/10.1145/2384616.2384630

[16] Michael R. Jantz and Prasad A. Kulkarni. 2013. Exploring Single and Multilevel JIT Compilation Policy for Modern Machines 1. *ACM Trans. Archit. Code Optim.* 10, 4, Article 22 (Dec. 2013), 29 pages. https://doi.org/10.1145/2541228.2541229

[17] Prasad Kulkarni, Matthew Arnold, and Michael Hind. 2007. Dynamic Compilation: The Benefits of Early Investing. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE '07)*. ACM, New York, NY, USA, 94–104. https://doi.org/10.1145/1254810.1254824

[18] Prasad A. Kulkarni. 2011. JIT Compilation Policy for Modern Machines. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. ACM, New York, NY, USA, 773–788. https://doi.org/10.1145/2048066.2048126

[19] Feng Mao and Xipeng Shen. 2009. Cross-Input Learning and Discriminative Prediction in Evolvable Virtual Machines. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '09)*. 92–101.

[20] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java hotspotTM Server Compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1 (JVM'01)*. USENIX Association, Berkeley, CA, USA, 1–1. http://dl.acm.org/citation.cfm?id=1267847.1267848

[21] Barton G. Prieve and R. S. Fabry. 1976. VMIN–An Optimal Variable-space Page Replacement Algorithm. *Commun. ACM* 19, 5 (May 1976), 295–297. https://doi.org/10.1145/360051.360231

[22] Forrest J. Robinson, Michael R. Jantz, and Prasad A. Kulkarni. 2016. Code Cache Management in Managed Language VMs to Reduce Memory Consumption for Embedded Systems. In *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems (LCTES 2016)*. ACM, New York, NY, USA, 11–20. https://doi.org/10.1145/2907950.2907958

[23] Jonathan L. Schilling. 2003. The Simplest Heuristics May Be the Best in Java JIT Compilers. *SIGPLAN Not.* 38, 2 (Feb. 2003), 36–46. https://doi.org/10.1145/772970.772975

[24] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, and Thomas Würthinger. 2012. Compilation Queuing and Graph Caching for Dynamic Compilers. In *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages (VMIL '12)*. ACM, New York, NY, USA, 49–58. https://doi.org/10.1145/2414740.2414750

[25] Statista. 2017. Mobile OS Market Share 2017. (2017). https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems

[26] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. 2001. A Dynamic Optimization Framework for a Java Just-in-time Compiler. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*. ACM, New York, NY, USA, 180–195. https://doi.org/10.1145/504282.504296

[27] Lingli Zhang and Chandra Krintz. 2004. Profile-driven Code Unloading for Resource-constrained JVMs. In *Proceedings of the 3rd International Symposium on Principles and Practice of Programming in Java (PPPJ '04)*. Trinity College Dublin, 83–90. http://dl.acm.org/citation.cfm?id=1071565.1071581