ShareJIT: JIT Code Cache Sharing across Processes and Its Practical Implementation

XIAORAN XU, Rice University, USA
KEITH COOPER, Rice University, USA
JACOB BROCK, University of Rochester, USA
YAN ZHANG, Futurewei Technologies, USA
HANDONG YE, Futurewei Technologies, USA

Just-in-time (JIT) compilation coupled with code caching are widely used to improve performance in dynamic programming language implementations. These code caches, along with the associated profiling data for the hot code, however, consume significant amounts of memory. Furthermore, they incur extra JIT compilation time for their creation. On Android, the current standard JIT compiler and its code caches are not shared among processes—that is, the runtime system maintains a private code cache, and its associated data, for each runtime process. However, applications running on the same platform tend to share multiple libraries in common. Sharing cached code across multiple applications and multiple processes can lead to a reduction in memory use. It can directly reduce compile time. It can also reduce the cumulative amount of time spent interpreting code. All three of these effects can improve actual runtime performance.

In this paper, we describe ShareJIT, a global code cache for JITs that can share code across multiple applications and multiple processes. We implemented ShareJIT in the context of the Android Runtime (ART), a widely used, state-of-the-art system. To increase sharing, our implementation constrains the amount of context that the JIT compiler can use to optimize the code. This exposes a fundamental tradeoff: increased specialization to a single process' context decreases the extent to which the compiled code can be shared. In ShareJIT, we limit some optimization to increase shareability. To evaluate the ShareJIT, we tested 8 popular Android apps in a total of 30 experiments. ShareJIT improved overall performance by 9% on average, while decreasing memory consumption by 16% on average and JIT compilation time by 37% on average.

CCS Concepts: • **Software and its engineering** → **Just-in-time compilers**; **Dynamic compilers**; **Runtime environments**; *Source code generation*; • **Computer systems organization** → *Embedded systems*;

Additional Key Words and Phrases: JIT Compilation, Code Cache Sharing, Android Runtime System

ACM Reference Format:
Xiaoran Xu, Keith Cooper, Jacob Brock, Yan Zhang, and Handong Ye. 2018. ShareJIT: JIT Code Cache Sharing across Processes and Its Practical Implementation. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 124 (November 2018), 23 pages. https://doi.org/10.1145/3276494

1 INTRODUCTION

Runtime systems, execution engines and emulators for dynamic languages typically employ Just-in-Time (JIT) compilation on frequently executed sequences, and store the resulting compiled code in *code caches* for the use of subsequent executions. This technique improves the execution speed

Authors' addresses: Xiaoran Xu, Rice University, USA, xiaoran.xu@rice.edu; Keith Cooper, Rice University, USA, keith@rice.edu; Jacob Brock, University of Rochester, USA, jbrock@cs.rochester.edu; Yan Zhang, Futurewei Technologies, USA, yan.zhang@huawei.com; Handong Ye, Futurewei Technologies, USA, ye.handong@huawei.com.

2475-1421/2018/11-ART124
https://doi.org/10.1145/3276494

of those hot sequences, but it also consumes significant amounts of memory and CPU resources to generate and store those code caches, along with the data structures to manage them. As JIT compilers have matured, they have been extended to include aggressive optimization, profiling and other features. The resulting growth in code size and associated data structures, coupled with the presence of code caches in many distinct and simultaneous processes, has created a situation where the memory occupied by code caches and the CPU cycles used to manage them can degrade the system's overall performance. Since applications running on the same platform tend to share multiple common libraries, e.g. graphics libraries in Android, user-interface and animation libraries in JavaScript, and general software development frameworks, the opportunity for sharing cached code and amortizing costs exists. However, the virtual machines (VM) that execute those applications are isolated, which forces repeated, independent compilation of shared libraries in multiple VMs, along with duplicate copies of the compiled code kept in process-private code caches.

This obvious drawback of the current software architecture motivates our work: sharing JIT code caches across applications. During the exploration of this idea, we have encountered several challenges. First of all, most JIT compilers leverage both runtime context and profile information to generate optimized code. The compiled code may be embedded with runtime-specific pointers, simplified through unique class-hierarchy analysis, or inlined recursively. Each of these "*improvements*" can decrease the "*shareability*" of JIT compiled code. Managing the trade-off between more highly optimized code and more shareable code is a serious problem. A second challenge for this work is choosing the granularity of code to share. If we choose to share at the class level, that will require changes to both the memory layout and management of the class-data area and the heap; those actions, in turn, will weaken the *portability* of the sharing system. If we choose to share at the method level, that will require a guarantee that the class data referenced by a method will be located consistently and correctly in each runtime system. Other challenges arise, including the need for new polices for updating the shared code cache and garbage-collecting it; determining if two runtime methods from different processes are functionally equivalent; and discovering the right timing to move from local interpretation to global sharing of compiled code. This paper presents a new code caching and sharing architecture, ShareJIT, which addresses all of these problems mentioned. It describes one implementation, in the Android Operating System (OS), and several open implementation options. The experiments section (Section 5) shows performance results from running ShareJIT on a collection of popular apps.

We decided to implement ShareJIT in Android due to both the ubiquity of Android devices and their resource constrained environments. It appears that Android production apps are growing in size; at the same time, the market expansion of Android devices into developing countries is creating a base of systems with smaller RAM configurations. These twin pressures make memory efficient execution of apps a pressing problem. The problem is serious enough that Google is already targeting small-memory devices (as little as 512MB to 1GB of RAM) with a separate version of Android Oreo (Go Edition). The Go Edition uses about half the memory footprint of Android N, and ships with slimmed-down versions of popular Google apps [Google 2018a; TeamAA 2018]. ShareJIT provides a partial answer to the problem of reducing memory pressure in app execution by identifying code that is used by multiple apps and allowing the apps to share JIT compiled versions of this code. This approach has the added benefit of reducing JIT warm-up time in cases where an app can use a method that has already been compiled by another app.

Contributions: The main contributions of this paper are,

- A design of a global JIT code cache that shares across different processes, with minimal modifications to the VM layer memory layout. ShareJIT was designed to port easily to other versions of the runtime system.

- A set of policies to update and garbage-collect the global shared code cache, which coordinates and orchestrates all participating processes.
- A cost model that analyzes the performance-critical parameters in sharing compiled code among inter-process methods.
- A detailed implementation in Android 7 and evaluation on 8 widely used mobile applications that have billions of downloads. ShareJIT achieves an average of 16% reduction in memory usage and an average of 9% speedup in overall performance.

The rest of this paper is organized as follows. Section 2 explains the background knowledge of Android Runtime system which is integral to the implementation of ShareJIT. Section 3 describes the architecture and implementation of ShareJIT, including its key components, workflow, as well as garbage collection policies for the global shared cache. Section 4 builds a cost model for ShareJIT to analyze the performance-critical parameters, and examines how to set thresholds to most effectively share compiled code. Section 5 demonstrates our experiments and discusses the results. Section 6 outlines related work and Section 7 offers conclusions.

2 THE ANDROID RUNTIME

The Android Runtime (ART) system was introduced in Android 5 (Lollipop) as a new execution model for application code. Since then, the internal structure of ART has evolved. When we started the ShareJIT project, Android 7 (Nougat) was the newest version, so we used it to implement ShareJIT. In the rest of this paper, we use "ART" to refer to the runtime in *Android 7.1.1_r26*, a device-specific version of Android Nougat for the Google Pixel Phone. The following sub-sections briefly introduce background knowledge of ART to help demonstrate the implementation of ShareJIT in Section 3.

2.1 Zygote Process

As mentioned in Section 1, a large fraction of Android devices suffer from the tightening resource constraints. On such devices, launching an app can cause noticeable delays. To address this problem, ART provides a specialized process—*zygote*, that spawns all other app processes. Zygote starts execution during the Android OS startup; it is initialized by preloading all the runtime Java classes and other shared resources that make up Android's rich frameworks. Because all other app processes start as a fork from zygote, they inherit zygote's memory and resources. We may think of zygote as a "warmed-up" process that speeds the startup of every other app's virtual machine. ShareJIT makes critical use of zygote's implicit resource sharing; it creates data structures for the global JIT cache in the zygote that are therefore shared to all other app processes. (See details in Section 3.1.)

2.2 JIT Compiler and Cache

Brock et al. [2018] provide a description of the JIT compilation policy and the code cache structure in Android N. As background, we summarize that here. First, ART compiles at method granularity instead of trace granularity (a trace is any series of instructions that may span multiple methods). The advantage of this is a simpler compiler implementation, and reducing overhead for managing compiled code to achieve better performance [Inoue et al. 2011]. The method-granularity compilation in ART makes sharing of compiled code feasible.

There are three separate hotness thresholds used for JIT compilation. After 5,000 method invocations or loop iterations, the method is "warm" and ART begins profiling the method and schedules it for ahead-of-time compilation when the device is idle and charging. After 10,000, the method is "hot" queued for JIT compilation. Finally, if a method's hotness reaches 20,000, it is likely to be in a hot loop. Thus, the method is scheduled with high priority for on-stack replacement (OSR)

compilation. The code produced by OSR is invoked by some specialized bytecode, which is typically a loop-closing branch. (The branch is replaced by code that invokes the OSR-compiled method.) Once the OSR compilation is completed, the interpreter will jump to the OSR-compiled code when it next crosses the specialized entry bytecode in the middle of the method.

ShareJIT does not share OSR-compiled code, because of the way that code is invoked. Instead, ShareJIT limits sharing to code compiled at the "hot" threshold (10,000); that code presents a standard method-invocation interface.

The JIT cache[1] initially allocates separate but adjacent 32 kB areas for code and data (stack maps and profile information for methods). Whenever either space becomes full, garbage collection is triggered. Garbage collection occurs in two modes, partial collection and full collection, and they are performed alternately. Partial collection removes non-entrant code and increases both code and data cache capacity equally. Full collection additionally removes the profile information of methods which are warm but not yet hot, along with code that has not been executed since the last partial collection. If the current JIT cache capacity is already at the maximum capacity (64 MB) and the collection could not free any space, the compiled code is discarded.

2.3 From Dex Code to Machine Instructions

Android applications are often compiled from Java bytecode or directly from Java to ART's dedicated register-based *dex* bytecode format [Google 2018b]. The dex file format [Google 2018c] (`.dex`) is structured differently from Java bytecode files (`.class`). A `.class` file only contains one Java class. During runtime, the JVM will dynamically load the bytecode for each class from its corresponding `.class` file. By contrast, a `.dex` file may contain all the classes of an Android app. The constant pool is a per-class data structure existing in each Java `.class` file, but in a `.dex` file it is a global data structure shared by all classes. Thus, each symbolic reference in a `.dex` file is unique; if multiple classes reference the same constant, the `.dex` file will have a single copy of that value. This is space-efficient but also has the side effect of limiting the number of references to 2^{16}, since the global reference index is represented as an unsigned 16-bit integer. So from Android 5, ART started to support multiple `.dex` files for apps whose references are more than 2^{16} [Google 2018d].

The global namespace of symbolic references in the dex file format leads to a different **symbolic reference resolution** procedure in ART compared with those in standard JVM. Symbolic reference resolution is an optional phase of class linking in dynamic language execution. It is a process for a dynamic language's virtual machine to locate classes, interfaces, fields, and methods referenced symbolically from a type's constant pool, and replace those symbolic references with direct references. Resolution only touches a type's constant pool entries but not the method bytecodes that reference them. But ART implements resolution as, (1) locating a global symbolic index referenced by a dex method when it's invoked; (2) storing the resolution result in a local, per-class data structure called the "*dex cache*", which is similar to constant pool, but only contains resolved references; (3) replacing the global symbolic index used in the dex method bytecode with the local index in the dex cache. (Note that the runtime representation of class data in the JVM is immutable.)

Whether it is a global symbolic index or a local dex cache index, the ART JIT compiler will encode the index itself, but not the direct address it references, into the compiled code when generating machine instructions as long as there's no optimization associated to this index. (See details in section 3.3) When the compiled code is executed as a frame in the virtual machine stack, a reference to the runtime constant pool (dex cache) of the class of the current method is maintained, so that the real direct address can be located. Unresolved references will trigger a resolution request that

[1] When we refer to "JIT cache", we intend to both the JIT code cache and the JIT data cache, together.

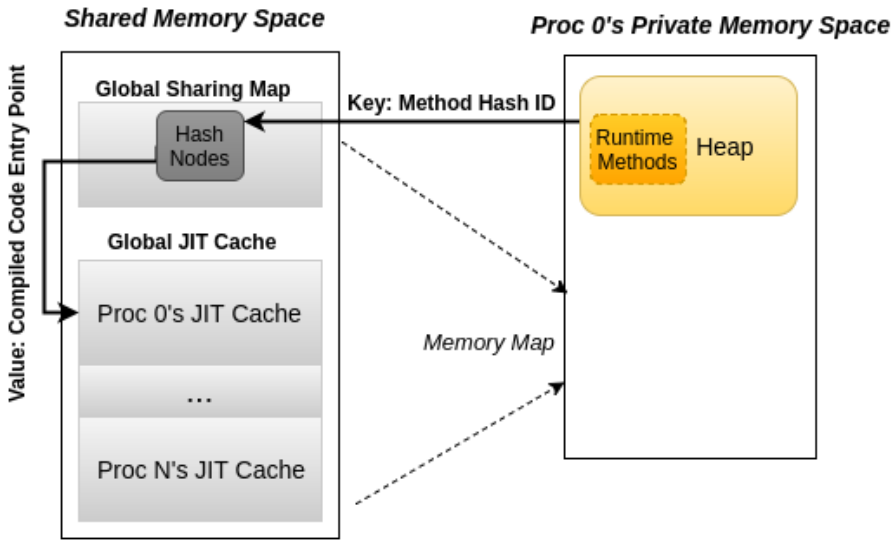

Fig. 1. The layout of the memory areas of ShareJIT

will be handled by the runtime system. Under such code generation rules, ShareJIT only needs to guarantee that all the symbolic references that two dex methods use at runtime are the same when sharing compiled code from one method to another, while it does not care about the absolute direct addresses they reference in each virtual machine.

3 THE DESIGN AND IMPLEMENTATION OF SHAREJIT

3.1 ShareJIT Internals

ShareJIT consists of two key components: a global JIT cache and a global sharing map. Figure 1 shows the layout of the memory areas of ShareJIT. The left side of the figure shows the shared memory space and the right side shows an example process's private memory space. Shared memories across processes are implemented as regions of Android Shared Memory (Ashmem), a component provided by Android OS to facilitate memory sharing. It is supported by the POSIX shared memory (Shmem) API in the kernel but wrapped with features to alleviate the low-memory pressure on Android devices. An Ashmem region is simply a memory segment backed by a file/device-driver, which can be mapped into the virtual address space of any process that has the file descriptor of that Ashmem region.

ShareJIT uses the zygote process to create the shared memory space it needs during the zygote's startup (i.e. *file_descriptor = ashmem_create_region(name, size)*). Every app process inherits the resulting file descriptor when it is forked from zygote. Then the shared memory is mapped into the app process' own virtual address space when that process' virtual machine is initialized (i.e. *shared_memory_address = mmap (file_descriptor, size, MAP_SHARED,...)*). Note that since (1) all the app processes inherit the zygote's address-space layout, (2) they all use the same startup sequence, and (3) that process is deterministic, the shared memory area is mapped at the same virtual address by each participating virtual machine. This ensures that pointers to shared memory are valid across all processes. (Even in runtime systems which do not guarantee that the shared area is always mapped to the same virtual address across processes, ShareJIT can still locate objects inside the

shared area through offsets between the base address of the shared area and the absolute address of those objects.)

The Global JIT Cache: The global JIT cache consists of consecutive, equal-size JIT cache segments owned by each app process.[2] The management and access-control of each segment is as follows.

- The owner process of a JIT cache segment has access to store compiled code in the segment. Memory allocation for the code inside the segment obeys the original mechanism in ART.
- Once compiled code is written into the cache segment, neither the owner process nor any other process may modify the code.
- The owner process may remove some compiled code during garbage collection, but only when it is not in use by any process. Memory deallocation for the code inside this segment also obeys the original mechanism in ART.
- Non-owner processes only have the right to read and execute the code inside the segment.

By assigning each cache segment an owner and restricting other processes' access rights, ShareJIT carefully avoids the potential of read-write and write-write conflicts within the code cache; it also reduces the risk of memory attacks from security exploits that have low-level access to memory.[3] By reserving the memory allocation and deallocation mechanism of the original runtime system inside each segment, ShareJIT minimizes the modification to the VM-layer memory management, leading to strong portability.

Theoretically, ShareJIT is able to declare as many cache segments in the global JIT cache as necessary, because the memory mapping facility `mmap` supports demand paging. This means the operating system copies a disk page into physical memory only if an attempt is made to access it and that page is not already in memory. Thus, the use of physical memory is determined by demand, rather than by the size of the backing file. But to be memory efficient, we set the maximum number of cache segments as 64 in our implementation. After the number of app processes reaches this maximum count, ShareJIT offers two options for a newly started process: (1) reclaim a cache segment if its owner is dead (killed by the OS, e.g., Android low-memory killer [Google 2018f], or killed by the user); (2) if no pre-occupied cache segment is freed, create traditional JIT cache in this process's own private memory space, which does not communicate with the global JIT cache. Note that we implemented a "lazy" reclamation of a dead process's cache segment; another option is to let a process broadcast its death signal and be reclaimed by ShareJIT in an "eager" style.

The Global Sharing Map: The global sharing map is the bridge from runtime methods in each virtual machine's heap to the compiled code in the global JIT cache. Whenever a method m is hot enough and JIT compiled in some process n, n is responsible for creating a hash node for m in the global sharing map. The key of the node is the *hash-identification* of m and the value is the entry

[2]The equal-size segment design simplifies parts of the ShareJIT implementation. Because the shared cache is file-backed, demand-paged, and mapped into each app's virtual address space, the cost of enlarging the per-app segment size is minimal; it only incurs a physical memory cost if the space is used. Thus, our approach is to choose a "large enough" per-app segment size and avoid the complications and costs that would come with managing different per-app segment sizes.

[3]The problem of an attacker spoofing a code-cache entry is difficult; the best solution will depend on assumptions about the rest of the system. If the JIT is corrupted, then neither code cache policies nor implementation will prevent the attacker from taking over the system. Thus, we assume that the JIT is uncorrupted and that only the JIT has write access to the code cache. In this scenario, a hash key computed over the method bytecode should be sufficient to ensure a method's validity. (The hash key will be described in the next subsection.)

If arbitrary code in the process can write into the code cache, then additional measures and additional computation would be necessary to detect a corrupted native-code segment. For example, the JIT could compute a secondary key over the native code and securely retain it in a map from hash key to secondary key. The JIT could, on demand from the prospective sharer, recompute the seconday key and compare it to the value stored in its secure map. Such a scheme could provide an added degree of detection and reassurance to the prospective sharer.

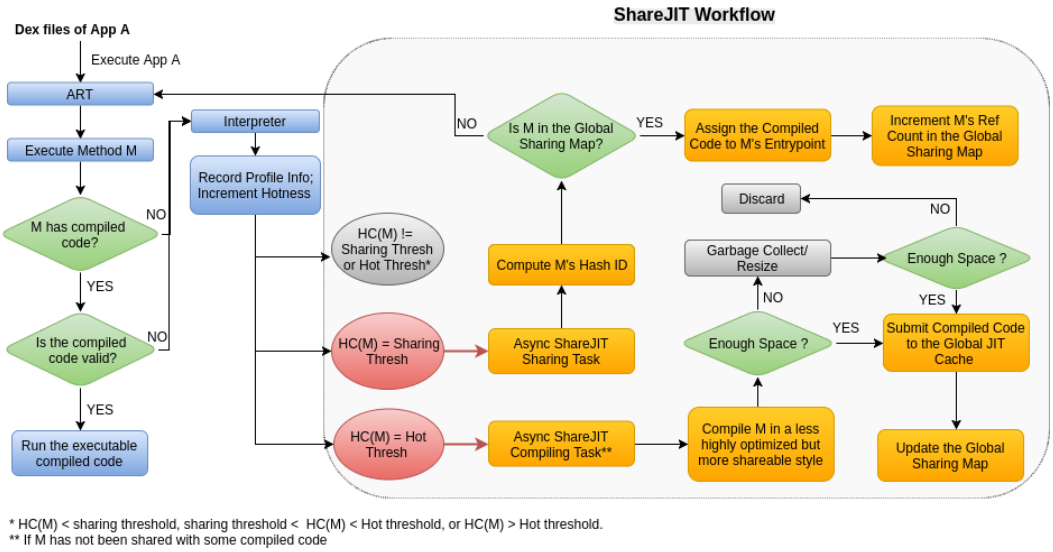

Fig. 2. The Work-flow of ShareJIT

point of the compiled code of m in process n's cache segment. We define the **hash-identification** of a method m as the hash code of m's method signature and byte-by-byte dex code.[4] Section 2.3 proves the legitimacy of identifying functionally equivalent methods with such a hash-id, whether the symbolic references used in the dex methods are statically linked or will be dynamically linked. A hash node also contains an unsigned integer reference count of the processes that are using the compiled code associated with this node. The reference count tells the garbage collector when the method's compiled code can be collected and freed. (See details in Section 3.4)

The global JIT cache is divided into contiguous segments, each with a single writer, to simplify both management and synchronization. This scheme does not work for the global sharing map. Each process that uses the global sharing map needs the right to read and write it. Since Ashmem does not have a serialization mechanism for a multitude of readers and writers in a concurrent system, we use a *semaphore* to protect the global sharing map.

3.2 ShareJIT Workflow

Figure 2 shows the main workflow of ShareJIT. An Android app's lifetime starts with being installed, being launched which will trigger zygote to fork a child process for it and initialize a runtime system inside the process, and eventually being executed by the runtime system. When a method M from the app A is invoked, ART will first check if M has corresponding compiled code as its execution entry-point. A method's entry-point could point to the JIT code cache area, the interpreter or null. (It could also point to the native code area, but we omit the scenario of native code execution in our discussion since it's irrelevant to the design and implementation of ShareJIT.) If M has compiled code, ART will execute it directly from its entry-point; if not, ART will interpret it. After the interpretation, M's hotness count will be incremented; and if M is already warm, which means

[4]We use a 128-bit hash code value in our implementation. According to our experiments, large apps such as Facebook have up to 10^6 methods in total, while small apps tend to have around 10^4 methods. Even if we have 1,000 the largest apps and 10^9 methods in one system, the hash collision possibility is as low as 10^{-21}, which is negligible.

a profile was created and associated with it, the profile will also be updated according to the execution.

Two thresholds govern the decisions in the ShareJIT workflow: the *sharing threshold* and the *hot threshold*. The **sharing threshold** determines how hot a method must be before ShareJIT will attempt to share it. The **hot threshold** determines how hot a method must be before ShareJIT will attempt to compile it.

These two thresholds divide the hotness count of some method m, HC(m), into five distinct ranges.

(1) $HC(m) < sharing\ threshold$: A value of HC in this range triggers no immediate action.

(2) $HC(m) = sharing\ threshold$: When m reaches the sharing threshold, ShareJIT will create a sharing task for it and add that task to the JIT thread's task queue. When the JIT thread reaches this task, it will (a) compute the hash-identification for m, (b) check if the global sharing map has code for this hash-identification, (c) update m's code to use the compiled code if found, as well as incrementing the reference count of the corresponding hash node.

(3) $sharing\ threshold < HC(m) < hot\ threshold$: A value of HC in this range triggers no immediate action.

(4) $HC(m) = hot\ threshold$: When m reaches the hot threshold, if m has not been shared with corresponding compiled code, ShareJIT (a) compiles the code for m in a shareable manner, (b) inserts it into the shared cache, (c) updates the local code in m to use the newly compiled code, and (d) adds a new hash node for m in the sharing map.

(5) $HC(m) > hot\ threshold$: By the time a method reaches this range with its hotness count, it should have been shared with compiled code or compiled into the shared cache. If m reaches the OSR threshold (20,000), it triggers an OSR compilation, but the resulting code is not shared by ShareJIT because of its non-standard entry provisions. We do not discuss the details of OSR mechanism since it is not in our scope.

ShareJIT uses the same hot threshold as ART's default JIT—that is, 10,000. Sections 4 and 5.4 explain how we set the sharing threshold. Section 3.3 describes some of the constraints placed on compilation of globally shared code mentioned in range (4).

3.3 The Shareability of JIT Compiled Code

JIT compilers often generate more highly optimized code than ahead-of-time (AOT) compilers do because JIT compilers have extra runtime context. But overuse of such runtime context can make the code unsharable across processes. For example, replacing symbolic references in method invocation instructions (i.e. `invoke-direct` instruction in dex bytecode) with the absolute addresses of callee methods can reduce function call overhead. (This transformation happens as part of method devirtualization driven by class-hierarchy analysis.) Unfortunately, embedding absolute addresses makes the code unsharable because the method bytecode is located in the class-data area and is likely to be at different locations in different processes. Thus, ShareJIT disables such usage of process-specific runtime context. The alternative would be to require that ShareJIT translates a private pointer in one VM to the corresponding address in another VM, which is impractical.

Another optimization that poses problems for ShareJIT is inline substitution. As discussed in Section 2.3, ShareJIT only needs to ensure the symbolic references two dex methods use are literally identical to determine that sharing compiled code between them is legal, while it can ignore the content that those symbolic methods reference. However, inlining a callee method means dereferencing the pointer in an invocation instruction, and replacing it with the actual method

code.[5] So to share code that has been inlined, ShareJIT needs to ensure that all of the code in the inlined section is identical in both "sharer" process and "sharee" process. Specifically, ShareJIT could record a list of the inlined callee methods, and check that each pair of callee method in the "sharer" process and the "sharee" process are also functionally equivalent. This mechanism would require extra space and time to compute and store the hash-identifications of callee methods; it would significantly increase the overhead of sharing, particularly when inlining is performed recursively, i.e. ART JIT compiler allows an inlined callee method to inline its own callee, up to 3-level depth.

Because of this overhead, the current ShareJIT implementation only inlines built-in methods—methods that are defined in the classes preloaded by the zygote process during its initialization. It does not inline app-specific methods—methods defined in app's `.dex` files. This restriction guarantees that if a callee is inlined, its definitions in both the "sharer" and the "sharee" runtime system are identical; they are the same implementation inherited from their common super-process: the zygote.

ShareJIT sacrifices some potential performance gain from inlining. It could inline more callee methods at the cost of a decrease in the shareability of compiled code. We discuss how this reduction in inlining affects the overall performance of ShareJIT in Section 5.5.

3.4 Garbage Collection

As described in Section 2.2, garbage collection occurs in two modes: partial collection removes non-entrant code and increases both code and data cache capacity equally; full collection additionally removes the profile information of methods which are warm but not yet hot, along with code that has not been executed since the last partial collection. Non-entrant code are the compiled code that has been discarded for various reasons, the most frequent of which is deoptimization from aggressively compiled code, e.g. bounds check elimination, dynamic type assertion, inline caching, etc. JIT compilers could try these optimizations not only because they could gather more profile information of the input method, but also because they could afford possible deoptimizations if such too-aggressive optimization attempts end up failing during the subsequent method executions–it just switches back to the interpreter and let the garbage collector free the discarded code.

Unlike those optimizations described in Section 3.3 that have to be disabled or restricted in ShareJIT for shareability, these aggressive optimizations could be reserved in ShareJIT with proper modification of the garbage collector. We will use inline caching as an example to explain how these optimizations are compiled, when the deoptimization would be triggered, and how the garbage collector should deal with the concomitant discarded compiled code in a global sharing JIT cache.

Inline caching [Deutsch and Schiffman 1984] is an optimization technique to speed up runtime method binding. It remembers the results of previous virtual method lookups "inline", i.e. directly at the call site. Those results are called "inline caches". The concept relies on the empirical observation that the objects that occur at a particular call site are often of the same type. ART collects inline caches for every virtual callee method invocation in a caller method, and stores them in the JIT data cache as "profile information" for this caller method. The JIT compiler will perform this optimization when an inline cache is monomorphic or polymorphic [Hölzle et al. 1991] during compilation, and also emit a backup "slow-path" which leads to a normal unoptimized/safe execution flow. If the "slow-path" is executed, which means the type assertion optimization fails, a concomitant deoptimization exception will be thrown to the runtime system afterwards.

[5] In the ART JIT compiler, it inlines the intermediate representation (control flow graph) and then translates the two methods together.

The handler for deoptimization exceptions invalidates and discards the offending compiled code. It resets the hotness count of that method to zero. If the method becomes hot again in the future, a new version of compiled code will be generated. If this situation occurs soon enough, i.e., before the next collection can free the older version, (this situation arises often in practice,) the original JIT would simply use the newly compiled version and let the garbage collector free the old version because the old version is discarded by itself; however, ShareJIT must decide which version to keep in the global sharing map from a global perspective.

To deal with this situation, we introduce the first rule for the ShareJIT garbage collector: *"Newer is better."* Because the newer compiled code results from richer and more precise execution information, ShareJIT always overwrites the old version in the global sharing map with the new one. To prevent "sharee" processes from executing the discarded old code, a validity check is put before any compiled code is executed as shown in Figure 2.[6] Another way to cope with discarded compiled code is to broadcast its invalidation, which allows the "sharee" processes to proactively delete that entry point. In either approach, the garbage collector can safely collect and free the old code.

We use inline caching as an example to introduce why and how the ShareJIT garbage collector should behave differently, not only because it's one of the common deoptimizations, but also because inline caches are part of the method profile information that is stored in the JIT data cache. (The runtime system starts to collect profile information for a method when it becomes "warm".) From the description of inline caches above, we could see that once a sharing relationship is built, the "sharee" method's profile becomes useless since it was intended to help in the JIT compilation but the "sharee" method will simply not be compiled.[7] So the garbage collector could free the profiles of all "sharee" methods during a full collection. Thus, a "sharee" method has no associated memory in the JIT cache at all—compiled code and stack maps are never created, while the profile is deleted. (Note that the JIT data cache consists of two parts—method profiles and stack maps, while JIT code cache only contains compiled code, so ShareJIT saves more data cache than code cache, which will be demonstrated in Section 5.5.)

The second rule for the collector is *"Only collect compiled code that is not being used by any process"*. Recall that each hash node in the global sharing map has a reference count which records the number of "sharee" processes for this node. The garbage collector will only collect and free the compiled code that has a reference count of 0. This policy might reduce the amount of code ShareJIT would collect compared with the default JIT, but it's beneficial to the performance.

Detailed pseudo code is shown in Algorithm 1. The variable *own_method_map* holds the methods whose compiled code are stored in a process's own cache segment, while *sharee_method_map* holds the methods whose compiled code is shared from other processes' cache segments. Traverse the *own_method_map*, if some compiled code c is not its corresponding method m's entry point, and the reference count of m in the global sharing map is 0, delete it in the *own_method_map*, and also delete it in the global sharing map; traverse the *sharee_method_map*, if some compiled code c is not its corresponding method m's entry point, delete it in the *sharee_method_map* and decrement the reference count of m in the global sharing map. Note that for a method m in the *sharee_method_map*, even if its reference count $RC(m)$ becomes zero after being decremented, the collector is not allowed to delete the node in the global sharing map or to free the compiled code c, because ShareJIT only grants the owner process the privilege to allocate/deallocate memories in its cache segment.

[6] Another scenario where a process might attempt to access invalid compiled code is when the process which produced the compiled code died and its cache segment was reclaimed.

[7] If a "sharee" method deoptimizes from running the compiled code and subsequently becomes warm again, which happens infrequently, the method can re-collect the profile information while it is getting hot.

ALGORITHM 1: Checking Method Entry Points in a Partial or Full Garbage Collection

```
for every pair (compiled code c, method m) in own_method_map do
    if c is not m's entry point and RC(m) == 0 then
        delete (c, m) in own_method_map;
        delete m in global_sharing_map;
        free c;
    end
end
for every pair (compiled code c, method m) in sharee_method_map do
    if c is not m's entry point then
        delete (c, m) in sharee_method_map;
        decrement RC(m) in global_sharing_map;
    end
end
```

4 UNDERSTANDING THE COSTS OF SHAREJIT ENABLED EXECUTION

ShareJIT introduces complex behavior that affects the runtime performance of apps. To understand that behavior, it helps to consider three different perspectives on how the shared global JIT cache affects performance and memory utilization, as described in the following three subsections. In each case, the performance improvement is difficult to predict or model because it depends in a detailed way on both the future behavior of each process and on the overlap between processes.

4.1 Single-Process Performance

From the perspective of a single executing app, ShareJIT has three direct effects.

(1) Once a method m has been invoked *sharing threshold* (ST) times, the process executing the app checks to see if m has corresponding compiled code in the global shared cache. Specifically, the running process computes a global hash-identification for m, and checks in the global sharing map for an instance of that hash-identification.

(2) If the running process discovers a shared, compiled copy of m, it links the runtime system's call to the shared, compiled implementation, and updates the reference count of m in the global sharing map.

(3) If the lookup fails to find a shared, compiled copy of m, the running process continues to interpret m until $HC(m)$ reaches the *hot threshold* (HT). At that point, the executing process schedules a JIT compilation of m.

Each of these actions has a cost. From the single-process perspective, the search for a shared, compiled copy of m is an added cost. But if the search finds a copy of m in the shared global cache, the cost pays off, because the running process begins to execute compiled code earlier than it would with the standard Android JIT, and additionally avoids the cost of JIT-compiling m.

Defining $\Delta T(m)$ as the reduction in execution time from running the compiled version of m rather than the interpreted version, then it should save at most $\Delta T(m)(HT - ST)$ time. Because the standard JIT would have interpreted m until $HC(m) \geq HT$ while ShareJIT started executing compiled code when $HC(m) \geq ST$. Once $HC(m)$ exceeds HT, the runtime cost of executing m *approximates* the cost that would occur with the standard Android JIT. We say it approximates the cost because ShareJIT restricts optimizations in the shared compiled code in a way that the standard JIT does not.

Thus, the factors in ShareJIT execution that may slow the app are limited:

(1) the cost of computing the global hash-identification for a method m, defined as $H(m)$,
(2) the cost of checking the global sharing map for m, defined as $L(m)$,
(3) the cost of linking the call site to the shared, compiled copy of m,
(4) the cost of updating the reference count for m in the global sharing map,
(5) and any lost efficiency from restricted optimization.

Of these costs, the first two are the most significant because they are incurred every time some method crosses the sharing threshold and does a lookup in the global sharing map. While the cost is trivial for a single method, in aggregate, it occurs many times. Items (3) and (4) only occur if that lookup succeeds: a much smaller number. The lost efficiency from restricted optimizations is a default cost—the sum of all JIT compiled methods' performance degrading, which is not correlated with the sharing attempt.

Note that in the implementation of ShareJIT in Android, the cost of computing method hash-identifications is paid at runtime. For other virtual machines in which the runtime representation of class data including bytecode is immutable, such as a standard JVM, method hash-identifications could be generated at compile time and stored in `.class` files. One drawback of this approach is that it consumes extra runtime space after these `.class` files being loaded. ShareJIT allows an implementation flexibility as to trading-off between time cost and space cost at runtime for identifying methods globally. In ART, computing method hash-identifications at runtime is the only choice as explained in Section 2.3. Besides, Android devices are already suffering from an increasing memory pressure.

On the other hand, ShareJIT has two major sources of improvement: avoiding the cost of JIT-compiling the method m and the savings from additional compiled executions up to $(HT - ST)$ times. As might be expected, the total number of executions of m in the app determine the expected improvement. Assume that m already exists in the shared code cache. If $HC(m)$ stops at $(ST + 1)$, then the impact will be negligible. As $HC(m)$ grows from ST to HT, the savings from executing compiled code will increase. Once $HC(m)$ exceeds HT, the savings from executing compiled code stop, and the only additional improvement comes from the fact that ShareJIT avoided the cost of JIT-compiling m.

4.2 System-Wide Performance

From the perspective of the overall system, ShareJIT will fundamentally change the relationship between the amount of time spent JIT-compiling code and the number of method invocations. With ShareJIT, a single compile step can improve the performance of multiple processes and multiple apps. With ShareJIT, a single app can begin executing compiled code at an earlier point in its progress than it would with the standard Android JIT—earlier by (HT - ST) method invocations.

The impact of this effect depends, heavily, on how invalidations in the global cache are handled. The benefit from cross-processes sharing depends on the compiled code remaining in the shared cache. As explained in Section 3.1, the current implementation keeps a process's image in the shared cache intact until it must be reclaimed. That is, if process p compiled and cached method m and, subsequently, exited, p's copy of the compiled code for m will remain in the shared cache until some newly spawned process needs a segment *and* p's segment is the next segment to be reclaimed. This policy maximizes the residency of shared code in the cache and increases the impact of the shared cache on overall performance. Although recompilation resulted from deoptimization also causes shared cache invalidation and update, it happens infrequently on shared code in practice, and sharing relationship can be rebuilt again with the recompiled code quickly.

To summarize the analysis above, we could build a cost model for executing a method m as: (in terms of the difference between ShareJIT and default JIT)

$$F(m) = \begin{cases} 0, & \text{if } HC(m) < ST \\ -H(m) - L(m) + S(m)\Delta T(m)(HC(m) - ST), & \text{if } ST \le HC(m) < HT \\ -H(m) - L(m) + S(m)\Delta T(m)(HT - ST) + S(m)J(m), & \text{if } HC(m) \ge HT \end{cases} \quad (1)$$

where $H(m)$, $L(m)$, $\Delta T(m)$, $HC(m)$, ST and HT have been defined before; $S(m)$ returns a binary value 1 or 0 depending on whether the lookup in the shared global cache for m succeeds or not; $J(m)$ is the JIT compilation cost for m. (To simplify, we ignore the difference in $J(m)$ between ShareJIT and default JIT.) Then $F(m)$ would be the ultimate performance improvement for m we could gain from ShareJIT, compared with default JIT.

$\Delta T(m)$, the speedup from executing the compiled code of m versus interpreting m, is computable at runtime, although we didn't implement the calculation. Given $Ti(m)$, time cost of interpreting m, and $Tc(m)$, time cost of running the current version of compiled code of m, then obviously $\Delta T(m) = Ti(m) - Tc(m)$. $Ti(m)$ and $Tc(m)$ could be recorded during the execution of m.

However, the final hotness of a method, $HC(m)$, is unknowable. We cannot precisely predict how many times a method m will be invoked in the future. We cannot predict if the compiled code will become invalid due to the recompilation of m or the reclamation of the "sharer" cache segment, either. So we are not able to analyze the performance gain from Equation 1 through mathematical modeling. Then when is the best time to share?

The similar problem motivated all the studies for JIT compilation policy in academic history—"when is the best time to JIT compile a hot method?" The mainstream of JIT compilation policy in industry is using a hotness threshold or a method invocation threshold, e.g. ART, HotSpot JVM server compiler [Paleczny et al. 2001], and IBM JIT compiler [Suganuma et al. 2000]; methods whose hotness counts or invocation counts reach the threshold will get compiled at runtime. It is both easy-to-implement and efficient. Applying that heuristic on sharing policy, we assumed a single sharing threshold ST of method hotness, at which ShareJIT attempts to share, in the workflow of ShareJIT and the analysis above. The total system performance improvement ShareJIT gains compared with default JIT are:

$$Y(ST) = \sum_{HC(m) \ge ST} F(m) = -\sum_{HC(m) \ge ST} C(m) + \sum_{HC(m) \ge ST} G(m) \quad (2)$$

where $C(m)$ is simplified for $H(m)+L(m)$, and $G(m)$ is simplified for the positive terms in Equation 1, despite of the value of $HC(m)$. We can see that the major tradeoff comes from the **warming methods**, those with $ST \le HC(m) < HT$. For these warming methods, the benefit is the sum of inevitable increased cost from checking more methods and potential increased gains from sharing methods earlier and sharing more methods, as ST decreases. The key to maximizing total benefit is to find the best sharing threshold ST for this term, given as $Y(ST)$ above. Section 5.4 describes our experiment to find this value.

4.3 System-Wide Memory Utilization

From the perspective of system-wide memory utilization, the effect of ShareJIT is conceptually simple. ShareJIT reduces system-wide memory use for compiled-code in proportion to the amount of sharing and the size of those methods. In practice, we can measure total memory use for compiled code with the standard Android JIT and with ShareJIT and subtract. To make predictions, however, is difficult because the actual savings depend on how different apps overlap in time and how the eviction policy for the shared cache works.

In practice, we measured this effect and found that ShareJIT reduced overall JIT cache use by roughly 16%. (see Section 5 for more discussion)

5 EXPERIMENTS

5.1 Benchmarks

To evaluate the performance of ShareJIT compared with the default JIT, we chose 8 widely used apps as benchmarks: *Airbnb, Amazon, Chrome, Facebook, Firefox, Instagram, Googlemaps, Skype.* The specific version and release date information are listed in table 1. They were chosen for three primary reasons:

- They are all mass-market apps in real life; (Except for *Firefox*. *Firefox* was chosen because it's in the same category with *Chrome*–they are both web browsers, and we would like to see if apps in the same category tend to share more code in common.)
- They all have easy-to-use graphical interfaces which can be automated by our scripts to imitate the real users' activities;
- They all have reasonable number of JIT events during run time. (Games are also heavily downloaded, but they are often AOT compiled to a specific ISA, and do not involve JIT compilation at all.)

5.2 Experimental Setup and Steps

The experiments were running a Google Pixel 32GB smartphone. It has a Qualcomm Snapdragon 821 64-bit quad-core processor, which implements the ARM big.LITTLE architecture. Two of the four cores have frequency scaling from 0.31 GHz - 1.59 GHz, and the other two scale from 0.31 GHz - 2.15 GHz. We measured the CPU cycles used by each process as the metric to evaluate ShareJIT's and default JIT's performance. Since this metric is sensitive to thread-core configuration and to frequency scaling, we disabled the two "little" cores and pinned the two big cores to 1.5 GHz prior to each experiment, in order to make our experimental results repeatable and reproducible.

Each app was run by an automated script, instead of a human operator. The scripts are written with Android ADB shell input commands[8], which could send events like clicking, swiping, and typing text to the device, simulating a real user's activities on a certain app. Because *Airbnb* detects and prevents robot login, we needed to input a user name and a password manually. In all other

Table 1. Apps Used in Experiments

APK	Version	Release Date
Airbnb	17.50	Dec. 16, 2017
Amazon	12.7.0.100	Aug. 28, 2017
Chrome	57.0.2987.97	March 9, 2017
Facebook	100.0.0.20.70	Oct. 18, 2016
Firefox	52.0-2015474475	March 3, 2017
Instagram	10.25.0-60813718	June 9, 2017
Googlemaps	9.61.1-961102122	Sep. 12, 2017
Skype	8.12.0.14	Dec 11, 2017

[8]See details about ADB shell input commands at www.raizlabs.com/dev/2017/09/automating-input-events-android/

apps, the launch, execution, and data collection were entirely automated. We used automated scripts to eliminate the data deviation that might be caused by human operators, and to maximize the repeatability of the experiments.

We also randomized the order of running apps in each experiment. Because in a sharing relationship, the sharer is always the app used earlier and the sharee is always the app used later, the experimental results for one app depend on the order in which all the apps run. Thus, we randomized the apps running order in each experimental run to amortize the potential result bias that might be caused by any fixed ordering of the apps.

The length of each app's running time ranged from 2 to 5 minutes, depending on the functionalities of the app and what was possible to do from the perspective of a real user. For example, the script for *Chrome* launches the app, inputs a keyword at Google search bar, then browses the items and images for 2 minutes; the script for *Facebook* launches the app, inputs a user name and a password to login, browses the news feed, and browses the market items around the user's location for a total of 5 minutes. The amount of work in the script for the same app is consistent across multiple runs.

Each experimental run was performed in the following steps:

(1) flashing the experimental device, a Google Pixel Phone, with either the default Android OS Image, or the ShareJIT Android OS Image;

(2) installing all 8 benchmark apps;

(3) running the 8 automated scripts in a random order;

(4) dumping and collecting the data.

The data produced by the experimental runs was, on a per-process basis: compilation time for all the JIT compiled methods, CPU cycles used by the app processes, JIT code cache size and data cache size which includes runtime methods profile information, and stack maps as we mentioned. The JIT compilation time, compiled code size and data size of a method are logged at runtime through the Android Logcat tool[Google 2018e]; the CPU cycles are read from the */proc/pid/stat* file, which provides status information about the process identified by *pid*. From the */proc/pid/stat* data, we extracted both user mode and kernel mode cycles consumed by each process. We believe this number, representing the time resources spent by a process, is an accurate and precise metric to compare the performance between ShareJIT and default JIT .

5.3 Data Noise

During the experiments, there existed several inevitable sources of data noise. One is some content provider apps served different content from one run to another. For example, *Facebook* and *Instagram* had different posts in each run. And it was impossible to control the content being served in any app or any run. Another source is the inconsistency of the quality/speed of the WiFi network and apps' servers. This effect stands out when there are automatically played videos in an app's content. Higher WiFi network speed or server responding speed brings shorter loading delay of a video before its automatic playing, which leads to more data transfer and communication between an app and its server. To eliminate these noises, we conducted 30 experiments (15 pairs) and took averages; we ran default JIT and ShareJIT alternately and consecutively to minimize the effect of environmental and app-content changes.

5.4 Sharing Threshold

As we mentioned in Section 4, the benefit of sharing the compiled code for a method is unpredictable because the number of future invocations of a given method is unknowable. So we performed an

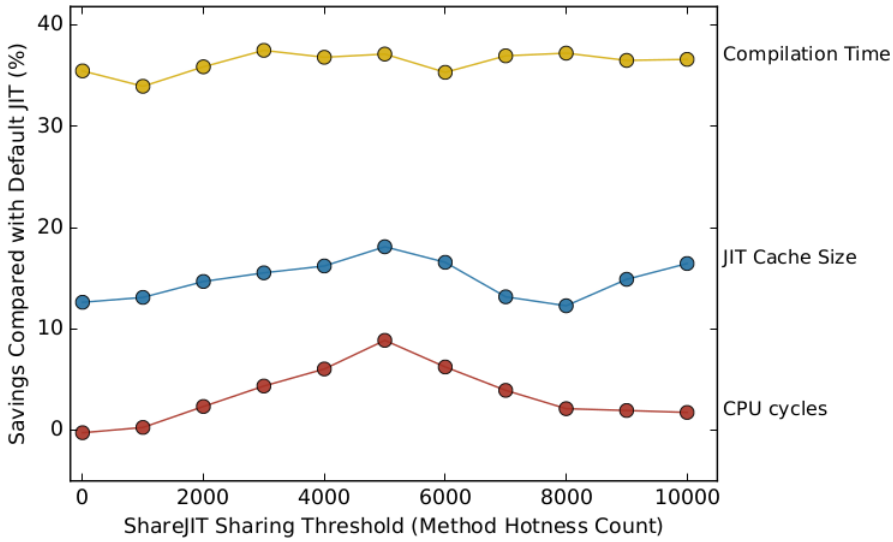

Fig. 3. Savings in CPU Cycles, Compilation Time and JIT Cache Size (including both code cache and data cache), compared with default JIT in ART, as a function of different sharing threshold values

empirical study to discover the threshold where, in practice, sharing compiled code gains the most performance improvement.

As mentioned in Section 2.2, the threshold for JIT compilation in ART is a hotness count of 10, 000 for a method. At a count of 10, 000, its own JIT compiler will compile it. Thus, the sharing threshold must be lower than 10, 000. Therefore, we divided the range from 1 to 10, 000 equally and tested the thresholds at 1, 1,000, 2,000, 3,000, ... , 10,000. For each tested threshold, we ran the experiment following the procedure described in Section 5.2. Figure 3 shows the results. The x axis shows different thresholds at which a sharing task is created for a method. The y axis shows the percentage of savings on compilation time, JIT cache size and CPU cycles, compared with default ART. The values of the data points in this figure are the average results of all 8 apps.

From this figure, we can see that the compilation time is not significantly statistically correlated with the sharing threshold; all of the compilation time savings fluctuate slightly between 35% to 37%. Both cache size and CPU cycles show a peak at a threshold of 5, 000. It appears that, for CPU cycles, as the threshold ST increases and the number of methods which qualify for $HC(m) \geq ST$ decreases, both the cost $\sum C(m)$ and the benefit $\sum G(m)$ in equation 2 decrease; but on the left of the peak, the cost decreases faster, while on the right, the benefit decreases faster.

For the JIT cache size, it is tempting to think the increased sharing leads to increased code space savings. In truth, however, final code space saving depends on the total number of methods compiled and shared. The value of ST only affects cache size if it changes the set of methods that are ultimately compiled. If a method's hotness count stays in between the sharing threshold and the hot threshold, it will not get compiled when run with either the default system or ShareJIT. If it does not get compiled in either, then ShareJIT shows no space savings. We cannot resolve this problem satisfactorily using arithmetic because the app's behavior is unpredictable. Nevertheless, the empirical study gives us an answer—the best sharing threshold appears to be 5, 000. And the

experimental results depicted in the next section are generated with the sharing threshold set to 5, 000.

5.5 Results and Discussion

The results we show in this section are the average values of all 15 pairs of experiments. If an app consists of multiple processes, e.g., *Chrome* created as many as 6 processes including the main process `com.chrome`, the sandbox processes such as `com.chrome:sandboxed_process0` and the privilege processes such as `com.chrome:privileged_process0`, etc., the results include the data from all its processes.

Table 2 lists all the detailed numbers of our experimental results. ShareJIT reduces the total JIT code cache by an average of 12.7% and the total JIT data cache by an average of 19.7%. Together, ShareJIT reduces the total JIT cache by an average of 16.4%. From another perspective, ShareJIT saves about 1.9*MB* space across all 8 apps in an average total running time of 32.5 minutes. ShareJIT consumes about 300 KB space for the global sharing map in an entire experimental run.

Figure 4 compares the end-of-experiment JIT cache sizes of each app between default JIT and ShareJIT. More specifically, the left panel of Figure 4 shows that for every app, ShareJIT decreases the JIT code cache size. The maximum average decrease is in *Facebook*, where ShareJIT reduces the code cache size by an average of 22.7% (303.9 KB). The right panel of Figure 4 demonstrates that for JIT data cache, every app also shows reduction by running ShareJIT. The maximum average decrease is still in *Facebook*—26.9% (410.5 KB). The results of other apps are listed in Table 2.

As we mentioned in Section 5.2, we measured the CPU cycles used by each process to represent ShareJIT's and default JIT's performance. The left panel of Figure 5 shows the number of CPU cycles used by each app, including both kernel mode and user mode CPU cycles, per second of run. In order to eliminate the effect of different experimental times of different apps, we display the cycles of per second of run instead of the whole run. Thus the computation of the average speedup across all apps is meaningful. From this figure, we could see ShareJIT improves the performance of every app. The maximum speedup percentage appears in *Amazon*—an average of 21.9%. The average speedup across all apps per second of run is 9.0%.

Table 2. Code space saving and performance improvements due to ShareJIT compared with default JIT. The bottom row shows average total reductions across all 8 apps.

	JIT Cache Reduction (%)			CPU Cycles	Compilation Time
App	**Code**	**Data**	**Total**	**Reduction (%)**	**Reduction (%)**
Airbnb	10.3	17.7	14.2	1.5	28.5
Amazon	9.9	20.1	15.2	21.9	19.1
Chrome	8.0	11.8	10.0	3.2	21.4
Facebook	22.7	26.9	24.9	13.9	57.3
Firefox	3.3	8.4	6.0	3.9	20.1
Googlemaps	9.8	17.3	13.6	1.6	30.7
Instagram	9.2	19.7	14.8	15.5	39.8
Skype	11.3	18.9	15.0	1.7	25.7
Average	12.7	19.7	16.4	9.0	37.0

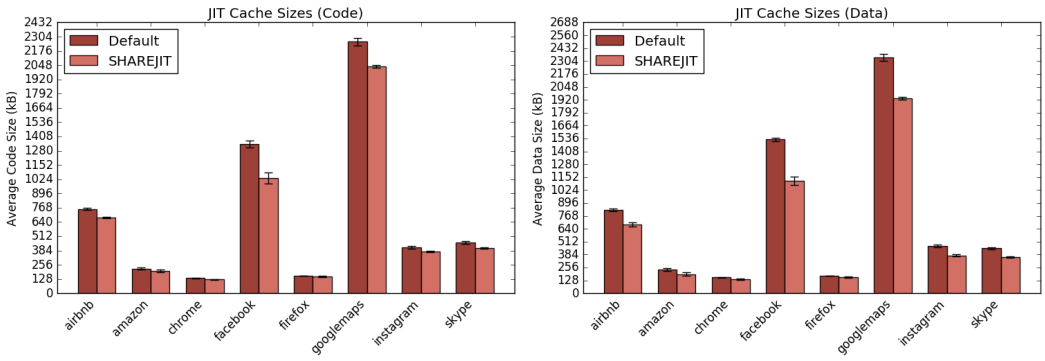

Fig. 4. Average end-of-experiment JIT code cache and data cache sizes for each app over all experiments using default JIT and ShareJIT. Error bars show 95% confidence intervals.

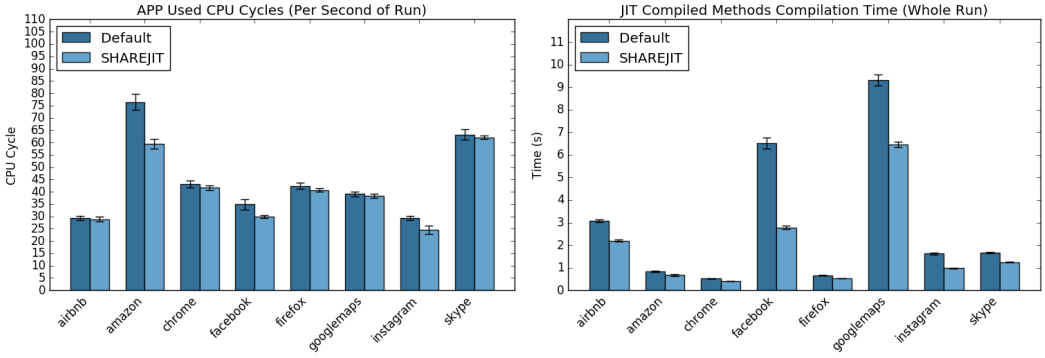

Fig. 5. Average app used CPU cycles per second of run and average total compilation time of JIT compiled methods for each app over all experiments using default JIT and ShareJIT. Error bars show 95% confidence intervals.

The right panel of Figure 5 illustrates the total compilation time of JIT compiled methods during run time. It is consumed and logged by the JIT compiler thread in ART. ShareJIT reduces the compilation time of each app. The average total compilation time reduction is 37.0%. And the maximum total compilation time reduction appears, again, in *Facebook*—an average of 57.3% (saving 3.7 seconds for a whole run).

Recall that in Section 4.3, we've discussed that theoretically, the system-wide memory utilization of ShareJIT is in proportion to the amount of sharing and the size of those methods. However, in practical experiments, the end-of-experiment JIT cache size of each app we measured was actually resulted from a combination of several factors. ('-" indicates this factor reduces the end-of-experiment cache sizes while "+" factors increase them.)

- ShareJIT eliminates the duplicate copies of compiled code for the same methods existing across apps;
- Restricted optimizations, e.g. less inline substitution, could reduce the size of a method's compiled code;

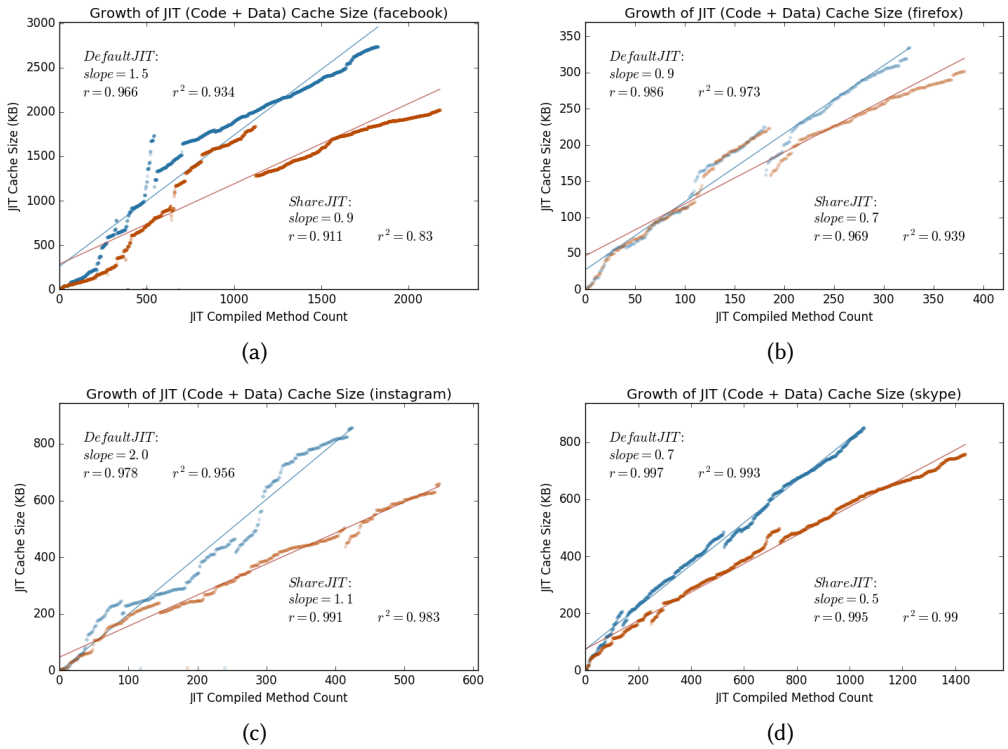

Fig. 6. A scatter plot to show the growth curve of JIT cache size of example apps during a whole experiment run. X axis is the count of JIT compiled methods, and y axis is the corresponding JIT data and cache size. Figure (a)(b)(c)(d) respectively show the growth curve of Facebook, Firefox, Instagram and Skype.

+ A ShareJIT garbage collector only collects the compiled code that is not currently being used by any process, so it may delay the collection of shared compiled code, leading to a temporary cache size increase;
+ Given a fixed time frame, ShareJIT could complete more compilation tasks in the JIT thread task queue because each task costs less time compared with default JIT. And more compilations can clearly cause an increase in the end-of-experiment cache size.

Figure 6 shows the example growth curves of JIT cache sizes during a whole experiment run for 4 apps. In sub-figure (a)(c)(d) which demonstrates the cache size growth of *Facebook*, *Instagram* and *Skype*, the ShareJIT curve (the red one) is consistently below the default JIT curve (the blue one). It is resulted from a combination of both restricted optimization and elimination of duplicate compiled code. All of the four figures also show that ShareJIT compiled more methods than the default JIT did in a given time frame. For example, in *Skype*, default JIT compiled about 1,100 methods while ShareJIT compiled about 1,500 methods (in a 235 seconds of running). Overall, ShareJIT manages to reduce the JIT cache size by an average of 16.4%.

6 RELATED WORK

The earliest related work we found with regard to code sharing at runtime is Dillenberger et al. [2000]—an implementation of the JVM for OS/390. In this work, multiple JVMs could share class data, constant pools, etc, which are stored in a shared heap during class loading, linking and

verification, while the compiled code is not shared. The authors briefly discussed these ideas at a high level without evaluating the actual performance of the system.

Czajkowski et al. [2002] described two systems—one allows the sharing of class meta-data, including bytecodes, among virtual machines, and the other additionally allows the sharing of dynamically compiled code. Both systems were implemented as modifications to the HotSpot JVM client compiler [Kotzmann et al. 2008]. The first one is similar to Dillenberger et al. [2000]. It divides the heap area into a shared part, a private part and an extra indirection part. Objects in the shared area have to reference objects in the private area via an entry in the indirection table. Each indirection holds the virtual address of the object associated with it, which can be different for each process. The second system uses these mechanisms to share compiled code with little modifications in JIT compilation, e.g. static variables access. Czajkowski et al. pointed out that their systems are less robust/secure because of the use of shared heap.

After Czajkowski et al.'s work, a variety of similar projects have addressed sharing class meta-data, class loaders, and other runtime representations in the JVM [Back et al. 2000; Berry et al. 2004; Bhattacharya et al. 2017; Daynes and Czajkowski 2005; Kawachiya et al. 2007; Kuck et al. 2009; Landau et al. 2011; Schmidt et al. 2008; Wintergerst 2008; Wong et al. 2003]. Some of them implemented multi-tasking VMs to share runtime memories [Czajkowski and Daynàs 2012; Czajkowski et al. 2003; Yan et al. 2016]. In addition, Oracle [Oracle 2018] and IBM [IBM 2018] also have production Java execution environments, which enable the sharing of class data, e.g., Java class bytecode and class file metadata, across JVMs. The sharing of class data requires significant modification to the virtual-machine-layer memory-layout, which couples the implementation tightly to a specific runtime system. In particular, for runtime systems like ART, which have a different class file format than the Java standard (one `.dex` file contains almost all the classes of an app, and symbolic references are globally unique and mutable during runtime), sharing class data would add too much overhead and, thus, be impractical.

Studies of code sharing across processes have been expanded to persistent code caches [Bruening and Kiriansky 2008; Guckert et al. 2013]—an effective way to reduce the overhead of dynamic binary translation, which translates binary code from one instruction set architecture to another at runtime, and is often used in system virtualization, system debugging, system security and whole program analysis. These projects have an orthogonal focus to our project's focus.

Huang et al. [2010] did work that is similar to our work. They proposed a native-code sharing mechanism across Dalvik virtual machines [Ehringer 2010] on Android 2.1, by storing all the compiled code in a shared file and implementing a daemon process to control the sharing. By contrast, ShareJIT composes a global shared cache and allows each process to manage its own cache segment instead of using a centralized agent. Since Android 5, the Dalvik VM was replaced by ART, and, in Android 2.1, the dex code of apps was only interpreted before being JIT compiled. By contrast, Android 7 uses a combination of interpretation, JIT compilation and AOT compilation to execute dex code.

Intuitively, Android 2.1 could offer more sharing opportunities than Android 7 does since ART has already AOT compiled some shared libraries in the system after its first boot-up. In Android 2.1, the JIT compiler is more like a typical AOT compiler since it generates position independent code with relocation information, while modern JIT compilers often leverage rich runtime context. The experiment shown in Huang et al. [2010] is preliminary—they tested 10 simple, small-size benchmarks instead of popular real-life apps with billions of downloads as we use; they measured the system performance through a score given by a single benchmark app called *Caffeinemark*. In contrast, we evaluate the performance of ShareJIT by measuring the CPU cycles used by full apps which include both the time spent in executing app methods and the time spent in the runtime system, e.g. compilation, garbage collection, etc.

7 CONCLUSION AND FUTURE WORK

The ShareJIT system provides a new framework for code caching and for sharing cached code across multiple applications in an Android environment. The goal of ShareJIT is to eliminate repeated JIT compilations of the same code and duplicate copies of the resulting compiled code, which occur in existing systems because applications share library code while the runtime system maintains process-private caches.

ShareJIT provides a global shared cache created by composing the private cache segments and providing coordinated, controlled access, lookup, and memory management across the full set of caches. The design recognizes that there is a fundamental tradeoff between shareability and optimization; the more optimized the compiled code is, the less shareable it is. ShareJIT increases shareability by restraining the amount of runtime context used during JIT compilation. For example, it limits the scope of inlining to increase sharing. Through ShareJIT, a single compile step can improve the performance of multiple processes and multiple apps. With ShareJIT, a single app can sometimes begin executing compiled code at an earlier point in its progress than it would be in the standard runtime system. ShareJIT improves overall system performance while reducing the total amount of memory devoted to caching compiled code and its associated data structures.

We build a practical implementation of ShareJIT in the Android Runtime system, and provide details of that implementation. We use this implementation to show that ShareJIT improves the overall system performance by an average of 9%, and decreases the memory utilization by an average of 16%. Additionally, ShareJIT also decreases the amount of time spent on JIT compilation by an average of 37%.

ShareJIT opens up opportunities for future work.

- There are a wide range of management policies to explore. For example, we intend to explore more deeply the relationship between the expected costs and benefits of sharing a method so that we could improve the decision making for when to share a method. E.g., an intuitive heuristic is that larger methods produce smaller payoff, because it costs more to compute their hash-identifications and they tend to exhibit less dramatic speedup from compiled execution.
- We also plan to experiment on some inter-process coordinations, e.g. maintaining a global sharing method hotness notion together, changing ownership of a method to a sharee process when the sharer/owner dies. Both create a better global perspective of runtime methods, but since they transfer information across the process boundary in Android, they also bring the overhead of inter-process communication and synchronization.
- We intend to port the Android implementation of ShareJIT to other open-source systems. In fact, ShareJIT was designed and implemented in an easy-to-port style, e.g., ShareJIT does not change the structure of the process-private code cache; it composes the original code cache segments together to form the global cache and gives each process execute access to the entire cache. This minimized changes to the memory management layer and the in-process JIT.
- Other subjects of possible interests include eviction and garbage collection policies, data mining ShareJIT's behavior to identify methods that tend not to be shared and should, thus, be compiled for performance rather than shareability.

REFERENCES

Godmar Back, Wilson C Hsieh, and Jay Lepreau. 2000. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*. USENIX Association, 23.

Robert F Berry, Donna N Dillenberger, Elizabeth A Hutchison, Susan P Paice, Donald W Schmidt, and Alan M Webb. 2004. Class sharing between multiple virtual machines. (May 18 2004). US Patent 6,738,977.

Dev Bhattacharya, Kenneth B Kent, Eric Aubanel, Daniel Heidinga, Peter Shipton, and Aleksandar Micic. 2017. Improving the performance of JVM startup using the shared class cache. In *Communications, Computers and Signal Processing (PACRIM), 2017 IEEE Pacific Rim Conference on*. IEEE, 1–6.

Jacob Brock, Chen Ding, Xiaoran Xu, and Yan Zhang. 2018. PAYJIT: space-optimal JIT compilation and its practical implementation. In *Proceedings of the 27th International Conference on Compiler Construction*. ACM, 71–81.

Derek Bruening and Vladimir Kiriansky. 2008. Process-shared and persistent code caches. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 61–70.

Grzegorz Czajkowski and Laurent Daynàs. 2012. Multitasking without compromise: a virtual machine evolution. *ACM SIGPLAN Notices* 47, 4a (2012), 60–73.

Grzegorz Czajkowski, Laurent Daynès, and Nathaniel Nystrom. 2002. Code sharing among virtual machines. In *European Conference on Object-Oriented Programming*. Springer, 155–177.

Grzegorz Czajkowski, Laurent Daynès, and Ben L Titzer. 2003. A Multi-User Virtual Machine.. In *USENIX Annual Technical Conference, General Track*. 85–98.

Laurent Daynes and Grzegorz Czajkowski. 2005. Sharing the runtime representation of classes across class loaders. In *European Conference on Object-Oriented Programming*. Springer, 97–120.

L Peter Deutsch and Allan M Schiffman. 1984. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 297–302.

Donna Dillenberger, Rajesh Bordawekar, Clarence W Clark III, Donald Durand, David Emmes, Osamu Gohda, Sally Howard, Michael F Oliver, Frank Samuel, and RW St John. 2000. Building a Java virtual machine for server applications: The JVM on OS/390. *IBM Systems Journal* 39, 1 (2000), 194–210.

David Ehringer. 2010. The dalvik virtual machine architecture. (2010).

Google. 2018a. Android – Android Oreo (Go edition). (2018). https://www.android.com/versions/oreo-8-0/go-edition/

Google. 2018b. Dex Bytecode | Android Open Source Project. (2018). https://source.android.com/devices/tech/dalvik/dalvik-bytecode

Google. 2018c. Dex File Format | Android Open Source Project. (2018). https://source.android.com/devices/tech/dalvik/dex-format

Google. 2018d. Enable Multidex for Apps with Over 64K Methods | Android Studio. (2018). https://developer.android.com/studio/build/multidex.html

Google. 2018e. Logcat | Android Studio. (2018). developer.android.com/studio/command-line/logcat.html

Google. 2018f. Low RAM Configuration | Android Open Source Project. (2018). https://source.android.com/devices/tech/perf/low-ram#lowmem

Lauren Guckert, Mike O'Connor, S Kumar Ravindranath, Zhuoran Zhao, and V Janapa Reddi. 2013. A case for persistent caching of compiled javascript code in mobile web browsers. In *Workshop on AMAS-BT*.

Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *European Conference on Object-Oriented Programming*. Springer, 21–38.

Yao-Chih Huang, Yu-Sheng Chen, Wuu Yang, and Jean Jyh-Jiun Shann. 2010. File-based sharing for dynamically compiled code on Dalvik virtual machine. In *Computer Symposium (ICS), 2010 International*. IEEE, 489–494.

IBM. 2018. Java Class data sharing between JVMs. (2018). https://www.ibm.com/support/knowledgecenter/en/SSYKE2_7.1.0/com.ibm.java.lnx.71.doc/user/classdatasharing.html

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

Kiyokuni Kawachiya, Kazunori Ogata, Daniel Silva, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. 2007. Cloneable JVM: a new approach to start isolated java applications faster. In *Proceedings of the 3rd international conference on Virtual execution environments*. ACM, 1–11.

Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the Java HotSpotTM client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization (TACO)* 5, 1 (2008), 7.

Norbert Kuck, Oliver Schmidt, and Ralf Schmelter. 2009. Sharing classes and class loaders. (Nov. 3 2009). US Patent 7,614,045.

Erez Landau, Dean RE Long, and Nedim Fresko. 2011. Shared JAVA jar files. (Feb. 1 2011). US Patent 7,882,198.

Oracle. 2018. Java Class data sharing. (2018). https://docs.oracle.com/javase/10/vm/class-data-sharing.htm

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

Oliver Schmidt, Norbert Kuck, Edgar Lott, Martin Strassburger, Arno Hilgenberg, and Ralf Schmelter. 2008. Sharing objects in runtime systems. (Aug. 19 2008). US Patent 7,415,704.

Toshio Suganuma, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue, Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani. 2000. Overview of the IBM Java just-in-time compiler. *./i systems Journal* 39, 1 (2000), 175–193.

TeamAA. 2018. What is Android Go? (2018). https://www.androidauthority.com/android-go-773037/

Michael Wintergerst. 2008. Centralized cache storage for runtime systems. (Aug. 26 2008). US Patent 7,418,560.

Bernard Wong, Grzegorz Czajkowski, and Laurent Daynès. 2003. Dynamically loaded classes as shared libraries: An approach to improving virtual machine scalability. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*. IEEE, 10–pp.

Yin Yan, Chunyu Chen, Karthik Dantu, Steven Y Ko, and Lukasz Ziarek. 2016. Using a multi-tasking VM for mobile applications. In *Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications*. ACM, 93–98.