HHVM Jump-Start: Boosting Both Warmup and Steady-State Performance at Scale

Guilherme Ottoni, Bin Liu
Facebook, Inc.
Menlo Park, CA, USA
{ottoni,binliu}@fb.com

Abstract*—Just-In-Time (JIT) compilation is often employed in Virtual Machines (VMs) to translate their virtual-machine languages into real-machine code. This approach not only brings portability, but it also enables aggressive compiler optimizations based on runtime behavior observed via profiling. The downside of JIT compilation, compared to Ahead-Of-Time native compilation, is that the profiling and compilation overheads are incurred during execution. To mitigate these overheads, previous work have proposed sharing either profile data or final JIT compiled code across VM executions. Unfortunately, these techniques have drawbacks, including steady-state performance degradation and difficulty of use. To address these issues, this paper presents the Jump-Start mechanism implemented inside the HipHop Virtual Machine (HHVM). Jump-Start is a practical approach to share VM profile data at a large scale, being used to power one of the largest websites in the world. In this paper, we argue for HHVM's Jump-Start approach, describe it in detail, and present steady-state optimizations built on top of it. Running the Facebook website, we demonstrate that Jump-Start effectively solves the warmup problem in HHVM, reducing the server capacity loss during warmup by 54.9%, while also improving steady-state performance by 5.4%.

***Index Terms*—virtual machine, JIT compilation, warmup, performance optimization**

I. Introduction

Virtual Machines (VMs) have become a common strategy for implementing a variety of high-level programming languages. This approach, which became very popular with the Java Virtual Machine (JVM), is currently used to implement languages like C#, JavaScript, Python, PHP/Hack, Lua, and many more. In a VM-based implementation, the application's source code, instead of being directly compiled to machine code, is compiled down to a higher-level virtual machine instruction set, typically called a *bytecode*.[1] During runtime, the application is executed via either interpretation or native machine code that is translated from the bytecode using a Just-In-Time (JIT) compiler. Compared to implementations based on Ahead-Of-Time (AOT) compilation to machine code, there are two important advantages of VMs: portability and the ability to transparently leverage runtime information to generate optimized machine code.

Optimized VMs use JIT compilation to improve performance. As mentioned above, an advantage of using JIT instead of AOT compilation is the ability to transparently use runtime information for optimizations. This is achieved via profile-guided optimizations (PGO), also known as feedback-driven optimizations (FDO). To implement PGO, VMs use multi-tier compilation, in which the code is first compiled with a simplistic JIT (or even interpreted) to collect profile data (*tier 1 compilation*), and later recompiled in optimized mode by leveraging the profile data (*tier 2 compilation*).

Compared to AOT, the downside of JIT compilation is the runtime overhead, particularly during the application's start-up. Because the compilation is performed at runtime, every cycle spent compiling the code is a missed cycle that could have been used to actually execute the application. This overhead is very pronounced during the start of the application, while it is being profiled and compiled, which is typically called the execution's *warmup phase*. After this phase, the application's execution reaches its *steady-state* or *peak* performance by executing optimized JITed code.

The poor performance during a VM's warmup phase is a well-known and important problem [1]–[5]. Although this problem has been extensively studied in the past, the lack of proper solutions in most production-quality VMs attests to the difficulty of solving this problem in practice.

In this paper, we describe Jump-Start, which is the approach used to address the warmup problem in the HipHop Virtual Machine (HHVM) [6], [7]. HHVM is a production-quality, high-performance VM implementing the Hack dialect of PHP [8], and which has been used to run some of the largest sites on the web, including Facebook, Wikipedia and Baidu [9]. Jump-Start adopts the approach of reusing JIT profile data across VM executions, similar to earlier work done in the context of Java [1], [4], [10]. In contrast to earlier work though, this paper describes how HHVM leverages Jump-Start to improve not only warmup but also steady-state performance. Furthermore, this paper describes practical aspects of our experience deploying Jump-Start in a large-scale production scenario, namely running the Facebook website.

Overall, this paper makes the following contributions:

1) It presents the design of Jump-Start and its implementation within HHVM.
2) It describes several Jump-Start-based steady-state optimizations implemented in HHVM.
3) It presents a thorough evaluation of the impact of Jump-Start on HHVM for running a real-world, large-scale workload, the Facebook website.

[1]The high-level representation may even be the source language, in so-called *language VMs*, which are typical for JavaScript.

4) It discusses reliability and other practical aspects of deploying Jump-Start in a production environment.

The rest of this paper is organized as follows. Section II presents background on HHVM as well as motivation for Jump-Start. Section III then makes the case for the design decisions behind Jump-Start, which is then presented in Section IV. The optimizations to improve steady-state performance using Jump-Start are presented in Section V. Section VI discusses reliability concerns and how this work addresses them. After that, Section VII presents an evaluation of Jump-Start, followed by a discussion of related work in Section VIII. Finally, Section IX concludes the paper.

II. Background and Motivation

The ability to seamlessly leverage runtime information is particularly important in JIT compilers for dynamic languages. This is because, in lack of profiling information, the quality of the compiled code for applications written in such languages suffers tremendously due to the wide realm of potential runtime behavior enabled by various dynamic features. Although VMs can naturally support profile-guided optimizations through JIT compilation, the overheads of JIT compilation can be significant, particularly for large-scale applications with huge amounts of code.

In this work, we study how to reduce the overhead of JIT compilation in the context of HHVM [6], [7]. HHVM is an ideal case study for this problem due to a combination of several factors: (1) it runs large-scale applications, (2) it implements a dynamic language where profile-guided optimizations are important to obtain performance, (3) it is a highly optimized, production-quality VM, and (4) it employs state-of-the-art compilation techniques [7]. In this section, we first provide a brief background about HHVM's compilation pipeline (Section II-A), then we outline the challenges faced by HHVM regarding JIT compilation overhead (Section II-B), followed by a discussion of available opportunities (Section II-C).

A. HHVM Background

HHVM employs a traditional VM architecture where the source code is compiled to a bytecode representation offline, before the application starts to execute. HHVM uses a custom bytecode designed specifically for representing PHP and Hack programs [6]. Due to the dynamically typed nature of these languages, the bytecode is untyped. The bytecode representation of the source code is deployed to the execution environment — in Facebook's case, to its fleet of web servers spread across many data centers. Compared to language VMs, this bytecode-based architecture allows some early compilation steps and optimizations to be performed offline, thus reducing runtime overheads. For production deployment (as opposed to development), where performance is more important, the application's bytecode is aggressively optimized offline using whole-program analyses and optimizations that benefit from the fact that the application code will not change at runtime [6], [7]. These optimizations significantly boost the application's steady-state performance, but they prevent incremental deployment

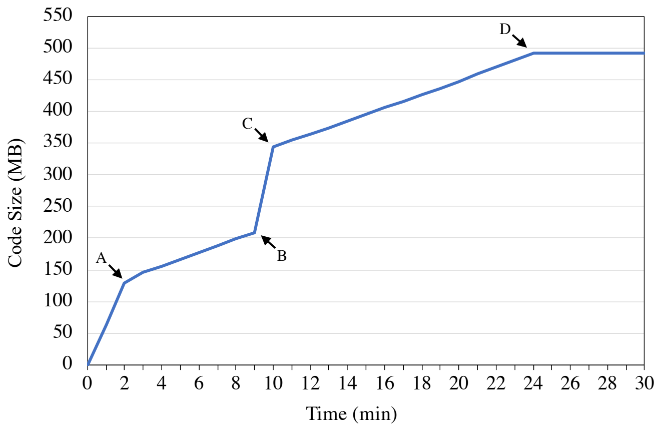


Fig. 1: JITed code size over time for HHVM running the Facebook website without Jump-Start.

and require the HHVM server to be restarted whenever a new revision of the application is deployed. At runtime, HHVM can execute the application's bytecode representation through either interpretation or JIT compilation. We discuss these two components next.

HHVM implements a traditional threaded bytecode interpreter [11], which serves two main purposes. First, it allows execution to make progress while the code is being compiled. Second, the interpreter provides a last resort for execution. This second purpose is particularly useful in case of HHVM because it removes the need to compile the entire source code which, as discussed in Section II-B, can be quite large.

The HHVM JIT compiler includes two alternative compilation strategies. The first one is a tracing-based compiler, which breaks the application into *tracelets* that can be efficiently compiled purely based on information collected by inspecting the live VM state [6]. The machine code produced for a compilation unit using this strategy is called a *live translation*. The second compilation strategy uses a region-based approach that aggressively optimizes arbitrary code regions by leveraging runtime information collected through profiling, thus implementing a traditional PGO framework [7]. For a given compilation unit, the machine code produced for profiling is called a *profiling translation*, while the final optimized code is called an *optimized translation*. In practice, the hottest portions of the application are compiled by the profile-guided region compiler to obtain the highest performance, while the remaining portions of the application that are still regarded worth compilation use the tracelet JIT.

B. Challenges

HHVM was primarily motivated by and designed to execute the Facebook website. This website has a monolithic architecture, with a huge code base consisting of over 100 million lines of code. In addition to this huge amount of code, compiling the website to machine code is even more challenging because of its very flat execution profile, with no single function representing anywhere close to 1% of the execution cycles and a very long tail of functions that execute. As a result, a very large number of functions need to be optimized by the JIT to significantly improve performance.

To illustrate the amount of code that HHVM deals with, Figure 1 plots the size of the JITed code over the initial 30 minutes of executing the Facebook website. In this figure, the Jump-Start technique described in this paper is not used. Figure 1 shows that HHVM produces nearly 500 MB of JITed code before it stops compiling, which happens after nearly 25 minutes into the execution (marked as point "D"). A few other points of interest are marked in Figure 1. At point "A", the JIT stops profiling new code and starts tier-2 compilation. Between points "A" and "B", the rate at which JITed code is produced reduces because the optimized code is not immediately placed in the code cache. Instead, the JITed code is placed in temporary buffers and only later relocated into the code cache according to the result of the function-sorting optimization (cf. Section 5.1.1 in [7]). The relocation of the optimized code into the code cache happens between points "B" and "C". Only at point "C" is that all the optimized code is available to execute — at this point, HHVM is already able to reach about 90% of its peak performance. Between points "C" and "D", the JIT continues to compile new code that is encountered via the tracelet JIT as described in Section II-A. At point "D", JITing ceases and the system finally achieves its peak performance.

During warmup, the server is able to handle an increasing number of *requests per second* (*RPS*). Figure 2 shows the normalized RPS over time for a typical web server when it restarts. At time 0, the old server process stops accepting new requests and shuts down. Then, the new server process starts, and it begins to handle requests after initialization. It takes about 25 minutes until the server reaches peak performance. In Figure 2, the area below the curve represents the amount of requests served, while the area above the curve represents the amount of extra requests that could have been served had the server not restarted. We call the latter the *capacity loss*.

The warmup overhead outlined above was reasonable in the early days of Facebook, when the website used to be updated (and HHVM restarted) once a day. However, over time, Facebook started to move to more frequent website updates, until it eventually switched to a continuous deployment approach in 2017 [12], [13]. With continuous deployment, Facebook is updated as soon as a new release from the master branch is cut and fully tested, which typically happens every 75 minutes in the common case where all tests succeed. Although continuous deployment improved developer productivity, it also exacerbated HHVM's JIT overhead. Suddenly, each HHVM server was spending about 13% of its life span until optimized code was produced and decent performance was reached, and 32% of its life span until reaching peak performance.

The effect of capacity loss during restart is even more alarming in the context of a large-scale service like Facebook, which serves over one billion users every day. The Hack code running on HHVM implements the core functionality of the Facebook website, thus running on a significant portion of Facebook's server fleet distributed over multiple data centers

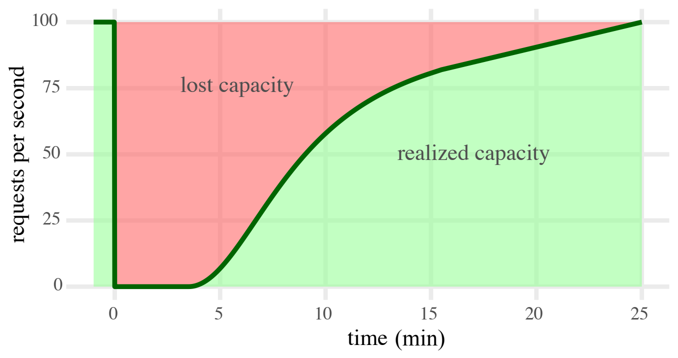


Fig. 2: Server capacity loss due to restart and warmup.

across the globe. While it is theoretically possible to limit capacity loss by limiting the number of servers that restart simultaneously, doing so would increase the time for updating the website, and thus prevent its continuous deployment. As a result, efficiency during warmup is important to reduce capacity demand for running the website with continuous deployment.

C. Opportunities

Fortunately, the large scale of Facebook's website that multiplies HHVM's warmup overhead also brings some opportunities. Given the website's monolithic nature, the Hack source code that runs on all web servers across the globe is identical. The web traffic driven to each data-center region varies greatly though, with very different patterns of web requests sent to each region. However, within each region, the web requests are much closer in nature. Within each data center, the load balancers route requests to servers using a technique called *semantic routing*, which partitions the website's endpoints into a fixed set of partitions (currently 10). The web servers are also partitioned into the same number of *semantic buckets*, each one corresponding to a semantic partition. The load balancers try to assign traffic according to this partitioning scheme, such that a request for an endpoint is preferably assigned to a server in the corresponding bucket, except when the preferred bucket is overloaded. Semantic routing improves the overall site performance because, due to HHVM's profile-guided JIT architecture, a request tends to execute on a server with JITed code that was more tailored for that particular endpoint. Within a pair of data-center region and semantic bucket though, the traffic is very similar, and so is the code produced by the JIT. This provides an opportunity to share some of the work involved in producing the JITed code across the large set of servers within the same region and semantic bucket.

Each time a new version of the Facebook website is deployed, the rollout happens in three phases: C1, C2 and C3 [13]. The C1 phase restarts HHVM on the web servers that handle requests coming from employees of Facebook, Inc. The second phase, C2, restarts 2% of the fleet and, finally, C3 restarts the rest of the fleet. The C1 and C2 phases serve to test and validate that a version of the website is good before it is deployed to the entire fleet. These initial phases contain extensive monitoring and alerts, which must remain healthy before the C3 deployment starts. This phased deployment technique, which is common practice for large-scale deployments, provides a very useful framework for improving HHVM's warmup by sharing JIT profile data collected at earlier phases.

III. The Case for Jump-Start JIT Compilation

When considering the possible language-implementation approaches, AOT compilation is the best alternative if the goal is warmup performance. In the specific case of Facebook, an earlier approach was to use an AOT compiler, HipHop [14]. HHVM was designed to replace the HipHop compiler for three main reasons. The first one is the already mentioned ability to transparently leverage runtime information for optimizations. Second, HHVM had the goal of supporting a fast edit-save-and-reload development cycle, which is very important for developer productivity and a main reason for using scripting languages. Finally, HHVM had the goal of solving the problem of dealing with huge amounts of compiled code produced for large-scale applications such as the Facebook website. Back in 2012, using HipHop to compile this website was already challenging due to resulting in binaries that were hitting Linux's 2 GB static-code limit. Today, Facebook's source code is multiple times larger, which would make AOT compilation even more challenging. For all these reasons, switching back to AOT compilation is not a compelling option, particularly for applications of Facebook's scale. Instead, we focused our effort on improving HHVM's warmup, while retaining the aforementioned VM benefits.

Our approach to address the warmup problem is to rely on the vast number of servers running the same workload, which is typical of large-scale applications, and to reduce the redundant work done across the servers during warmup. Two approaches have been used in the past to avoid some redundant warmup work: sharing profile data [1], [4], [10] and sharing JITed machine code [15]. The advantage of sharing JITed machine code is that a larger portion of the redundant work is reused, thus reducing the warmup overhead to a greater extent. Despite that advantage, instead we opted for sharing the profile data in Jump-Start for a few reasons:

1) *Full Optimizations.* As discussed in the ShareJIT work, sharing optimized code may require disabling some code optimizations [15]. For example, ShareJIT disables optimizations that embed absolute addresses and even inlining of user-defined methods. Unfortunately, disabling such optimizations can significantly degrade steady-state performance.
2) *Safety and Robustness.* There may be subtle cases where machine code must not be shared, and these may be tricky to detect. In the case of HHVM, the JIT compiler contains dozens of runtime options that control how the code is optimized and generated. For example, there are options to increase error levels (such as turning some specific warnings into fatal errors) and options that emit extra checks for debugging. Another usage is for emitting code differently depending on the target architecture or micro-architecture, such as to avoid issues specific to a particular processor model (e.g. [16]). Runtime options in HHVM are easily overwritten via configuration files, which provide a very rich mechanism to set different values for any option based on the machine model, server name, or user-defined machine groups/tiers. Besides JIT options, PHP/Hack also provide APIs to query these server properties, so even the application code may behave differently depending the value of these properties. In such scenarios, it is very challenging to robustly ensure the safety of sharing JITed code.
3) *Simplicity.* Sharing profile data is arguably simpler than sharing machine code. Once a mechanism for saving and reloading the profile data is built, the rest of the VM can operate in the same way, regardless whether the profile data was collected during that same VM execution or came from elsewhere. In contrast, sharing machine code requires making sure that all the emitted code is relocatable so that, once the code is loaded in a new VM instance, the VM can properly patch the addresses of various code and data elements that may reside at different locations.
4) *JIT Debugging.* Besides supporting sharing across VMs, having a mechanism to save and reload the JIT profile data is also useful for debugging JIT issues. If a collected profile triggers a JIT bug, compiler engineers can use that to replay and step through the execution of the JIT in order to reproduce and understand the issue, as well as to verify whether or not a candidate fix actually works.

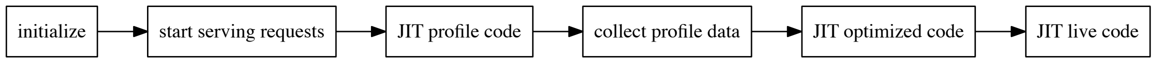
IV. HHVM Jump-Start

In this section, we give an overview of HHVM's workflow with Jump-Start (Section IV-A) and then describe the contents of HHVM Jump-Start's profile data (Section IV-B).

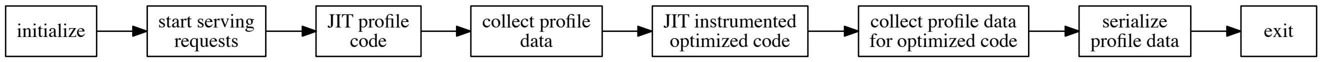
A. Workflow

Figure 3a summarizes HHVM's execution workflow without Jump-Start, which was described in Section II-B. After the HHVM process initializes, it starts serving requests and then JITing profile code. After running profile code for a few minutes to collect profile data, HHVM then recompiles the profiled code in optimized mode. After that, any new code that is reached is JITed in live mode. In steady state, the JITed code that executes is either optimized or live code.

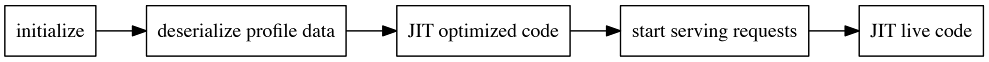
The goal of Jump-Start is to pre-compile the JITed code before starting to serve requests, so that the server can achieve high performance from the start. Given HHVM's architecture, we had two choices when it comes to what code could be pre-compiled: both optimized and live code, or just the optimized code. Although pre-compiling both the optimized and live code would completely eliminate the JIT overhead before starting to serve requests, we opted for just pre-compiling the optimized code. The reason for this is the time that it would take to collect the profile data needed for all live code. As Figure 1 shows, it takes HHVM nearly 25 minutes to finish producing all the



(a) No Jump-Start



(b) Jump-Start Seeder



(c) Jump-Start Consumer

Fig. 3: HHVM execution workflow without and with Jump-Start.

live code. Since we wanted to leverage HHVM's phased push, with profile data collected in the C2 phase, which lasts about 30 minutes, collecting profile data for the live code would not leave enough time to validate and distribute the profile data before the C3 phase. Therefore, we opted for only collecting profile data needed to JIT the optimized code, which takes about 10 minutes (cf. Figure 1). Furthermore, as mentioned in Section II-B, once optimized translations are produced, HHVM is already able to achieve about 90% of its peak performance, thus achieving a good trade-off between the time to produce the profile data and the performance benefit it provides.

Figure 3b illustrates HHVM's execution workflow on the *Jump-Start seeder* servers, which collect the profile data in the C2 push phase. The first four steps are similar to when Jump-Start is not used. The first difference occurs when JITing the optimized code, which contains some extra instrumentation used to collect profile data that feed some of the Jump-Start-based steady-state performance optimizations discussed in Section V. Once these profile data are collected, they are serialized and the server exits to perform validation of the profile data, which is discussed in Section VI.

The servers in the C3 phase execute in *Jump-Start consumer* mode. The execution workflow in this mode is illustrated in Figure 3c. In this case, after initialization, HHVM proceeds to deserialize the profile data, which was produced by the seeder servers and downloaded while the previous instance of the HHVM server was still running. After that, HHVM uses these profile data to JIT all the optimized code, which happens before the server starts to accept requests. Note that, because HHVM is not serving user requests yet, JITing happens in parallel using all the cores of the machine, which allows the optimized code to be compiled much faster than without Jump-Start. Once that finishes, HHVM starts to accept user requests and to produce live translations for any new code that was not compiled in optimized mode until the code cache fills up.

B. *Contents of the Profile-Data Package*

On Jump-Start consumers, it is necessary to make sure all inputs and context needed by the JIT are available before compiling optimized translations, which includes the following data categories:

1) *Necessary global data from the bytecode repo.* This category includes static (literal) strings, static arrays, and VM data representing units, classes and functions (including bytecode). Without Jump-Start, the in-memory data structures for repo global data are initialized on demand. For example, a unit (and classes/functions defined in it) is loaded into memory by the autoloader when executing the first request that uses it. This is not a problem without Jump-Start, because PGO is only applied to hot functions that are already loaded. However, on Jump-Start consumers, the JIT runs before any request. Thus, the lists of units, strings and arrays that need to be preloaded are included in the serialized data.
2) *JIT profile data.* This includes all the data necessary to create the optimized translations, including the bytecode instructions and types for each profile translation along with its execution count. This also includes all the JIT *target profiles*, which are custom counters that drive specific optimizations (e.g. call target profiles for method dispatch) [7].
3) *JIT profile data for optimized code.* This includes extra profile data that enables some of the steady-state optimizations based on Jump-Start discussed in Section V.
4) *Certain intermediate JIT results.* Some data, although they could be derived from the JIT profile data, may be worth including in the profile package to speed up Jump-Start, essentially moving their computation from the consumers to the seeders. The main example is the list with the order in which the functions should be placed in the code cache.

V. Boosting Steady-State Performance

Although the initial motivation for Jump-Start was to improve HHVM's warmup performance, we have also been able to leverage it to improve HHVM's steady-state performance. This section describes several optimizations that we were able to either add or improve by leveraging Jump-Start.

A. Improving Basic-Block Layout

Due to the huge amount of JITed code produced for running large-scale websites like Facebook, code-layout optimizations play a very important role in improving HHVM's performance on such workloads. As described in Ottoni [7], the HHVM JIT uses profile data collected by its tier-1 JIT in order to drive a set of code-layout optimizations including basic-block layout, hot/cold code splitting, and function sorting [17]. The HHVM JIT applies basic-block layout and hot/cold splitting together, driven by the same profile data, using the Ext-TSP technique [18]. This section describes how we were able to improve the effectiveness of these two optimizations using Jump-Start.

The profile data collected via the tier-1 JIT consists of instrumentation-based counters inserted at *bytecode-level* basic blocks. The basic-block layout and hot/cold splitting optimizations, however, are applied at the very end of the compilation pipeline, at HHVM's lowest-level intermediate representation, called Vasm [7]. The semantic gap between the granularities where the profile data is collected (bytecode) and used for code-layout optimizations (Vasm) inevitably results in inaccuracies in the Vasm block weights, due to both lowering steps and various code optimizations that are applied along the way. These inaccuracies in Vasm block weights can reduce the effectiveness of both the basic-block and the hot/cold-splitting optimizations. This same problem was identified in the context of static compilation by Panchenko et al. [19], who observed opportunities to improve the layout of code compiled by GCC and LLVM even when using their PGO frameworks. Their insight was that these opportunities resulted from both the gap in representations where the profile data is injected (at a high-level representation) and used for code layout (at the end of the compilation pipeline), and also the various code optimizations applied along the way. With that insight, they demonstrated that another pass of profile and code-layout optimizations, applied at the binary level using the BOLT binary optimizer, could greatly improve the quality of the code layout and the performance of the final binary. In this work, we used that insight to further boost the impact of HHVM's basic-block and hot/cold splitting optimizations.

In order to obtain accurate profile data to improve the layout of the optimized JITed code, it is necessary to profile the execution of the optimized code itself. In theory, the improved optimized code produced by leveraging this additional profile data could be produced by adding yet another optimization tier to the JIT (a 3$^{\text{rd}}$ tier). In practice, however, this would significantly increase HHVM's warmup time and the corresponding capacity loss, which were already a concern prior to Jump-Start. For this reason, such approach was never pursued before. Fortunately, Jump-Start provided a chance to explore the opportunity to improve HHVM's code-layout optimizations without sacrificing warmup performance, which we implemented as described next.

In the seeders, the optimized code is JITed with extra instrumentation to count the number of times that each Vasm-level basic block is executed. This instrumented optimized code runs for a few minutes to collect profile data, which is then included in the serialized profile. In the consumers, the only change is to read these extra profile counters from the profile data and use them to update the Vasm block counters right before applying the code-layout optimizations.

B. Improving Function Sorting

Another important code-layout optimization is function sorting. The HHVM JIT already implemented this optimization [7], using the C^3 algorithm proposed by Ottoni and Maher [20]. This section describes how Jump-Start enabled improving the effectiveness of this optimization.

The C^3 algorithm sorts the functions in a linear order based on a weighted, directed call graph [20]. In this call graph, the weight of an arc $(f \rightarrow g)$ is the frequency that function f calls function g. Prior to Jump-Start, this call graph was obtained by instrumenting the tier-1 JITed code produced by HHVM. The linear order produced by C^3 based on this call graph was then used to determine the order in which the optimized translations were placed in the JIT's code cache.

There is one major problem with the call-graph construction described above: HHVM's tier-1 compilation does not perform function inlining. As a result, although the call graph was representative of the tier-1 JITed code, it could be very inaccurate for the tier-2 optimized code, where function inlining is aggressively applied. And we note that enabling inlining in the tier-1 compiler is not a solution because it is unable to form non-trivial regions in the absence of profile data.

In order to fix the call-graph inaccuracy described above, we also used instrumentation of the optimized JITed code in the seeders, by inserting counters at the entry of each function. This profiling probe increments a counter corresponding to the caller-callee pair involved. Once the seeders have collected enough profile data for the tier-2 code, the JIT uses this new profiling data to build an accurate call graph. This call graph is then given to the C^3 algorithm to produce a linear function order, which is then appended to the serialized JIT profile data. The consumer servers simply load the desired function order out of the serialized profile data and emit the JIT optimized code following this order.

C. Improving Object Layout

In addition to improving code locality, we have also been able to improve data locality by leveraging Jump-Start. One optimization we created to improve data locality was to optimize the layout of user-defined objects. The goal of this optimization is to reorder the properties within objects by using profile information collected through Jump-Start.

Reordering of structure fields has been studied in the context of C applications in the past [21]–[23]. However, in C structures, reordering fields can affect correctness due to unrestricted use of pointers, thus rendering this optimization unsafe. Therefore, previous work focused on creating recommendation tools that suggest new orders for the fields, leaving it up to the programmer to verify the correctness of the transformation and then manually apply it.

Our work, in the context of PHP/Hack, does not face the issue with pointers in unmanaged languages like C, but it has two different constraints: (a) inheritance and (b) the declared order of the object properties is observable in PHP/Hack[2]. Since objects can inherit properties from its ancestor classes and subtyping must be honored, our technique only reorders properties within each layer of the class hierarchy. To deal with the observable order of properties, each object representing a PHP/Hack class is extended with an array that maps each property's declared index to the physical index in memory. This array is then used in all operations that require accessing the properties in their declared order. Although such operations become more expensive, they are not commonly used in practice.

In order to guide property reordering, we use the Jump-Start seeder machines to collect profile data about property accesses. For each non-static property P declared in class K, we count the number of accesses to this property by instrumenting the JITed profile code (tier-1). These counts are kept in a hash table mapping the string "K::P" to its corresponding counter. This hash table is then serialized along with the serialized JIT profile data.

In the consumer servers, when a PHP/Hack class K is created inside the VM, the order for K's inherited properties is copied from its parent class and then the order of its own, non-inherited properties is decided and appended. To decide this order, we use a simple hotness metric based on the property-access counters that were loaded from the JIT profile data. More specifically, the properties are sorted in decreasing order of their access counts. We note that previous work has also explored using the affinity of the fields/properties to decide on their order [21]. Using the affinity of access requires more expensive profiling, but it has the potential to further improve data locality. Exploring this opportunity inside HHVM is left for future work.

VI. Reliability Considerations

Although Jump-Start proves very effective in improving performance, it also raises some reliability concerns when deployed to large-scale production systems. This section discusses these concerns and how to address them in practice.

A. *Avoiding Widespread JIT Compiler Bugs*

Given HHVM's large-scale usage inside Facebook's data centers, combined with HHVM's own complexity and the huge size and constantly evolving nature of the Hack code base that it runs, it is nearly impossible to avoid hitting HHVM bugs in production. This is even more challenging due to HHVM's profile-guided JIT compilation approach. Even with unmodified source code, different profile data collected at runtime may expose bugs in the JIT compiler.

Although the profile-guided nature of the JIT compiler makes it harder to avoid compiler bugs, these characteristics also make HHVM more resilient to widespread compilation bugs. With each server collecting its own profile data, it is unlikely that all servers hit a rare bug. Furthermore, when a server crashes, it automatically restarts and the chances of hitting a bug again are small. Effectively, the rate of failed servers decreases exponentially with each attempt, because only the servers that crash will restart.

With Jump-Start, a straightforward deployment loses this natural resiliency that arises from each server independently collecting its own profile data. If a bad profile data is collected and shared with the entire server fleet, all the servers may crash at the same time. Even worse, upon automatically restarting, the servers will again crash in the same way, entering a crash loop and taking down the entire website.

To avoid the risks of a single "bad" profile-data package causing a major service disruption, we employed three techniques to improve Jump-Start's reliability:

1) *Validation of profile data.* Before publishing a profile-data package, each seeder validates that the profile data do not cause crashes during JIT compilation, and that the resulting optimized code does not lead to crashes or elevated error rates. To do so, a seeder server restarts HHVM in Jump-Start consumer mode using the profile data it just collected and only publishes the data if it remains healthy for a few minutes. Otherwise, the server restarts in seeder mode and repeats the entire process. We also store the problematic profile data on a database, so that rare bugs triggered by edge cases in the profile data can later be easily reproduced and debugged.
2) *Use of multiple, randomized profiles.* Instead of having a single seeder server for each data center and semantic partition (cf. Section II-C), we actually have several. Each seeder independently collects, validates, and publishes its own profile-data package. A consumer randomly picks a profile-data package for its corresponding data center and semantic partition each time it restarts. This way, even if a crash-inducing profile-data package slips through validation, it only affects a fraction of the consumers. Furthermore, when affected consumers crash and automatically restart, they will again pick a random, probably different, profile-data package, thus reducing the number of affected consumers exponentially with each restart.
3) *Automatic no-Jump-Start fallback.* If a consumer cannot find or download a suitable profile-data package, or it repeatedly fails to healthily start in Jump-Start mode, it will automatically restart with Jump-Start disabled, i.e. collecting its own profile data. This mechanism avoids

[2]For example, in PHP/Hack, an object can be casted to an array and then iterated over its properties in declared order.

not only issues with potentially bad profile data that managed to pass validation, but also the situation of the seeders failing to produce a package, or in case the package itself gets corrupted.

Finally, HHVM has a simple configuration option to disable Jump-Start, which can be quickly used as a last resort across the entire fleet of servers. Contrary to the techniques described above, which are fully automated, disabling Jump-Start via this configuration option is a manual process done at the discretion of the engineers. However, in our experience running the Facebook website using HHVM Jump-Start for over one year, we never had to resort to this mechanism.

B. *Avoiding Inferior Performance*

Improperly collected profile data on the seeders can negatively affect the performance of the entire fleet. This may happen for many reasons, with the most common one being that a seeder did not receive enough requests during profile collection because its data center was partially or fully drained at the time (e.g. due to network disruptions). The randomized selection of profile-data packages discussed in Section VI-A also helps reducing the performance impact in such cases. In addition, profile coverage, including the number of functions profiled and the total size of profile data, is checked against pre-configured thresholds before the profile data is published.

VII. Evaluation

This section presents an evaluation of Jump-Start's impact on HHVM's performance for running the Facebook website. We evaluate Jump-Start's impact on both warmup performance (Section VII-A) and steady-state performance (Section VII-B).

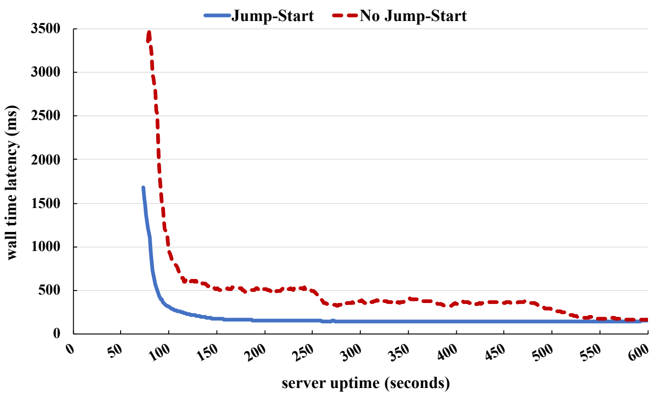
The servers used in this evaluation are powered by 1.8-GHz Intel Xeon D-1581 (Broadwell) microprocessors, with 16 cores and 32 GB of RAM. The servers run the Linux operating system.

A. *Warmup Performance*

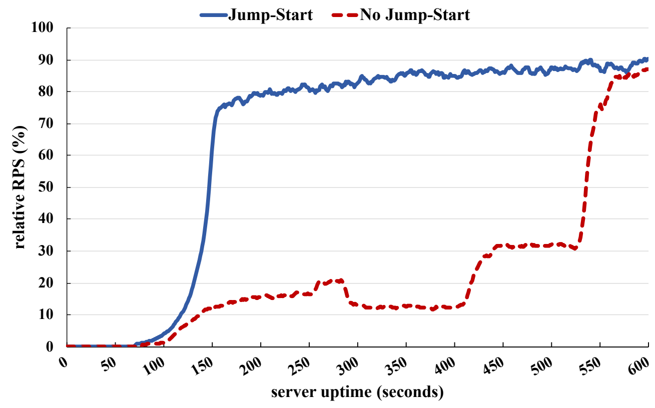
To quantitatively evaluate the warmup impact, we set up two production test tiers, one with and one without Jump-Start. Both tiers consist of more than 2000 web servers, and they are simultaneously pushed every 1 to 2 hours. Note that the servers were not loaded to their maximum capacity; instead, they were taking their typical production load.

Performance during warmup is evaluated in two aspects: *throughput*, as measured by RPS, and *latency*, as measured by the average wall time it takes to process a request. Both throughput and latency improve with server uptime during warmup. Thus, we plot performance-over-uptime curves in Figure 4 to show the trend during the initial 10 minutes of the servers' lifetime. We compare the warmup performance during the first 10 minutes because these are the most critical for our production deployment and, even without Jump-Start, HHVM is able to finish JITing all the optimized code and to achieve nearly 90% of its peak performance after that.

In Figure 4, the server uptime (x-axis) starts when the new HHVM process starts, while the curves start when HHVM starts to serve user requests. Recall from Figure 3c that, before starting to serve requests, Jump-Start already deserializes the profile data and JITs the optimized code, while no JITing happens in the no-Jump-Start case before starting to serve requests (Figure 3a). Despite doing more work prior to start serving requests, the Jump-Start servers still start to take user requests slightly earlier than the servers without Jump-Start. The reason for this discrepancy is due to how the initialization work is performed in either case. During initialization, HHVM executes a number of *warmup requests*, which serve the purposes of both warming up some HHVM extensions that talk to backend services and preloading some important bytecode and VM metadata into memory. Because the order in which these data are loaded affects data locality and steady-state performance, without Jump-Start, the warmup requests are executed sequentially. However, with Jump-Start, the serialized profile data already includes lists encoding the order in which these metadata should be loaded, and these data are preloaded before running the warmup requests. As a result, with Jump-Start, the warmup requests can be run in parallel, which greatly reduces the time to process them. In the end, this win in initialization with Jump-Start is more than enough to offset the time that it takes to JIT all the optimized code, which is



(a) average latency (wall time) per request over uptime



(b) normalized RPS over uptime

Fig. 4: Benefits of Jump-Start on (a) latency and (b) throughput during warmup.

also faster with Jump-Start because the JIT runs concurrently using all hardware cores.

Figure 4a compares the average wall time per request over the first 10 minutes after the server starts. From the chart, we can see that Jump-Start brings about a 3× reduction in wall time per request between when HHVM starts to serve requests and 250s. This is mostly because, without Jump-Start, the initial requests spend most of their time loading and interpreting bytecode. Such overhead is greatly reduced after 250s, when the hot units are loaded, and profiling translations are created. However, a request still takes notably more time until optimized translations are finished, which happens a bit after 550s. In contrast, with Jump-Start, the per-request wall time on servers not only starts much lower but also converges to a level closer to the steady-state value at about 150s. Even after 550s, as the curves get much closer, the wall time is still slightly lower with Jump-Start due to the optimizations described in Section V.

Figure 4b shows the comparison of RPS over uptime. Note that the RPS values are normalized to those of servers that are fully warmed up running the same workload. Overall, in the first 10 minutes, a server without Jump-Start loses 78.3% of its capacity compared to the ideal case of a server that does not restart. In contrast, the capacity loss with Jump-Start during the initial 10 minutes is only 35.3%. Therefore, relative to running HHVM without Jump-Start, Jump-Start reduces the capacity loss during the initial 10 minutes by 54.9%.

B. Steady-State Performance

In order to measure steady-state performance, we conducted experiments while running the Facebook website serving production traffic under high load. A total of 200 servers were used per experiment, with one half running without Jump-Start and the other half running with Jump-Start. We note that both sets of servers run the same HHVM binary in our experiments, with just different configuration options used to enable/disable Jump-Start. For these experiments, we used an in-house performance-measurement tool. This tool synchronizes the start of web servers in all servers and then waits for them all to warmup. After that, the tool loads the servers to a target 80% CPU utilization and then collects performance data for a minimum of 30 minutes. The main performance metric reported by the tool is the throughput measured in RPS.

Figure 5 summarizes the steady-state performance impact of Jump-Start for HHVM running the Facebook website, as well the effects on some key micro-architectural metrics. The throughput speedup from Jump-Start is 5.4%, i.e. HHVM is able to serve 5.4% more requests in steady-state. The micro-architectural metrics shown in Figure 5 demonstrate the effectiveness of the Jump-Start-based optimizations. The code-layout improvements help to reduce front-end metrics, reducing branch misses by 6.8%, I-cache misses by 6.2% and I-TLB misses by 20.8%. Jump-Start's data-layout optimizations, including object-property layout and how Jump-Start preloads the bytecode meta-data, help by reducing D-cache misses by 1.4% and D-TLB misses by 12.1%. Finally, Jump-Start reduces last-level cache (LLC) misses by 3.5% due to a combination of both code and data locality improvements.

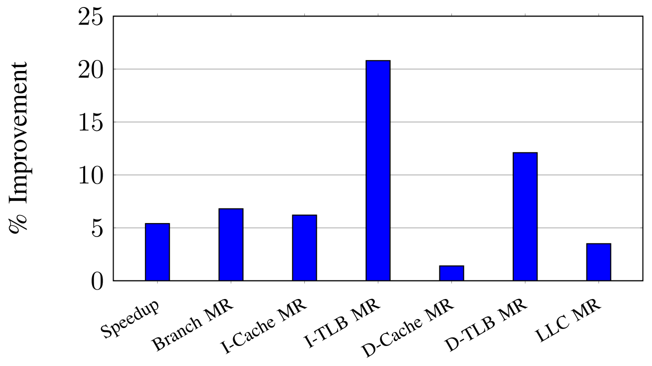


Fig. 5: Speedup and various micro-architectural miss reductions (MR) for Jump-Start over no Jump-Start.

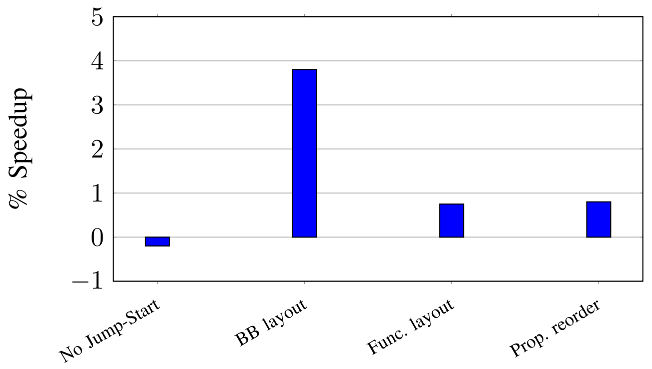


Fig. 6: Speedup of no Jump-Start and Jump-Start-based optimizations over Jump-Start without optimizations.

Figure 6 presents the steady-state performance impact of enabling Jump-Start and each of the optimizations described in Section V. In this graph, the baseline is Jump-Start without any of those optimizations. The first bar measures the impact of disabling Jump-Start, which causes a 0.2% regression. In other words, HHVM is about 0.2% faster with Jump-Start enabled versus disabled, even without the optimizations from Section V. The other three bars show the speedup of individually enabling each of the optimizations described in Section V. Basic-block reordering (along with hot/cold splitting) gets the largest speedup, 3.8%. This confirms the observations of the BOLT work [19] regarding the importance of very accurate profile data to drive these optimizations. The third bar in Figure 6 shows that HHVM gets a 0.75% speedup by improving the accuracy of the call graph used to sort the functions in the code cache. Finally, the last bar shows that the new object-property reordering optimization enabled by Jump-Start results in a 0.8% speedup.

VIII. Related Work

To the best of our knowledge, Arnold et al. [1] was the first work to describe reusing profile data across virtual-machine executions. Their work differs from ours in several aspects.

First of all, their profile repository is local, restricted to a single server, while our approach is to share profile data across many servers, which is very important for large-scale deployments. Second, they use the profile data mostly to decide the optimization level to compile each method given the amount of time spent in the given method and the estimated speedup of compiling the code at that optimization level. Unlike Jump-Start, their mechanism is not used to perform any additional optimization. Finally, compared to our approach, Arnold et al. [1] also has the disadvantage that the application still initially runs in interpreted mode and compiled code is produced during the application's execution. This results in two overheads: slower execution due to the initial interpretation and JIT overhead to produce optimized code. Our technique, in contrast, pre-compiles the optimized code before starting to execute the application, therefore eliminating both of these overheads.

Majo et al. [4] presents an approach similar to Arnold et al. [1], in which profile data is cached and reused across executions. Similar to Jump-Start, but in contrast with Arnold et al. [1], Majo et al. [4] uses the profile data to directly produce optimized code (tier 2) for all the profiled methods. In contrast to Jump-Start though, this compilation only happens after the code is run through the interpreter to trigger compilation. This is similar to Arnold et al. [1]'s approach, and it shares with them the same disadvantages compared to Jump-Start regarding runtime interpretation and compilation overheads.

Another work that closely relates to ours is Azul Zing's ReadyNow [10]. Similar to HHVM Jump-Start, ReadyNow also allows reusing JIT profile data across runs, so that the profile can be collected in a small set of servers and then reused across a large set of servers. However, ReadyNow requires the user to insert directives in the code to prevent invalid optimizations that would break the application's semantics [24], [25]. In contrast, HHVM Jump-Start does not require user annotations to guarantee correctness, thus being more transparent and robust. These properties are particularly important for HHVM's dominant use-case of running large-scale and rapidly evolving web applications.

Krintz [26] describes a hybrid approach that combines online and offline profiling. The goal of that work was to reduce the drawbacks of each one, namely the overhead of online profiling and the potentially staleness of offline data. In our work, we solely use offline profiling to reduce the JIT overhead, and we address the potentially staleness of the offline profile by freshly collecting it for the exact workload and usage scenario where the VM is going to be used.

Xu et al. [15] describes ShareJIT, which is a technique to share JIT compiled code across processes, in the context of Java. In order to make the compiled machine code shareable, ShareJIT disables some optimizations (e.g. embedding absolute addresses and inlining of user-defined methods), which then degrades steady-state performance. In contrast, the approach presented in this paper not only allows all JIT optimizations to be applied but also supports additional optimizations to further boost steady-state performance.

Béra et al. [2] snapshots the optimized intermediate representation instead of the final machine code to save some of the compilation overhead.

Finally, we highlight that all previous approaches from the literature that reuse profile data across runs lack the additional optimizations included in HHVM Jump-Start presented in Section V. To the best of our knowledge, HHVM Jump-Start is the first of these techniques to improve not only warmup performance but also steady-state performance.

IX. Conclusion

This paper motivated and argued for the Jump-Start compilation approach developed in the context of HHVM. This technique has been deployed across the Facebook website, leading to a 54.9% reduction in the capacity loss during HHVM warmup. Furthermore, this paper also described several optimizations that were either enhanced or completely built on top of the Jump-Start mechanism, providing an additional 5.4% steady-state performance. Finally, although we studied and evaluated these techniques in the context of HHVM, we believe that they are general enough and thus can be useful to improve both warmup and steady-state performance of other optimizing VMs.

Acknowledgments

We would like to thank Mark Williams for contributing the initial implementation of HHVM's mechanism to serialize and deserialize the JIT profile data used by Jump-Start. We also thank Joe Gasperetti and Russell Cloran who helped with monitoring and other production-related aspects necessary for successfully deploying Jump-Start inside Facebook's production environment. Finally, we thank Sergey Pupyrev and the anonymous reviewers for their valuable feedback on earlier drafts of this paper.

References

[1] M. Arnold, A. Welc, and V. T. Rajan, "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*, ser. OOPSLA, 2005, p. 297–311.

[2] C. Béra, E. Miranda, T. Felgentreff, M. Denker, and S. Ducasse, "Sista: Saving optimized code in snapshots for fast start-up," in *Proceedings of the 14th International Conference on Managed Languages and Runtimes*, ser. ManLang, 2017, p. 1–11.

[3] D. Lion, A. Chiu, H. Sun, X. Zhuang, N. Grcevski, and D. Yuan, "Don't get caught in the cold, warm-up your JVM: Understand and eliminate jvm warm-up overhead in data-parallel systems," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI, 2016, p. 383–400.

[4] Z. Majo, T. Hartmann, M. Mohler, and T. R. Gross, "Integrating profile caching into the hotspot multi-tier compilation system," in *Proceedings of the 14th International Conference on Managed Languages and Runtimes*, ser. ManLang, 2017, p. 105–118.

[5] E. Barrett, C. F. Bolz-Tereick, R. Killick, S. Mount, and L. Tratt, "Virtual machine warmup blows hot and cold," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017.

[6] K. Adams, J. Evans, B. Maher, G. Ottoni, A. Paroski, B. Simmers, E. Smith, and O. Yamauchi, "The Hiphop Virtual Machine," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA, 2014, pp. 777–790.

[7] G. Ottoni, "HHVM JIT: A profile-guided, region-based compiler for PHP and Hack," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, 2018, p. 151–165.

[8] Hack Language, Web site: http://hacklang.org, 2020.

[9] HHVM Team, "HHVM Users," Web site: https://github.com/facebook/hhvm/wiki/users, October 2017.

[10] Azul, "Zing readynow! – faster java execution at application startup and beyond," Web site: https://assets.azul.com/files/Readynow-Data-Sheet-1-19v1.pdf, 2019.

[11] J. R. Bell, "Threaded code," *Commun. ACM*, vol. 16, no. 6, pp. 370–372, Jun. 1973.

[12] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm, "Continuous deployment at Facebook and OANDA," in *Proceedings of the 38th IEEE/ACM International Conference on Software Engineering Companion*, ser. ICSE, 2016, pp. 21–30.

[13] C. Rossi, "Rapid release at massive scale," Web site: https://engineering.fb.com/web/rapid-release-at-massive-scale/, August 2017.

[14] H. Zhao, I. Proctor, M. Yang, X. Qi, M. Williams, Q. Gao, G. Ottoni, A. Paroski, S. MacVicar, J. Evans, and S. Tu, "The HipHop compiler for PHP," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA, October 2012, pp. 575–586.

[15] X. Xu, K. Cooper, J. Brock, Y. Zhang, and H. Ye, "ShareJIT: Jit code cache sharing across processes and its practical implementation," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018.

[16] Intel, "Mitigations for jump conditional code erratum white paper," Web site: https://www.intel.com/content/dam/support/us/en/documents/processors/mitigations-jump-conditional-code-erratum.pdf, Nov. 2019.

[17] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, ser. PLDI, 1990, pp. 16–27.

[18] A. Newell and S. Pupyrev, "Improved basic block reordering," *IEEE Transactions on Computers*, vol. 69, no. 12, pp. 1784–1794, 2020.

[19] M. Panchenko, R. Auler, B. Nell, and G. Ottoni, "BOLT: A practical binary optimizer for data centers and beyond," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO, 2019, p. 2–14.

[20] G. Ottoni and B. Maher, "Optimizing function placement for large-scale data-center applications," in *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO, 2017, pp. 233–244.

[21] T. M. Chilimbi, B. Davidson, and J. R. Larus, "Cache-conscious structure definition," in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, ser. PLDI, 1999, p. 13–24.

[22] R. Hundt, S. Mannarswamy, and D. Chakrabarti, "Practical structure layout optimization and advice," in *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO, 2006, pp. 233–244.

[23] E. Raman, R. Hundt, and S. Mannarswamy, "Structure layout optimization for multithreaded programs," in *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO, 2007, p. 271–282.

[24] V. Grazi and G. Tene, "Azul readynow! seeks to eliminate jvm warm-up," Web site: https://www.infoq.com/news/2014/03/Azul-ReadyNow-Eliminates-Warmup/, March 2014.

[25] F. Yin, D. Dong, S. Li, J. Guo, and K. Chow, "Java performance troubleshooting and optimization at alibaba," in *Proceedings of the IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2018, pp. 11–12.

[26] C. Krintz, "Coupling on-line and off-line profile information to improve program performance," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO, 2003, p. 69–78.