Towards Optimal Remote JIT Compilation Scheduling for the JVM

Richard Kha*
University of Toronto

Nikhil Sreekumar*
University of Toronto

Alexey Khrabrov
University of Toronto and IBM

Eyal de Lara
University of Toronto

Angela Demke Brown
University of Toronto

Moshe Gabel
University of Toronto

Marius Pirvu
IBM

Abstract

In the Java Virtual Machine (JVM), Just-In-Time (JIT) compilation is used to speed up Java applications, however, JIT compilation incurs significant memory and CPU runtime overheads. JITServer is a disaggregated JIT compiler in the Eclipse OpenJ9 JVM – it decouples the JIT compiler from the JVM, thereby reducing CPU and memory requirements of JVM client applications.

JITServer schedules remote compilation requests from connected clients in first-come, first-served (FCFS) order. We show that when a JITServer instance is under high load, the scheduler performs better when it considers client information for each compilation request. Prioritizing requests from newly connected clients helps those clients warm-up rapidly, but risks starving requests from older clients. We developed a scheduling algorithm we call ALDCF (Alternating Least Done Client First) that prioritizes requests from clients with fewer completed compilation requests, while reducing starvation of older clients through alternating with FCFS. In our experiments, ALDCF reduces the average client JVM runtime by up to 9% compared to FCFS, while controlling starvation.

1 Introduction

Modern cloud applications are commonly implemented in languages such as Java, Scala, and Kotlin that run on the Java Virtual Machine (JVM) [3, 5], making the JVM runtime a key piece of cloud software infrastructure. The JVM initially executes applications by interpreting Java bytecodes, but performance depends on just-in-time (JIT) [7] compilation, which compiles bytecodes into optimized machine code tailored to the specific application and runtime environment. JIT compilation, however, consumes significant memory and CPU resources, especially during application start-up and warm-up phases [8]. This problem is exacerbated in cloud datacenters, where JVMs often run in resource-constrained environments to increase application density, and application instances are often short-lived (e.g., microservices, automatic load scaling, etc.).

JIT compiler disaggregation [1, 4] is a promising solution to the problem of JIT compilation overheads in cloud workloads. Decoupling the JIT compiler from the application JVM instance prevents competition for resources between compilation and application execution, leading to higher application quality of service, faster warm-up, and more predictable behaviour while reducing per-application-instance memory footprint [8]. A single JIT compilation server could support a large number of client JVMs by exploiting statistical multiplexing, i.e., as long as client compilation requests are reasonably spread out, the server should be lightly loaded and able to respond rapidly. Unfortunately, this is not always the case, e.g., a spike in demand for a particular application may trigger automatic load scaling to progressively launch new instances, leading to a storm of simultaneous compilation requests from many clients.

In this work, we focus on the challenges of scheduling compilation requests from multiple clients in an overloaded JIT compilation server. Since the performance gain due to compiling a method for a client may vary across clients, the overall benefit of the JIT compiler server can be increased by judiciously choosing the next compilation. We explore this challenge in the context of *JITServer*, the disaggregated JIT compiler in the popular open-source Eclipse OpenJ9 JVM [2].

JITServer Background: In the JITServer implementation, client JVMs decide which methods to compile, in which order, and at what optimization level, using the same heuristics as with local JIT compilation. Requests to compile these methods are then sent to the server using a configurable number of client compilation threads. Thus, each client typically has multiple pending compilation requests submitted to the server. As with local JIT, a method may be compiled multiple times at increasing optimization levels. Methods are initially compiled at the "cold" level which applies few optimizations but has a major performance impact since it moves the method from interpretation to native execution. Frequently invoked methods may be recompiled at "warm", "hot", or "scorching" levels, which progressively apply more

*Both leading authors contributed equally

HotInfra '24, November 3, 2024, Austin, TX, USA

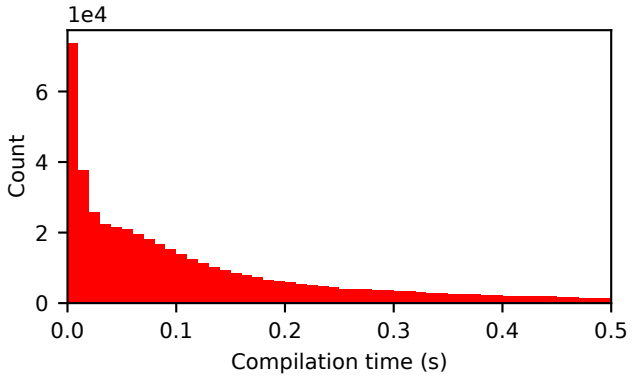

Figure 1. Compilation time distribution for Dotty

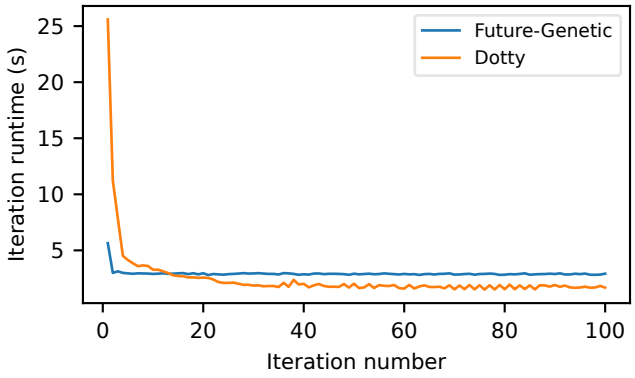

Figure 2. Client warm-up for Dotty and Future-Genetic

optimizations and consume more compilation resources to achieve higher application performance.

FCFS Scheduler: At the JITServer, remote compilation requests fill a queue, and the server schedules its entries for compilation in first-come, first-served (FCFS) order. For a lightly loaded server, this policy is reasonable since clients have already prioritized the order of their requests, and the server-side queue is short. We observe that when the server is overloaded with many clients, they accumulate a backlog of requests, and scheduling them at the server becomes increasingly important. While launching additional JITServer instances could reduce load, this comes with added costs, takes time, and may not always be possible.[1]

2 Improved Scheduling Policy

Wierman and Zwart [10] suggest that FCFS, the current JITServer scheduler, is optimal for light-tailed workloads. Figure 1 shows the distribution of compilation times on the JITServer for remote compilations requested by clients running the Dotty workload from the Renaissance benchmark suite [6, 9]. The distribution is spread out with high variability, and is clearly not light-tailed, suggesting that FCFS is unlikely to be optimal. Compilation times vary widely because of the different sizes and complexity of compiled methods, as well as higher costs at higher optimization levels.

We also examine the warm-up curves of two Renaissance benchmarks, Dotty and Future-Genetic, to show the diminishing returns of later compilation requests from the same client JVM. Dotty runs a Scala compiler on a set of source code files and focuses on data structures and synchronization. Future-Genetic runs a genetic algorithm and focuses on parallelization. These two benchmarks are well above and well below the *hot code size* and *hot method count* geometric means of all Renaissance workloads, respectively [9]. Thus, these two benchmarks represent opposite ends of the performance spectrum for the JVM and remote JIT compilation.

[1]Note that JITServer instances are shared only by client JVMs of the same end user. We are not concerned with issues of security or fairness across client application instances.

We used a single client JVM instance running 100 iterations of either benchmark to see the impact on warm-up time with a lightly loaded JITServer. Figure 2 shows the runtime for each iteration. For both benchmarks, we see that the per-iteration runtime decreases sharply during the first few iterations, but eventually reaches a steady state. The point where performance stabilizes is defined as the end of the warm-up stage. During warm-up, we can also see diminishing returns from JIT compilation, particularly for Dotty – during the first 5 iterations the per-iteration runtime quickly decreases by over 20s, while during the next 20 iterations runtime gradually decreases only by another 2s. The largest benefit comes from initial compilations at the "cold" optimization level, while recompiling methods at "warm" and "hot" levels has less impact, although these requests consume more JITServer resources.

In scenarios where multiple client JVMs start in a staggered fashion with overlapping warm-up periods, such as during automatic load scaling, the server queue will contain a mix of requests from clients at different execution stages. However, with an FCFS scheduler, the requests are handled in their arrival order, without consideration for how much benefit the client will gain. The results in Figure 2 suggest that we should prioritize requests from clients in the early execution stages to ensure that these impactful compilations are done earlier.

We developed a **Least Done Client First (LDCF)** scheduler that picks the earliest request from the client with the least number of compilations done so far as the next request to handle. However, a stream of requests from new clients may starve requests from older clients that have previously accumulated many compilations. Based on this observation, we propose the following **Alternating Least Done Client First (ALDCF)** scheduling policy: *alternate* selection of the next request to serve between (a) the earliest request from the client with the fewest completed requests; and (b) the next request in FCFS order, independent of the client. Alternating with FCFS scheduling prevents starvation of later-stage clients while still prioritizing clients in their warm-up stage.

3 Results and Discussion

In this section we evaluate the performance of the proposed ALDCF remote compilation scheduler compared to LDCF, Round-Robin (RR), and the baseline First-Come, First-Served (FCFS). We implemented ALDCF, LDCF, and (RR) in the OpenJ9 JITServer, modifying it to enable *client-aware* scheduling, and using the number of compilations processed for each client to prioritize scheduling decisions.

Experimental Setup: We ran our experiments on a machine with a 14-core (28-thread) Intel Xeon E5-2680 v4 CPU and 128GiB of RAM. We put the JITServer instance under high load by running it in a Docker container bound to a single core and having 50 client JVMs connect to it. Note that since the clients and server run on the same machine, there is minimal communication latency. We *stagger* the starts of the client JVMs to have them connect while in different execution phases. We chose the stagger intervals empirically depending on the benchmark to maintain load on the server and to spread the clients sufficiently for phase differences. This setup simulates a realistic workload as the server is CPU-bound as expected under high load, and the client staggering mirrors varying execution phases in connected clients.

We used the Dotty and Future-Genetic benchmarks from the Renaissance suite [6, 9]. In each experiment, we run 50 client JVMs that execute 100 iterations of the benchmark each, and use the ALDCF, LDCF, RR, and FCFS schedulers to compare their performance in the following modes:

- **Single workload**: All client JVMs run the same workload, starting in a staggered fashion with a 50s interval for Dotty and 10s for Future-Genetic.
- **Mixed workload**: The clients run alternating workloads (Dotty for even-numbered clients, Future-Genetic for odd-numbered ones), staggered with an interval of 30s.

Single Workload: Figure 3 shows CDFs of client completion times for single-workload Dotty and Future-Genetic experiments. We observe that most Dotty clients experience a significant speedup under the LDCF and ALDCF schedulers. We also see that ALDCF consistently decreases runtimes compared to RR or FCFS, while LDCF's behaviour varies depending on the workload.

For Dotty, compared to FCFS, ALDCF achieves a speedup of 1.08, while LDCF achieves a speedup of 1.18 at the 80th percentile of runtimes. We also see a 9% decrease in cumulative runtime for ALDCF and a 17% decrease in cumulative runtime for LDCF compared to FCFS. For Future-Genetic, ALDCF achieves a speedup of 1.04 and LDCF achieves a speedup of 1.09 vs. FCFS at the 80th percentile of runtimes. However, we see that ALCDF leads to a <1% increase in cumulative runtime, while LDCF leads to a sizeable 18% increase in cumulative runtime compared to FCFS.

Mixed Workload: Figure 4 shows the results for a *mix* of Dotty and Future-Genetic clients. In this scenario, LDCF is the worst scheduler for Dotty, but the best scheduler for

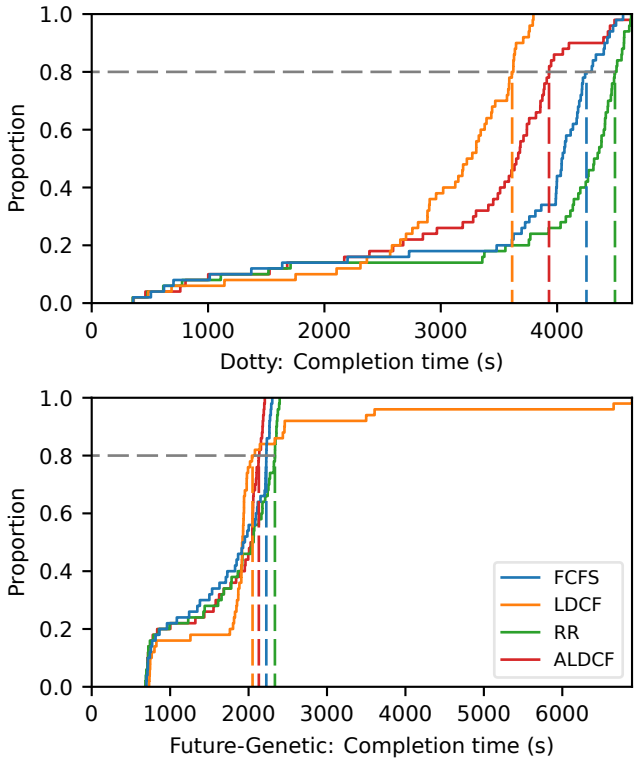

Figure 3. CDFs of single workload completion times

Future-Genetic, while ALDCF has more stable performance. For Dotty clients, FCFS is the best scheduler; in comparison, ALDCF and FCFS have speedups of 0.92 and 0.88, respectively, at the 80th percentile of runtimes. For Future-Genetic clients, ALDCF and LDCF have speedups over FCFS of 1.53 and 1.77, respectively, at the 80th percentile of runtimes.

During these experiments, the effect of starvation on client JVMs manifests as compilation request timeouts, which happen when requests take longer than 30s. For Dotty, only 0.16% of compilation requests time out with FCFS, but this increases to 7.0% of requests with LDCF. ALDCF alleviates the starvation issue with only 0.21% of requests timing out. Similar trends are seen for Future-Genetic.

Discussion: Our results demonstrate that incorporating per-client progress data in remote JIT compilation scheduling decisions can significantly impact client performance, providing a promising approach for optimizing remote JIT compilers. From the performance of FCFS and RR schedulers, we observe that treating all compilation requests equally is sub-optimal for most workloads. However, *over-prioritizing* newer clients can lead to starvation of older ones, especially clients within their warm-up stage. ALDCF provides a solution by alternating each scheduled request between prioritized and fair scheduling.

There is a stark contrast in the behaviours and performance benefits of schedulers, especially ALDCF and LDCF, in the *single workload* and *mixed workload* experiments. These

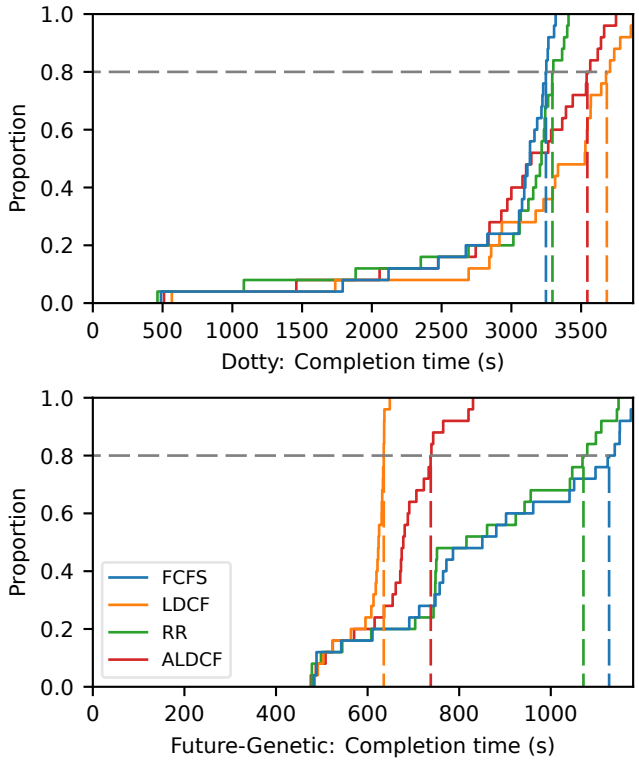

Figure 4. CDFs of mixed workload completion times

schedulers prioritize compilations based on client information. Therefore, the execution stages of clients with respect to each other, as well as the relative cost of compilations significantly affects scheduler performance. We discuss this and its implications below.

4 Future Work

We claim that ALDCF is a beneficial scheduling approach for remote JIT compilations and show early experimental results supporting this claim. Our next steps are to explore a wider variety of experiments for further validation and a nuanced understanding of its benefits. We plan on incorporating a larger set of workloads with a high variation of *code size* and *hot method count*. We are also interested in how the scheduler performs with varying workload intensities on the JITServer, as well as when running multiple JITServer instances to optimize reactive scale-out.

ALDCF alternates each scheduled request between prioritized and fair scheduling to balance the priorities of remote compilation requests. However, there are other approaches to seeking a *balanced* scheduler. We plan to explore different ratios and orders of performing prioritized and fair schedules. The respective client execution phase is an important factor to decide the aggressiveness of heuristic-based prioritization. Therefore, a Multi-Armed Bandit approach to dynamically decide the aggressiveness of prioritization is a promising direction to pursue in future work.

Prioritizing clients based on the number of compilations performed does not consider that clients running mixed workloads may have varying warm-up curves and execution phases. Profiling workloads to more accurately model the relative execution stage of clients may improve ALDCF's performance in handling mixed workloads. One possible simple approach is to use the proportion of compilations served for a client so far out of the maximum number of compilations seen for this client's application as a measure of how far the client is into its warm-up phase.

Our preliminary experiments have established the effectiveness of using client information in scheduling decisions. Future research will explore what specific information is most useful for scheduling. For example, we could use JITServer CPU time spent handling each client's requests, rather than the number of methods compiled. Since some compilations are costlier than others, using CPU time would be a fairer basis for client priority.

A remote JIT scheduler may benefit from multiple streams of client information. We will therefore investigate lightweight machine learning approaches to scheduling given client information. Since we know that different applications act differently based on the scheduler, this is potentially useful due to its ability to recognize underlying patterns of workloads, and schedule accordingly.

References

[1] [n. d.]. *Azul Cloud Native Compiler.* https://www.azul.com/products/prime/cloud-native-compiler/
[2] [n. d.]. *Eclipse OpenJ9.* https://eclipse.dev/openj9/
[3] [n. d.]. *Java Language and Virtual Machine Specifications.* https://docs.oracle.com/javase/specs/
[4] [n. d.]. *JITServer Technology — Eclipse OpenJ9 Documentation.* https://eclipse.dev/openj9/docs/jitserver/
[5] [n. d.]. *OpenJDK.* https://openjdk.org
[6] [n. d.]. *Renaissance Suite, a Benchmark Suite for the JVM.* https://renaissance.dev
[7] [n. d.]. *JIT Compiler | Eclipse OpenJ9 Documentation.* https://eclipse.dev/openj9/docs/jit/
[8] Alexey Khrabrov, Marius Pirvu, Vijay Sundaresan, and Eyal de Lara. 2022. JITServer: Disaggregated Caching JIT Compiler for the JVM in the Cloud. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 869–884. https://www.usenix.org/conference/atc22/presentation/khrabrov
[9] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: benchmarking suite for parallel applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 31–47. https://doi.org/10.1145/3314221.3314637
[10] Adam Wierman and Bert Zwart. 2012. Is Tail-Optimal Scheduling Possible? *Operations Research* 60, 5 (2012), 1249–1257. http://www.jstor.org/stable/23323693