

Performance Implications of JPMS Adaptation

Experiments with Open-Source Applications **RENEW** and **Kieker**

Marcel Hansson
marcel.hansson@uni-hamburg.de
University of Hamburg

Daniel Moldt
daniel.moldt@uni-hamburg.de
University of Hamburg

Abstract

The Java Platform Module System (JPMS) has been a central component of Java since its release in Java 9. Since the module system introduces a different architecture and JVM-related execution, we investigate which aspects of Java programs must be considered from a performance perspective. We conduct targeted experiments using two medium-sized open-source software systems: **RENEW** and **Kieker**. We compare modularized and non-modularized versions using application-specific workloads. The results show a nuanced picture: The JPMS introduces overhead in some scenarios, but primarily in the start phase. Based on these experiments, we provide initial insights into when JPMS should (or should not) be used.

1 Introduction

The Java Platform Module System (JPMS) was introduced in Java 9 [3] to provide stronger encapsulation and reliable configuration through modules that explicitly declare dependencies and exports. Applications using JPMS utilize the module path instead of, or in combination with, the classpath. Improved performance was also among the goals of its introduction, as stated in Java Specification Request 376 [4].

Whether JPMS affects the performance of real applications, and to what extent, remains seemingly unexplored. We investigate this question by comparing modularized and non-modularized versions of two real-world applications: the Petri net editor and simulator **RENEW** [8] and the observability framework **Kieker** [7]. For each application, we create both versions with minimal changes, perform the same application-specific tasks, and measure their performance.

Our results show that JPMS does not necessarily improve performance and can, in some cases, introduce performance overhead. Developers should therefore consider this overhead alongside the benefits of JPMS and the requirements of their specific use case. Based on our experience developing the investigated software, we also briefly discuss benefits that motivated the adoption of JPMS in **RENEW** and may motivate its use in **Kieker**.

The remainder of this paper briefly introduces **RE-**

NEW, **Kieker**, and JPMS, followed by the experiments and their results. We conclude with a discussion, summary and directions for future research.

2 Foundations

2.1 Renew

One of the example applications we investigated is **RENEW**, the Reference Net Workshop [8], a Petri net editor and simulator for the reference net formalism [1]. Reference nets extend colored nets with Java inscriptions, Java objects as tokens, and nets-within-nets [2]. **RENEW** was adapted to the JPMS with release 4.0 [6] and was chosen for two reasons. First, it uses JPMS module layers to dynamically load plugins, providing insights into module-layer performance. Second, our familiarity with its source code allowed us to re-engineer a comparable version without JPMS.

2.2 Kieker

The second example is the observability framework **Kieker** [7], which monitors and analyzes large-scale software systems, primarily Java applications. As part of a current project, JPMS adoption is being evaluated, providing both JPMS and comparable non-JPMS versions for our experiments.

2.3 Java Platform Module System

The Java Platform Module System (JPMS) [3], introduced in Java 9, organizes Java applications and libraries into explicit modules. Modules declare their dependencies and exported packages in a *module-info.java* file. A module layer is the runtime structure used to load and organize modules. Java starts with a boot layer containing the core Java modules, while additional layers can be created to load modules independently, enabling isolation and dynamic loading.

3 Experiments with Renew

3.1 System

The experiments were conducted on an Intel Xeon E5-2630 v4 with 128 GB RAM running Debian 12.15 and on an Intel Ultra 7 265 with 64 GB RAM running Debian 13.6. We refer to these systems as "Xeon" and "Ultra", respectively.

Table 1: Median method execution times (ns) in RENEW with 95% bootstrap confidence intervals. Change is relative to No-JPMS.

System	JPMS	No-JPMS	Change
Xeon	99484	98316	+1.17%
	[97820, 101217]	[96819, 99412]	
Ultra	15663	16002	-2.16%
	[14582, 16127]	[14685, 16324]	

3.2 Preparation of Renew

We used release 4.2 of RENEW to create comparable modularized and non-modularized versions. Later releases use the module service loader system, which cannot be easily reverted. For the non-modularized version, we adapted the modularized version to run on the classpath by disabling module-layer setup and the associated class loader. This required only minor code changes; the non-modularized version is then started by changing the Java invocation.

3.3 Test Scenario

We evaluated RENEW by simulating a reference net that repeatedly fires a transition 10,000 times and terminates the applications afterward.

3.4 Experiments

We conducted two experiments. First, we measured the execution time of RENEW’s *ExecuteFinder.execute()* method over 200 repetitions per version and system. The observed differences in median execution time were small, and the 95% bootstrap confidence intervals for the differences included zero, indicating no statistically significant difference between the two versions (see Table 1).

Second, we measured timestamps before Java invocation, at the beginning and end of `main`, and after termination. Since the Petri net runs in a separate thread, execution continues after `main`, which primarily handles plugin setup and loading.

As shown in Table 2, No-JPMS reaches `main` approximately twice as fast as JPMS on both systems. We compared the mean execution times using Welch’s t-test, testing the null hypothesis that the mean durations are equal between versions. No-JPMS has a significantly lower mean total runtime on both Xeon and Ultra ($p < 0.001$). The mean duration of `main` does not differ significantly on Xeon ($p = 0.369$), whereas on Ultra, No-JPMS is approximately 2.14% faster ($p < 0.001$). These results suggest that the additional runtime associated with JPMS occurs primarily before `main`.

3.5 Benefits of JPMS for Renew

JPMS provides several benefits for RENEW. Since version 2, RENEW has used a plugin architecture that allows plugins to be loaded and unloaded at runtime. Previously, RENEW relied on a custom class loader to unload plugins, as Java does not directly support this. JPMS module layers provide each plugin with

Table 2: Runtime comparison for RENEW (ms). Differences are No-JPMS minus JPMS; CIs are 95%.

System	Phase	JPMS	No-JPMS	Diff.	[95% CI]
Xeon	Pre	109.9	51.5	-58.4	[-59.2, -57.6]
	Main	341.4	342.4	+ 1.0	[- 1.2, + 3.2]
	Total	4280.2	4209.7	-70.6	[-97.6, -43.6]
Ultra	Pre	49.2	25.1	-24.2	[-24.4, -24.0]
	Main	136.0	133.1	- 2.9	[- 3.3, - 2.5]
	Total	1212.4	1159.5	-52.8	[-63.9, -41.7]

its own class loader, making dynamic unloading easier and more reliable. This is particularly beneficial for long-running RENEW instances that need to update plugins at runtime. JPMS also provides stronger encapsulation through explicit module boundaries, exports, and dependencies. This improves the architecture by allowing the module system to enforce clear dependency structures.

4 Experiment with Kieker

4.1 Preparation of Kieker

We adapted Kieker to JPMS with minimal changes by converting each Gradle subproject and its source set into a module using a `module-info.java` file. Due to its classpath-based dependencies, most modules additionally require access to the unnamed module. The trace analysis tool was fully adapted and tested using a new run script that executes it on the module path.

4.2 Test Scenario

We evaluated the JPMS impact on Kieker using its trace analysis tool, the only analysis tool currently fully adapted to JPMS. We used the trace analysis tool to create an aggregated deployment call tree for three datasets: *small* (6,540 records), *medium* (63,933 records), and *large* (781,453 records).

4.3 Experiments

We measured the system time in Bash before and after executing the trace analysis tool and repeated the measurement 500 times for each version, dataset, and system. As shown in Table 3, JPMS increased runtime by 2.65%–29.99%, with larger overheads on the Xeon and smaller overheads for larger datasets. All differences were statistically significant according to a Welch’s t-test ($p < 0.05$).

For the large dataset in an additional experiment, we measured pre-main, main, and total runtime, as in the second RENEW experiment. The results are shown in Table 4. As in RENEW, the largest runtime difference occurred before `main()`, where No-JPMS reduced mean duration by 78% on the Xeon and 76% on the Ultra.

4.4 Benefits of JPMS for Kieker

Unlike RENEW, Kieker does not require dynamic plugin loading. The main benefits of JPMS for Kieker are therefore improved encapsulation and modular or-

Table 3: Runtime comparison for the trace analysis tool (ms). Change is relative to No-JPMS.

System	Dataset	JPMS	No-JPMS	Change
Xeon	Small	1032.69	794.44	29.99%
Xeon	Medium	1452.30	1215.42	19.49%
Xeon	Large	6230.16	5916.56	5.30%
Ultra	Small	479.98	404.83	18.56%
Ultra	Medium	891.26	811.80	9.79%
Ultra	Large	6299.00	6124.58	2.65%

Table 4: Main runtime comparison on the large dataset (ms). Differences and 95% confidence intervals are No-JPMS minus JPMS.

System	Phase	JPMS	No-JPMS	Diff.	[95% CI]
Xeon	Pre	385.6	83.6	-302.0	[-304.3, -300.0]
	Main	5790.6	5809.3	+ 18.7	[- 8.6, + 46.1]
	Total	6215.7	5933.3	-282.4	[-309.9, -254.9]
Ultra	Pre	154.9	37.4	-117.5	[-117.9, -117.1]
	Main	6151.6	6052.4	- 99.2	[-122.8, - 75.5]
	Total	6327.0	6111.1	-215.9	[-239.6, -192.3]

ganization, as well as smaller deployment artifacts. The current development version already reduces the trace analysis tool’s artifact size from 135 MB to approximately 99 MB, further optimization is likely possible. JPMS also provides tools for creating custom runtime images. Using this approach, the artifact is approximately 130 MB and includes the required JRE, eliminating the need for a separate Java installation.

5 Discussion

The experiments show that JPMS does not necessarily improve application performance. No-JPMS was faster in all measured scenarios, although the single-method RENEW experiment showed no statistically significant difference. For the complete application, the largest difference occurred before `main`, suggesting that startup contributes substantially to the overhead.

This effect is particularly relevant for short-lived applications and frequently invoked command-line tools, whereas its relative impact is likely smaller for long-running applications. The results therefore do not imply that JPMS generally makes applications slower, but that its impact depends on execution characteristics. This is notable because improved performance, including startup performance [5], was among the goals associated with JPMS.

The observed overhead must be weighed against the benefits of modularization. For RENEW, module layers simplify dynamic plugin unloading and explicit dependencies and exports improve architectural structure. For Kieker, JPMS improves encapsulation and modular organization while reducing artifact size.

Several limitations should be considered. The adaptations themselves may affect performance, and only two applications and a limited number of workloads were examined. Furthermore, we investigated applications rather than libraries, leaving the performance impact of fully modularized libraries open. A mixture of module-path and classpath libraries may also affect performance.

6 Summary and Outlook

We investigated the performance impact of JPMS using two real-world Java applications, RENEW and Kieker, comparing minimally changed modularized and non-modularized versions on two systems. No-JPMS achieved shorter runtimes in all experiments. For RENEW, the largest difference occurred before `main`, suggesting a startup-related overhead. For Kieker, JPMS overhead ranged from 2.65% to 29.99% depending on system and dataset. These results indicate that JPMS performance should be considered when adopting it, particularly for applications where startup is significant. However, JPMS also provides benefits including stronger encapsulation, explicit dependencies, smaller deployment artifacts, and improved support for dynamic plugin unloading.

Further experiments are needed to identify the precise causes of the observed overhead. In particular, factorial experiments could vary the number of modules and classes independently to determine how each factor affects performance. It should also be evaluated, if there may also be a performance improvement for long-running pure module applications. Additional experiments could also investigate compile-time performance and the performance impact of modularized libraries.

Acknowledgement This research is funded by the Deutsche Forschungsgemeinschaft (DFG – German Research Foundation), grant no. 528713834.

References

- [1] O. Kummer. *Referenznetze*. Berlin: Logos Verlag, 2002.
- [2] R. Valk. “Object Petri Nets – Using the Nets-within-Nets Paradigm”. In: *Advances in Petri Nets: Lectures on Concurrency and Petri Nets*. Vol. 3098. Lecture Notes in Computer Science. Springer-Verlag, 2004, pp. 819–848.
- [3] A. Bateman et al. *JEP 261: Module System*. Oct. 23, 2014. URL: <https://openjdk.org/jeps/261>.
- [4] M. Reinhold. *Java Platform Module System: Requirements*. Apr. 1, 2015. URL: <https://openjdk.org/projects/jigsaw/spec/reqs/2015-04-01>.
- [5] N. Parlog. *The Java Module System*. Manning Publications, June 2019.
- [6] L. Clasen et al. “Enhancement of Renew to Version 4.0 using JPMS”. In: *Proceedings of PNSE 2022 co-located with PETRI NETS 2022*. Vol. 3170. CEUR Workshop Proceedings. CEUR-WS.org, 2022, pp. 165–176.
- [7] S. Yang et al. “The Kieker Observability Framework Version 2”. In: *Companion of the 16th ACM/SPEC International Conference on Performance Engineering, ICPE 2025, Toronto, ON, Canada, May 5-9, 2025*. ACM, 2025, pp. 11–15.
- [8] O. Kummer et al. *Renew – The Reference Net Workshop*. Release 5.0. May 2026.