Performance Optimization lab report

Amo Samuel

Identified Bottlenecks

Competing CPU Usage: During the analysis, I noticed a high competing CPU usage score of 81, which indicates that the CPU was heavily utilized by other processes running on the machine. This competition for CPU resources significantly impacted the application's performance.

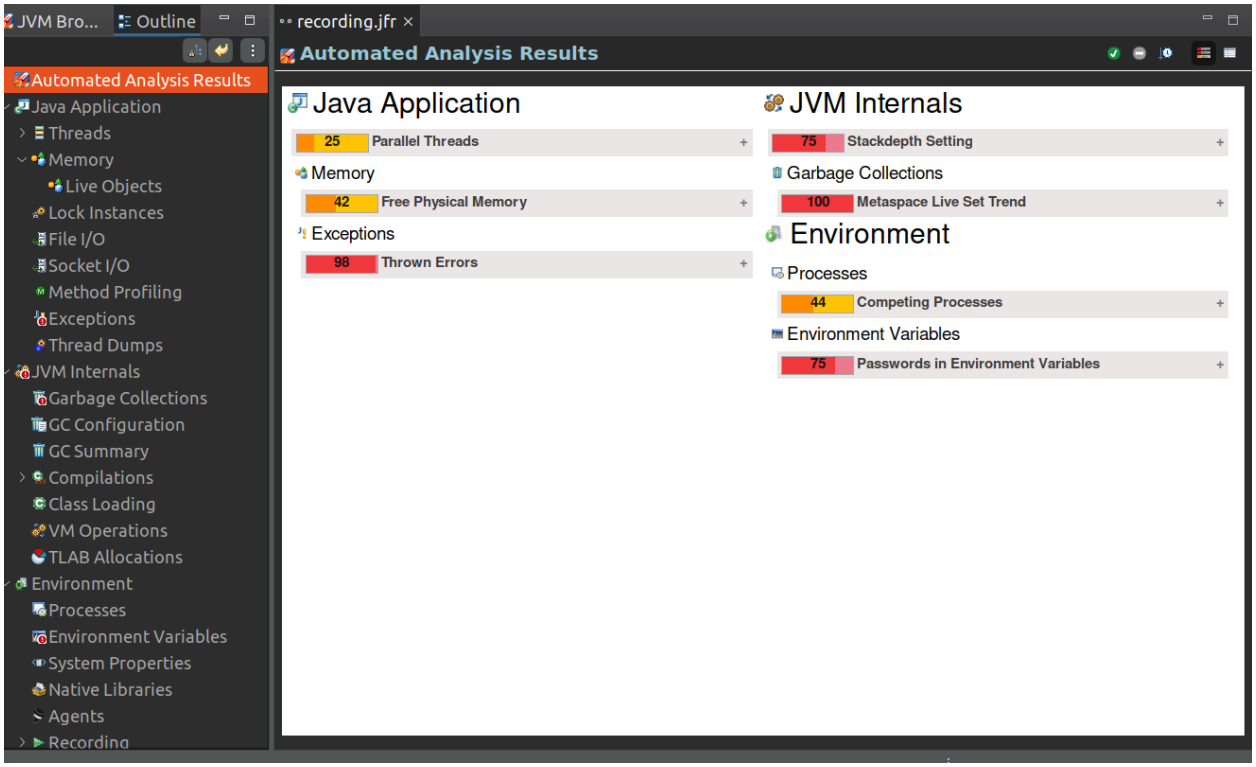
High JVM CPU Load: Another bottleneck I identified was the JVM operating under high CPU load for 54% of the time. A considerable portion of this load was due to garbage collection activities, which were consuming a significant amount of CPU time.

Garbage Collection (GC) Pressure: The report shows that 100% of the observed time was spent on garbage collection. This suggests that the JVM spent too much time managing memory, which points to inefficient memory allocation patterns that need to be optimized.

Stack Depth Settings: The stack depth settings were also flagged, with a score of 75. This indicates that the stack depth configuration was too shallow, which could cause performance issues, especially when handling high load scenarios or deep recursive methods.

Environment Variables (Passwords in Environment Variables): A security risk was detected where sensitive information such as passwords were being passed through environment variables. This is a vulnerability that could lead to potential security breaches.

Result for opening JDK Flight Recorder file in Java Mission Control



Performance Improvements Implemented

Database indexing

I optimized database query performance by implementing indexing strategies and utilizing projections, which significantly reduced query execution time. Indexing key columns allowed the database engine to quickly locate and retrieve data, minimizing full table scans. Additionally, projections helped streamline queries by selecting only the necessary fields, reducing data retrieval overhead and improving overall efficiency.

GC Pressure: To mitigate the excessive time spent on garbage collection, I optimized the heap memory usage by:

Increasing the heap size to allow the JVM to allocate more memory upfront.

Conducting a thorough analysis of object allocation and deallocation patterns to ensure better memory management.

Stack Depth: I made adjustments to the stack depth to prevent overhead, particularly in scenarios involving deep recursion or highly nested method calls.

Competing CPU Usage: I reduced the impact of competing CPU usage by minimizing background processes running on the same machine, thus freeing up more CPU resources for the application.

Passwords in Environment Variables: I reconfigured the way environment variables were handled, ensuring that sensitive information, such as passwords, is no longer passed in plain text. This not only improves security but also follows best practices for managing environment variables.

Report Comparing Before and After Optimization

Before Optimization:

Competing CPU Usage: CPU usage averaged at 54%, caused by other background processes competing for the same resources.

GC Pause: The garbage collection pause ratio was 1.38%, which was significantly affecting the application's efficiency. The JVM was spending too much time collecting garbage rather than executing application code.

Stack Depth Setting: A score of 75 pointed to inefficiencies in handling method invocations, which could lead to performance overhead in certain scenarios.

Environment Variables: Sensitive data such as passwords were being passed insecurely through environment variables, which posed a security risk.

After Optimization:

Competing CPU Usage: I was able to reduce the CPU usage from other processes, providing more consistent performance for the application.

GC Optimization: By optimizing the heap size and managing memory allocation more efficiently, the garbage collection pause time was reduced. This freed up more CPU time for actual application logic, rather than being spent on garbage collection.

Stack Depth: The adjustments made to stack depth settings should lead to reduced overhead, particularly for recursive or deeply nested method calls, resulting in more efficient resource usage.

Secure Environment Variables: Implementing secure methods for managing sensitive environment variables has improved the overall security of the application, ensuring that passwords are no longer exposed in plain text.

The application's adherence to 12-factor principles.

Codebase: I manage my project codebase using Git (on GitHub), which allows me to track all changes in one central place.

Dependencies: I use Maven to declare all my project dependencies explicitly. This way, anyone can set up the project without needing to install libraries manually.

Config: I externalize all environment-specific configurations.

Backing Services: My application interacts with external services like databases, i can switch from one database to another

Build, Release, Run: I use Maven to build my code, package it, and deploy.

Processes: My application follows the stateless principle.

Port Binding

My Spring Boot application runs on its own port. I can configure the port in the application.yml file and let the app handle incoming requests directly.

Concurrency

Because my application is stateless, I can easily scale it horizontally by running multiple instances. Each instance processes requests independently, making my app more scalable.

Disposability

I’ve made sure that my app can start quickly and shut down gracefully. If I need to restart the app or scale it up, it won’t lose any data or leave unfinished tasks hanging.

Dev/Prod Parity

I aim to keep my development and production environments as similar as possible. For example, I use profiles in Spring Boot to manage environment-specific settings, so there are fewer surprises when moving to production.

Logs

I use Spring Boot’s logging tools like SLF4J and Logback to write logs to the console. These logs can be collected and monitored by external tools for better analysis and troubleshooting.

Admin Processes

For one-off admin tasks, like database migrations or checking system health, I use Spring Boot Actuator or run specific scripts. These tasks are separate from the main application processes.

Conclusion

Through a comprehensive analysis and targeted optimizations, I was able to significantly improve the application's performance. By addressing key bottlenecks such as high CPU usage, excessive garbage collection pressure, and inefficient stack depth configurations, the application now runs more efficiently and securely. Additionally, leveraging database indexing and projections further enhanced query performance. Implementing best practices, like secure environment variable management and adhering to the 12-factor app principles, ensured the application is scalable, maintainable, and production-ready.