ej technologies

White Paper: The Strategic Value of JProfiler in Modern Software Development

Empowering Development Teams to Achieve Optimal Performance and Cost Efficiency

Executive Summary

In the competitive landscape of software development, efficiency and performance are not merely goals but prerequisites for success. JProfiler stands out as a comprehensive tool designed to enhance development productivity, optimize resource allocation, and mitigate the risks associated with production outages. This white paper explores the tangible benefits of investing in JProfiler, drawing a comparison with free profiling tools, and demonstrating how JProfiler justifies its cost through substantial time and money savings.

Introduction

Today's development environment demands the creation of robust and scalable applications with rapid development cycles. At the same time, cost-effective resource management is a principal focus for many organizations.

In this context, the **choice of profiling tools becomes critical**. The savings that organizations realize through the use of profilers primarily come from three categories:

Shortening development times

Lowering cloud costs

Reducing production Outages

We will examine these categories in detail and demonstrate how the savings quickly offset the licensing costs for JProfiler.

While free-of-charge profiling tools could be used for this purpose, they only offer basic capabilities. JProfiler provides a sophisticated suite of features designed to address a much broader range of challenges. Below, we will discuss the comparisons with the corresponding best-in-class tools that are free of charge.

Shortening Development Times

Focus on Business Logic

JProfiler empowers developers to concentrate on developing business logic rather than getting bogged down by performance issues. This focus significantly accelerates the development process, ensuring that projects meet their timelines without compromising on quality.

Without a profiling tool, pro-active optimization efforts during development and investigations into the performance characteristics of certain operations through the modification of code and the introduction of testing harnesses takes a considerable amount of time. With a profiling tool, this recurring activity can be reduced to a small fraction: Optimization efforts are only performed post-hoc on actual problems that are detected with JProfiler and not a priori on hypothetical issues derived from logical reasoning.

Performance and Memory Optimization

By pinpointing performance bottlenecks and memory leaks precisely where they occur, JProfiler ensures that optimization efforts are directed effectively, following the principle of addressing the most critical issues first. The benefits of using JProfiler include:

- Enhanced testing speed and more accessible profiling, thanks to its unparalleled integration into IDEs and build systems, and its support for profiling on remote machines, Docker containers and Kubernetes clusters.

- Early detection of unexpected performance issues and memory leaks, significantly improving the development process.
- Rapid identification of the causes behind performance problems and memory leaks, enabled by advanced data recording and analysis.

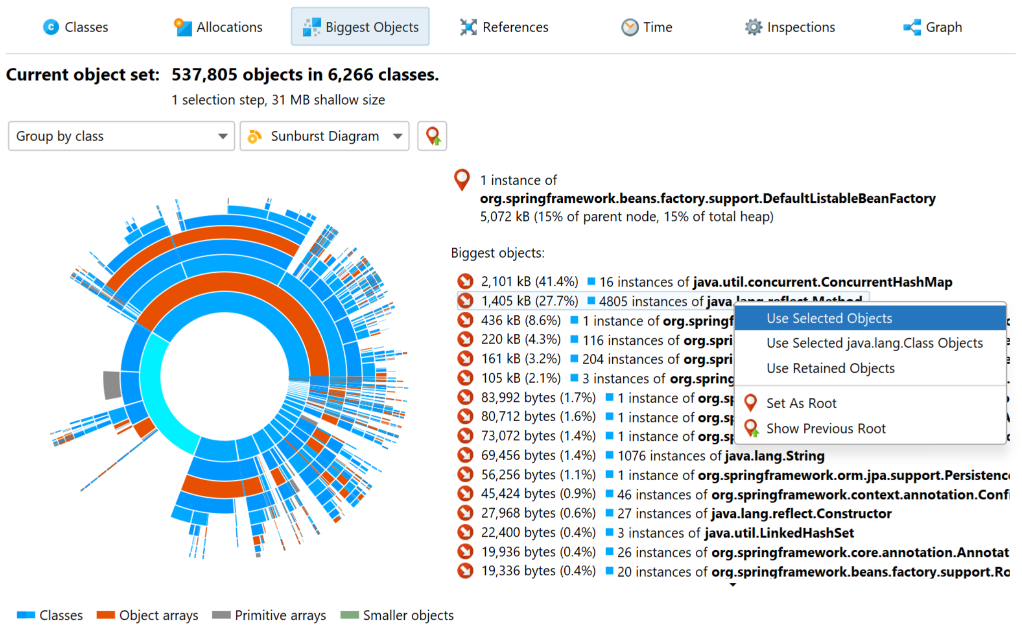
Example calculation for cost savings

We estimate a lower bound on these kinds of savings at an average of 30 minutes per developer per week. Considering developer salaries in industrialized countries, this translates to a **cost saving of over $1,000 per developer per year**. This alone amortizes the licensing cost of JProfiler in the first year, regardless of whether the following cost-saving topics apply to your production environment.

Lowering Cloud Costs

Optimizing Resource Usage

JProfiler aids in significantly reducing cloud expenses by enabling developers to optimize resource usage efficiently. Its comprehensive analysis capabilities help in understanding scalability issues, thereby facilitating the development of more efficient code that demands less CPU and memory resources.



Current object set: **537,805 objects in 6,266 classes.**
1 selection step, 31 MB shallow size

```
1 instance of
org.springframework.beans.factory.support.DefaultListableBeanFactory
5,072 kB (15% of parent node, 15% of total heap)

Biggest objects:
2,101 kB (41.4%)   16 instances of java.util.concurrent.ConcurrentHashMap
1,405 kB (27.7%)   4805 instances of java.lang.reflect.Method
436 kB (8.6%)      1 instance of org.springf
220 kB (4.3%)      116 instances of org.spri
161 kB (3.2%)      204 instances of org.spri
105 kB (2.1%)      3 instances of org.spring
83,992 bytes (1.7%)   1 instance of org.sp
80,712 bytes (1.6%)   1 instance of org.sp
73,072 bytes (1.4%)   1 instance of org.sp
69,456 bytes (1.4%)   1076 instances of java.lang.String
56,256 bytes (1.1%)   1 instance of org.springframework.orm.jpa.support.Persistenc
45,424 bytes (0.9%)   46 instances of org.springframework.context.annotation.Confi
27,968 bytes (0.6%)   27 instances of java.lang.reflect.Constructor
22,400 bytes (0.4%)   3 instances of java.util.LinkedHashSet
19,936 bytes (0.4%)   26 instances of org.springframework.core.annotation.Annotat
19,336 bytes (0.4%)   20 instances of org.springframework.beans.factory.support.Ro
```

🟦 Classes 🟧 Object arrays ▬ Primitive arrays 🟩 Smaller objects

With the extensive depth and breadth of JProfiler's features in the call tree and hot spot views, and a regular usage of JProfiler in the development process our customers routinely achieve significant reductions in CPU time by

- Optimizing hotspots
- Improving inefficient code paths
- Fixing algorithmic inefficiencies

With the help of the live memory views as well as the powerful feature set of JProfiler's heap walker, our customers reduce memory usage by

- Eliminating memory leaks
- Reducing unnecessary object creation
- Optimizing data structures

In total, we estimate that a development team using JProfiler can deliver applications that **consume 20%-50% less CPU time and 10% to 30% less memory**.

Reducing Data Transfer Rates

By minimizing unnecessary data exchanges, JProfiler helps in cutting down data transfer costs, which can be a significant portion of cloud expenses.

Insight into socket operations and HTTP connections becomes possible with JProfilers probes. Probes provide insight into high-level systems and are not present in simpler profilers. Both incoming and outgoing connections can be analyzed with JProfiler, For example:

- Incoming HTTP connections split the call tree with configurable granularity, so you can analyze requests separately
- Outgoing HTTP connections are shown in the views of the "HTTP client" probe, where information like URL hot spots, telemetries and single requests are available.
- The socket probe provides low-level information on data transfer and throughputs

Through the reduction of cacheable calls and the streamlining of communication in general, **reductions in data transfer of 10% to 30%** are obtainable when using JProfiler.

The total impact on cloud costs by reducing CPU times, memory consumption and data transfer is substantial. For organizations that have non-trivial cloud costs, these savings will massively exceed the investment of purchasing JProfiler licenses for their developers.

Reducing Production Outages

Production outages with JVM-based applications can take many forms: Crashes due to a lack of memory, unresponsive JVMs stuck in loops or deadlocks, single tasks or requests that don't perform as expected, and many other problems. The common theme is that these issues are not observable in development and staging environments and have to be investigated in the production environment. Having JProfiler at your disposable in these situations can be the difference between quick recovery and disaster.

In this respect, JProfiler's unique feature set stands out in three key areas: being able to attach to JVMs in all circumstances, getting high-level information for finding the cause of the problem, and analyzing post-mortem data from crashes.

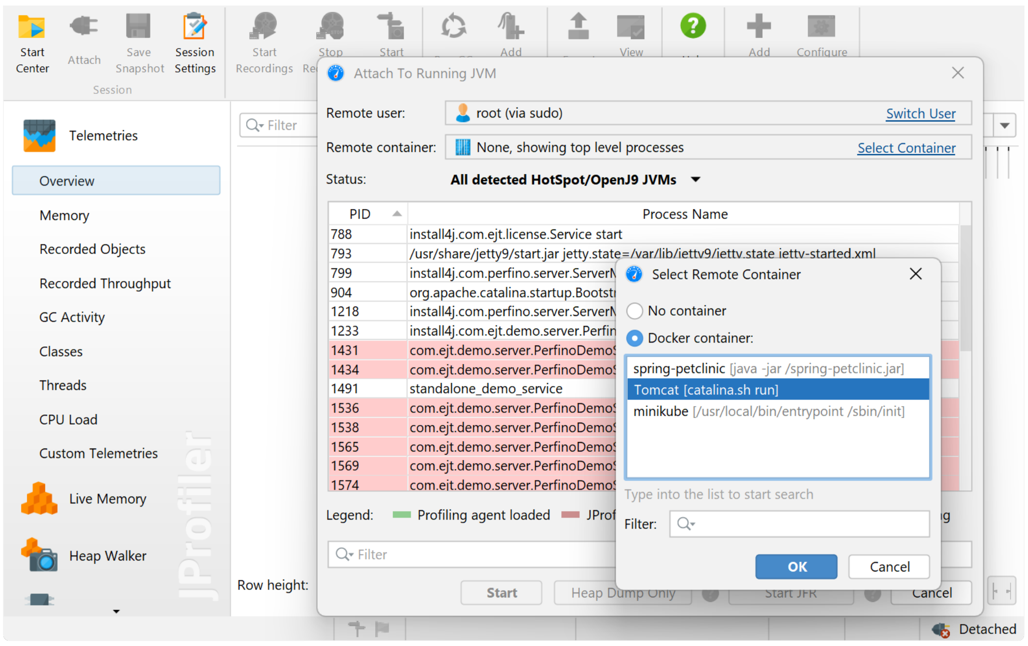
Advanced Production Profiling

JProfiler's remote attach functionality allows for profiling in production environments without introducing significant overhead. This capability is invaluable for diagnosing issues in real-time, thus minimizing the duration and impact of production outages.

A key capability which is required for this purpose is attaching to JVMs on remote machines. JProfiler supports "zero-configuration remote attach". We use this term because JProfiler

- Has a built-in SSH client that can be configured for multi-hop tunnels
- Automatically detects remote platforms, procures the necessary native agent libraries, and uploads them to the remote machine
- Uses a command line tool to scan the remote machine for running JVMs
- Presents these JVMs in the UI and allows you to start profiling immediately

In addition, production JVMs often run in Docker containers or on Kubernetes clusters. JProfiler has built-in functionality to look into Docker containers and attach to JVMs that run in containers managed by Kubernetes.



Attach To Running JVM

Remote user: root (via sudo) Switch User

Remote container: None, showing top level processes Select Container

Status: **All detected HotSpot/OpenJ9 JVMs**

PID	Process Name
788	install4j.com.ejt.license.Service start
793	/usr/share/jetty9/start.jar jetty.state=/var/lib/jetty9/jetty.state jetty-started.xml
799	install4j.com.perfino.server.ServerN
904	org.apache.catalina.startup.Bootstr
1218	install4j.com.perfino.server.ServerN
1233	install4j.com.ejt.demo.server.Perfin
1431	com.ejt.demo.server.PerfinoDemo
1434	com.ejt.demo.server.PerfinoDemo
1491	standalone_demo_service
1536	com.ejt.demo.server.PerfinoDemo
1538	com.ejt.demo.server.PerfinoDemo
1565	com.ejt.demo.server.PerfinoDemo
1569	com.ejt.demo.server.PerfinoDemo
1574	com.ejt.demo.server.PerfinoDemo

Legend: 🟩 Profiling agent loaded 🟥 JProf

Select Remote Container

No container [✓ Docker container:]

spring-petclinic [java -jar /spring-petclinic.jar]
Tomcat [catalina.sh run]
minikube [/usr/local/bin/entrypoint /sbin/init]

Type into the list to start search

Filter:

OK Cancel

Finally, JProfiler also comes with command line tools the let you prepare a JVM for profiling and control data recording and snapshot saving for profiled JVMs.

Debugging with Advanced Probes

The advanced probes in JProfiler offer deep insights into complex subsystems, enabling developers to understand and rectify issues swiftly.

Probes are a term that describes the semantic profiling features in JProfiler operating on a higher level than method-based profiling. They provide analytic access to performance-critical application layers that pure method call trees cannot provide. JProfiler's probes fall into three categories:

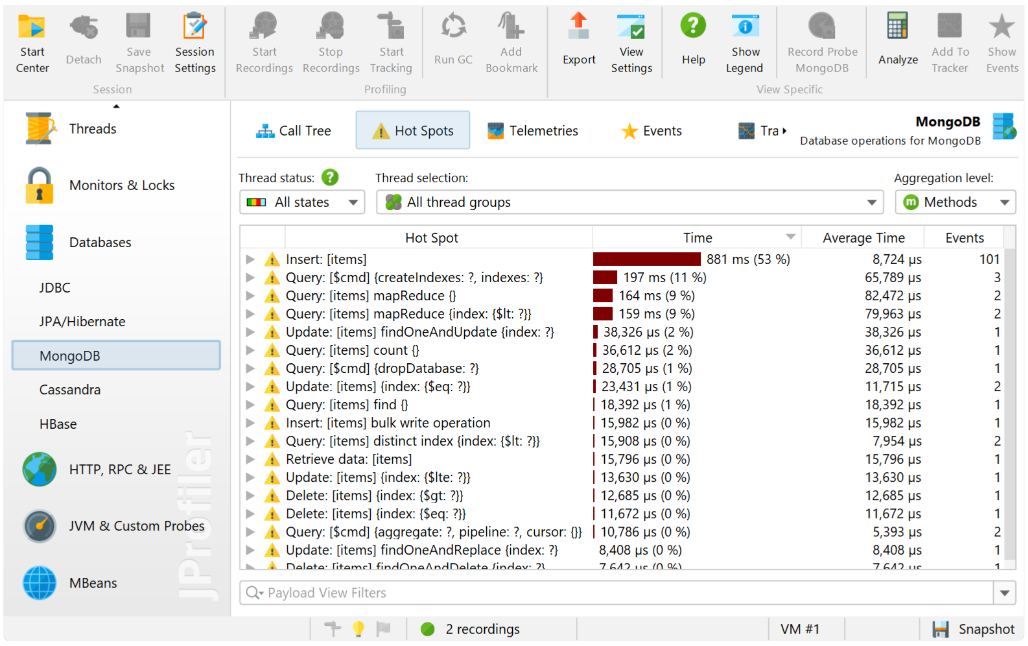
- **Databases**

  When database operations via JDBC, JPA, MongoDB or other databases are slow, the developer needs to see the offending SQL statements, JPA queries or MongoDB operations. Method-based hotspots are useless in this context because they do not isolate the cause of the problem.

- **Communication mechanisms**

  HTTP requests and other RPC mechanisms are also problematic in the same way. The URLs and connection parameters give clues on how to fix a problem, so they have to be available to the developer.

- **JVM subsystems**
  Other high-level data in the JVM needs to be visible in a fire-fighting operation, such as files, sockets and the communication with external processes.



MongoDB
Database operations for MongoDB

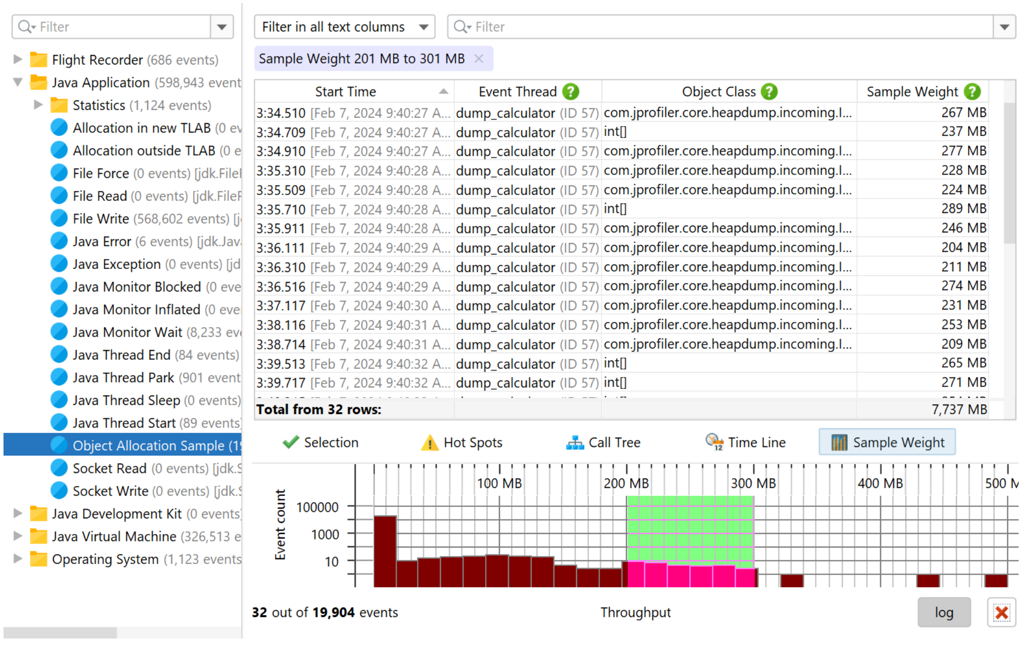
Thread status: All states
Thread selection: All thread groups
Aggregation level: Methods

Hot Spot	Time	Average Time	Events
Insert: [items]	881 ms (53 %)	8,724 µs	101
Query: [$cmd] {createIndexes: ?, indexes: ?}	197 ms (11 %)	65,789 µs	3
Query: [items] mapReduce {}	164 ms (9 %)	82,472 µs	2
Query: [items] mapReduce {index: {$lt: ?}}	159 ms (9 %)	79,963 µs	2
Update: [items] findOneAndUpdate {index: ?}	38,326 µs (2 %)	38,326 µs	1
Query: [items] count {}	36,612 µs (2 %)	36,612 µs	1
Query: [$cmd] {dropDatabase: ?}	28,705 µs (1 %)	28,705 µs	1
Update: [items] {index: {$eq: ?}}	23,431 µs (1 %)	11,715 µs	2
Query: [items] find {}	18,392 µs (1 %)	18,392 µs	1
Insert: [items] bulk write operation	15,982 µs (0 %)	15,982 µs	1
Query: [items] distinct index {index: {$lt: ?}}	15,908 µs (0 %)	7,954 µs	2
Retrieve data: [items]	15,796 µs (0 %)	15,796 µs	1
Update: [items] {index: {$lte: ?}}	13,630 µs (0 %)	13,630 µs	1
Delete: [items] {index: {$gt: ?}}	12,685 µs (0 %)	12,685 µs	1
Delete: [items] {index: {$eq: ?}}	11,672 µs (0 %)	11,672 µs	1
Query: [$cmd] {aggregate: ?, pipeline: ?, cursor: {}}	10,786 µs (0 %)	5,393 µs	2
Update: [items] findOneAndReplace {index: ?}	8,408 µs (0 %)	8,408 µs	1
Delete: [items] findOneAndDelete {index: ?}	7,642 µs (0 %)	7,642 µs	1

Post-mortem Analysis

JProfiler supports comprehensive post-mortem analysis for data saved with JFR (Java Flight Recorder), HPROF and PHD, providing a detailed overview of the system's state at the time of failure, which is crucial for preventing future outages.

Even if a JVM dies because of an OutOfMemoryError or some other crash, these built-in diagnostic tools can provide data for analysis. To successfully arrive at a swift conclusion, developers need a profiler that they are familiar with and a tool that provides them with advanced analysis features. JProfiler is the all-in-one solution that lets developers use the same skills for development profiling, in-production profiling and analysis of post-mortem data dumps.


- Flight Recorder (686 events)
- Java Application (598,943 event
  - Statistics (1,124 events)
  - Allocation in new TLAB (0 ev
  - Allocation outside TLAB (0 e
  - File Force (0 events) [jdk.FileF
  - File Read (0 events) [jdk.FileR
  - File Write (568,602 events) [j
  - Java Error (6 events) [jdk.Jav
  - Java Exception (0 events) [jd
  - Java Monitor Blocked (0 eve
  - Java Monitor Inflated (0 eve
  - Java Monitor Wait (8,233 ev
  - Java Thread End (84 events)
  - Java Thread Park (901 event
  - Java Thread Sleep (0 events)
  - Java Thread Start (89 events
  - Object Allocation Sample (1
  - Socket Read (0 events) [jdk.S
  - Socket Write (0 events) [jdk.S
- Java Development Kit (0 events
- Java Virtual Machine (326,513 e
- Operating System (1,123 events

Filter in all text columns

Sample Weight 201 MB to 301 MB

Start Time	Event Thread	Object Class	Sample Weight
3:34.510 [Feb 7, 2024 9:40:27 A...	dump_calculator (ID 57)	com.jprofiler.core.heapdump.incoming.I...	267 MB
3:34.709 [Feb 7, 2024 9:40:27 A...	dump_calculator (ID 57)	int[]	237 MB
3:34.910 [Feb 7, 2024 9:40:27 A...	dump_calculator (ID 57)	com.jprofiler.core.heapdump.incoming.I...	277 MB
3:35.310 [Feb 7, 2024 9:40:28 A...	dump_calculator (ID 57)	com.jprofiler.core.heapdump.incoming.I...	228 MB
3:35.509 [Feb 7, 2024 9:40:28 A...	dump_calculator (ID 57)	com.jprofiler.core.heapdump.incoming.I...	224 MB
3:35.710 [Feb 7, 2024 9:40:28 A...	dump_calculator (ID 57)	int[]	289 MB
3:35.911 [Feb 7, 2024 9:40:28 A...	dump_calculator (ID 57)	com.jprofiler.core.heapdump.incoming.I...	246 MB
3:36.111 [Feb 7, 2024 9:40:29 A...	dump_calculator (ID 57)	com.jprofiler.core.heapdump.incoming.I...	204 MB
3:36.310 [Feb 7, 2024 9:40:29 A...	dump_calculator (ID 57)	com.jprofiler.core.heapdump.incoming.I...	211 MB
3:36.516 [Feb 7, 2024 9:40:29 A...	dump_calculator (ID 57)	com.jprofiler.core.heapdump.incoming.I...	274 MB
3:37.117 [Feb 7, 2024 9:40:30 A...	dump_calculator (ID 57)	com.jprofiler.core.heapdump.incoming.I...	231 MB
3:38.116 [Feb 7, 2024 9:40:31 A...	dump_calculator (ID 57)	com.jprofiler.core.heapdump.incoming.I...	253 MB
3:38.714 [Feb 7, 2024 9:40:31 A...	dump_calculator (ID 57)	com.jprofiler.core.heapdump.incoming.I...	209 MB
3:39.513 [Feb 7, 2024 9:40:32 A...	dump_calculator (ID 57)	int[]	265 MB
3:39.717 [Feb 7, 2024 9:40:32 A...	dump_calculator (ID 57)	int[]	271 MB
Total from 32 rows:			7,737 MB

32 out of **19,904** events

As analyzed elsewhere [(1)] the cost of an hour of downtime can be considerable, even millions of dollars for large companies. By having developers who are familiar with JProfiler, **downtimes can be reduced to a fraction**. In these scenarios, the investment into JProfiler is tiny compared to the damage that is prevented with it.

Comparison With Free Profiling Tools

While free profiling tools provide basic functionality, they often fall short in offering the depth of analysis, ease of use, and comprehensive features found in JProfiler.

Comparison to Built-In Tools

Java comes with Java Flight Recorder (JFR), a built-in diagnostic tool that has some overlap with a profiler. JFR is primarily a structured logging tool, best suited for low-overhead data recording for post-mortem analysis. Its large recording granularity means that it cannot generally be used as a CPU profiler. Similarly, using HPROF dumps with free-of-charge, but generally hard-to-use display tools like MAT only provides limited analysis, because there is no way to analyze live data.

Sometimes, the only available data is from these built-in data dumps. JProfiler fully supports the corresponding file formats and lets developers use its advance analysis capabilities on them, all in the same UI that developers are already familiar with. Whenever possible, the full feature set of the JProfiler agent can be used for maximum insight with regular profiling sessions.

Comparison to Free-Of-Charge Profilers

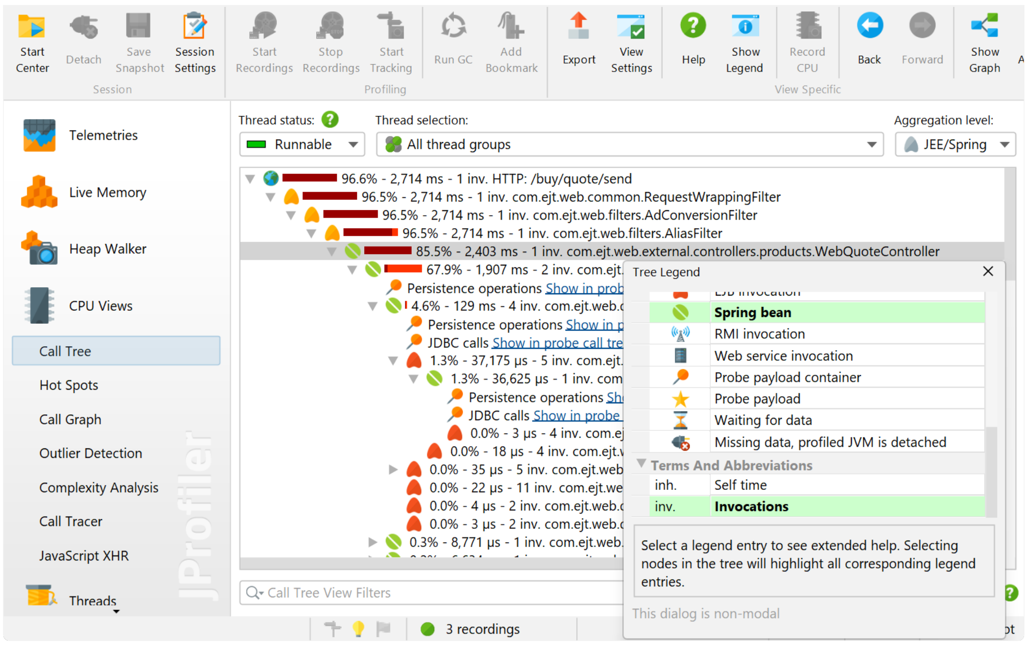
While you could use free-of-charge profilers to realize some parts of the savings discussed in this whitepaper, they will be much smaller due to the much smaller feature set. Discussing all features that make JProfiler stand out would be a very long list and involve many areas in its feature set. For the purpose of this whitepaper, we showcase the call tree, a view that is the core of CPU profilers and that is often used by developers on a daily basis.

(1) https://www.pingdom.com/outages/average-cost-of-downtime-per-industry/

JProfiler's unparalleled capabilities related to call trees and hot spots give developers the necessary tool to understand where optimization work should be focussed. For example, this includes features like

- Removing selected parts of the call tree
- Setting a selected node in the call tree as the new root
- Collapsing recursions in the call tree
- Calculating cumulated outgoing calls and backtraces to selected methods
- Displaying the call tree as an interactive graph or as an interactive flame graph
- Inlining async executions
- Splitting the call tree on selected methods with a script
- Recording exceptional method runs for selected methods
- Defining ad-hoc probes with scripts to create custom analyzes

The above functionality is not present in free-of-charge profilers like VisualVM or Aysnc Profiler and is just a small list of features that put JProfiler on a different level.



Thread status: Runnable

Thread selection: All thread groups

Aggregation level: JEE/Spring

```
96.6% - 2,714 ms - 1 inv. HTTP: /buy/quote/send
  96.5% - 2,714 ms - 1 inv. com.ejt.web.common.RequestWrappingFilter
    96.5% - 2,714 ms - 1 inv. com.ejt.web.filters.AdConversionFilter
      96.5% - 2,714 ms - 1 inv. com.ejt.web.filters.AliasFilter
        85.5% - 2,403 ms - 1 inv. com.ejt.web.external.controllers.products.WebQuoteController
          67.9% - 1,907 ms - 2 inv. com.ejt.
            Persistence operations Show in prob
            4.6% - 129 ms - 4 inv. com.ejt.web.c
              Persistence operations Show in p
              JDBC calls Show in probe call tre
              1.3% - 37,175 µs - 5 inv. com.ejt.
                1.3% - 36,625 µs - 1 inv. com
                  Persistence operations Sh
                  JDBC calls Show in probe
                  0.0% - 3 µs - 4 inv. com.ej
                0.0% - 18 µs - 4 inv. com.ejt.v
            0.0% - 35 µs - 5 inv. com.ejt.web
            0.0% - 22 µs - 11 inv. com.ejt.we
            0.0% - 4 µs - 2 inv. com.ejt.web.c
            0.0% - 3 µs - 2 inv. com.ejt.web.c
          0.3% - 8,771 µs - 1 inv. com.ejt.web.
```

Tree Legend

	EJB invocation
	Spring bean
	RMI invocation
	Web service invocation
	Probe payload container
	Probe payload
	Waiting for data
	Missing data, profiled JVM is detached

Terms And Abbreviations

inh.	Self time
inv.	**Invocations**

Select a legend entry to see extended help. Selecting nodes in the tree will highlight all corresponding legend entries.

This dialog is non-modal

The Value of Investing in the Right Tools for High-Stakes Environments

Imagine two mechanics tasked with repairing and optimizing a fleet of high-performance race cars. One mechanic has a standard set of hand tools. These tools are reliable for basic tasks and straightforward repairs but lack the precision and efficiency needed for diagnosing and fixing the complex issues that high-performance race cars often present. This mechanic can handle common problems but struggles with more complicated issues, taking longer to diagnose problems and sometimes missing the optimal solution altogether.

On the other hand, the second mechanic is equipped with a state-of-the-art diagnostic machine, similar to a developer armed with JProfiler. This advanced toolset allows the mechanic to quickly and accurately diagnose even the most complex issues, that would be time-consuming or even impossible to detect with basic tools. The high-tech equipment guides the mechanic to the exact problem area, suggests the best repair methods, and helps optimize the car's performance beyond its initial state.

The same scenario plays out on the JVM, where the developer with JProfiler not only fixes all problems faster but is also capable of solving complex issues than developers who only have access to free-of-charge profiling tools might not be able to handle.

Conclusion

JProfiler represents a strategic investment for development teams aiming for high efficiency, reduced operational costs, and minimized downtime. Its advanced features, focusing on critical aspects of modern software development, offer significant advantages over free profiling tools. The cost of JProfiler is not only justifiable, but is quickly amortized, delivering value that far exceeds its price tag.

In the fast-evolving world of software development, equipping teams with JProfiler is a decision that pays dividends in accelerated development cycles, reduced cloud costs, and enhanced application reliability.