SP Camp: Performance Measurement, Evaluation and Analysis

Due on Jul 9, 2024

Li Chi

Content

1. 概要

本报告总结了在项目中三个任务的成果，涵盖了 SPECjvm2008 基准测试的使用、基于基准测试的多 JVM 对比分析以及 Java 程序性能分析。并通过一些简单的分析揭示数据背后的性能特征和优化方向。

为了确保项目的可复现与结果的可靠性，每个任务下都有详细的实验方法过程记录与详细说明，并且已经将所有代码和数据上传至 GitHub 仓库，确保其他研究者可以复现或者进一步改进我们的项目。

2. 实验环境

本报告均在眞机上完成，实验环境如下：

2.1 硬件配置

- 处理器：12th Gen Intel(R) Core(TM) i5-12500H
- 内存：16GB RAM

2.2 软件环境

- 操作系统：Ubuntu 24.04.1 LTS
- **Java** 运行环境：
  - OpenJDK 8 （System-Default）
  - 支持多种 JDK 版本：毕昇 JDK、龙井 JDK、腾讯 Kona JDK8
- 性能分析工具：
  - `perf`：Linux 内核性能分析工具
  - FlameGraph：火焰图生成工具
  - perf-map-agent：Java 符号映射工具

3. Assignment 1: Software Performance Measurement

在该任务中，我下载、安装并运行 SPECjvm2008 标准基准测试套件对 OpenJDK 8 的性能进行了初步的评估。为了解决手工输入命令的繁琐性，我编写了一个 Shell 脚本来自动化执行测试，并将结果输出到指定文件中。该脚本确保了测试的可重复性和结果的可靠性。

3.1 使用 SPECjvm2008

SPECjvm2008 是一个广泛使用的 Java 性能基准测试套件，旨在评估 Java 虚拟机（JVM）的性能。该测试套件包含多个基准测试，涵盖了不同的计算任务和场景，能够全面反映 JVM 在实际应用中的性能表现。

在完成该测试套件的安装配置后，我编写了一个 Shell 脚本来自动化执行测试。该脚本的主要功能包括：

- 自动加载配置文件，可自定义测试参数
- 逐个运行指定的 SPECjvm2008 基准测试工作负载，并将结果输出到指定的文件路径中

- 记录每个测试的测试日志，便于检查
- 生成总结报告，包括每个测试的得分、系统信息和 JDK 信息

以下是运行脚本后的输出示例：

```
SPECjvm2008 自动测试汇总
时间: Sat Jul  5 06:40:41 PM CST 2025

=== 系统信息 ===
JRE 版本:
openjdk version "1.8.0_452"
OpenJDK Runtime Environment (build 1.8.0_452-8u452-ga~us1-0ubuntu1~24.04-b09)
OpenJDK 64-Bit Server VM (build 25.452-b09, mixed mode)

操作系统:
PRETTY_NAME="Ubuntu 24.04.1 LTS"

CPU 信息:
CPU(s):                              16
On-line CPU(s) list:                 0-15
Model name:                          12th Gen Intel(R) Core(TM) i5-12500H
Thread(s) per core:                  2
Core(s) per socket:                  12
Socket(s):                           1
CPU(s) scaling MHz:                  47%
NUMA node0 CPU(s):                   0-15
```

Figure 1: SPECjvm2008 基准测试系统信息汇总

从图中可以看出，JRE 的版本是 OpenJDK 8，操作系统的版本是 Ubuntu 24.04.1 LTS，处理器为 12th Gen Intel(R) Core(TM) i5-12500H，每个核的线程数为 2 等。

该报告还包含了每个工作负载的得分：

```
[compiler.compiler] 分数: 902.20 ops/m

[crypto] 分数: 699.93 ops/m

[scimark.fft.small] 分数: 599.63 ops/m

[startup.helloworld] 分数: 410.96 ops/m

[scimark.monte_carlo] 分数: 482.69 ops/m

[sunflow] 分数: 142.79 ops/m
```

Figure 2: OpenJDK 8 在各工作负载下的性能得分

从图中可以看出 OpenJDK 在不同工作负载下的性能表现存在差异，在 compiler.compiler 工作负载下得分最高，达到了 902.20 ops/m，在 sunflow 工作负载下得分最低，仅为 142.79 ops/m。这些结果表明，OpenJDK 在处理编译相关任务时表现较好，而在图形渲染等任务上则相对较弱。

3.2 问题回答

经过对 SPECjvm2008 基准测试套件的运行和结果分析，以及查阅测试套件相关文档，我对以下问题进行回答

3.2.1 What is the performance metric of SPECjvm2008? Why? What are the units of measurements?

SPECjvm2008 的性能度量指标是每分钟操作数（ops/m），它通过测量 Java 虚拟机在执行特定基准测试时的吞吐量来评估性能。该指标的单位是操作数每分钟（ops/m），反映了 JVM 在处理 Java 程序时的性能表现。

SPECjvm2008 侧重于评价 JRE 执行单个 java 应用程序时的性能，其中的工作负载是真实世界的 java 程序的一部分。执行工作负载不仅反映 JVM 的性能，同时也反映了执行 JVM 的操作系统和 CPU，内存子系统的性能。基于该 benchmark 的目标，同时为了便于不同平台之间进行对比，将度量粒度定为工作负载是合理的。这种度量反映了系统整体处理 Java 程序的效率，涵盖了 JVM 和系统支持两方面。并且标准化的工作负载与统一的运行规则，让不同系统之间的分数具有可比性。

3.2.2 What factors affect the scores? Why some get higher scores, but others get lower scores?

影响 SPECjvm2008 基准测试得分的因素主要包括以下几个方面：

- JVM 实现：不同的 JVM 实现（如 OpenJDK、毕昇 JDK、龙井 JDK 等）在性能优化和资源管理方面存在差异，这会直接影响基准测试的得分。
- JVM 参数配置：JVM 的启动参数和内存配置会影响垃圾回收策略和内存管理，从而影响性能得分。如果按照 SPECjvm2008 的要求配置 JVM 参数，可以获得更高的得分。
- 工作负载特性：不同的工作负载具有不同的计算复杂度和资源需求。例如，编译相关的工作负载通常需要更多的 CPU 资源，而图形渲染等任务可能对内存和 GPU 资源有更高的要求。
- 系统配置：CPU 的主频和架构直接影响机器指令的执行效率，指令执行越快，得分越高；核心数量越多，能够并行处理的线程就越多，从而提升整体吞吐能力。SPECjvm2008 的许多基准测试是多线程的，JVM 会根据系统的逻辑线程数自动设定工作线程数，因此线程越多，并发性能越强，得分也会越高。
- 缓存：缓存系统的容量（如 L2/L3 Cache）会显著影响程序运行速度。较大的缓存可以容纳更多热点数据和代码，减少访问主存的频率，从而降低延迟，提高性能。

3.2.3 Why is warmup required in SPECjvm2008, and does warmup time have any impact on performance test results?

在 SPECjvm2008 中，预热是一个重要的步骤，旨在使 JVM 达到稳定的性能状态。预热的主要目的是让 JVM 充分利用即时编译（JIT）优化和其他性能提升机制，从而在正式测试阶段提供更准确的性能数据。

预热时间对性能测试结果有显著影响。以下是预热的几个关键作用：

- JIT 编译：预热阶段允许 JVM 识别和编译热点代码，从而在正式测试时提供更高的执行速度。
- 垃圾回收优化：在预热阶段，JVM 可以进行多次垃圾回收操作，优化内存管理策略。这样可以减少在正式测试阶段的垃圾回收停顿时间，提高整体吞吐量。
- 缓存利用：预热阶段可以使 JVM 加载必要的类和资源到内存中，从而提高缓存命中率，减少后续测试阶段的内存访问延迟。

如果没有足够的预热时间，JVM 可能在正式测试阶段仍处于优化过程中，这会导致性能数据不准确，无法反映 JVM 在稳定状态下的真实性能。因此，预热时间的设置对于获得可靠的性能测试结果至关重要。

3.2.4 Did you get close to 100% CPU utilization running SPECjvm2008? Why or why not?

在运行特定负载时，我通过 linux 系统的 System Monitor 软件观察到 CPU 使用率达到了 100%，并且在绝大多数 CPU 密集型测试中，CPU 的使用率都在 100% 附近。原因是 SPECjvm2008 会根据系统的逻辑线程数量分配工作线程数量，并且很多工作负载都是可以并行执行的，没有显著的 I/O 和内存瓶颈，可以充分利用 CPU 资源。

在绝大部分时间，CPU 的使用率并没有达到 100%，这可能是由于 JVM 需要执行垃圾回收、线程调度等操作，这些操作会导致 CPU 使用率下降。此外，某些工作负载可能存在 I/O 瓶颈或其他资源限制，导致 CPU 无法持续满负荷运行。

4. Assignment 2:Software Performance Evaluation

在该任务中，我实现了一个完整的 SPECjvm2008 自动化基准测试系统，支持多种 JDK 的自动化性能测试，将 JDK 的性能表现可视化（柱状图和箱线图），并使用配对 t 检验等统计方法对 JDK 之间的性能差异进行显著性检验。

4.1 实现 SPECjvm2008 自动化基准测试

为了便于调用不同 JDK 版本进行基准测试，我将所有 JDK 放在了一个名为 `jdk` 的目录下，并沿用 Assignment1 的运行 workload 脚本的结构，进行一些改进。这包括增加脚本参数，支持指定 JDK 版本的选择运行。因为要对所有 JDK 版本进行对比测试，所以我使用一个 `run_all.sh` 脚本来自动化执行所有 JDK 的测试。该脚本会遍历 `jdk` 目录下的所有 JDK 版本，并对每个版本运行相同的 SPECjvm2008 基准测试。运行后的结果会被保存到 output 下中子目录，该子目录根据运行时间命名，便于区分不同的测试轮次。

以下是运行脚本后的输出目录示例：

```
output/                          # 测试结果目录
└── run_YYYYMMDD_HHMMSS/         # 按时间戳命名的运行结果
    ├── bisheng-jdk/             # 毕昇JDK测试结果
    │   ├── log_*.txt            # 各工作负载日志文件
    │   └── SPECjvm2008.*/       # 详细测试数据
    ├── dragonwell/              # 龙井JDK测试结果
    ├── openjdk/                 # OpenJDK测试结果
    ├── TencentKona/             # 腾讯Kona JDK测试结果
    └── all_jdks_summary.txt     # 所有JDK汇总报告
```

Listing 1: 多 JDK 测试输出目录结构

在运行脚本后，所有 JDK 的测试结果会被汇总到一个名为 `all_jdks_summary.txt` 的文件中。该文件包含了每个 JDK 在各个工作负载下的性能得分和 JDK 信息。

```
=== bisheng-jdk1.8.0_452 测试结果 ===
[compress] 分数: 278.78 ops/m
测试耗时: 1377秒

=== dragonwell-8.25.24 测试结果 ===
[compress] 分数: 275.33 ops/m
测试耗时: 1375秒

=== TencentKona-8.0.22-452 测试结果 ===
[compress] 分数: 279.58 ops/m
测试耗时: 1373秒

=== system-default 测试结果 ===
[compress] 分数: 272.87 ops/m
测试耗时: 1375秒
```

Figure 3: 不同 JDK 在 compress 工作负载下的性能得分汇总

4.2 可视化

为了更好地展示不同 JDK 在各个工作负载下的性能表现，我使用了 Python 的 Matplotlib 库生成了柱状图和箱线图。这些图表直观地展示了各个 JDK 在同一工作负载下的性能得分分布和差异。

Python 脚本会读取每个 JDK 在工作负载下的每轮迭代测试得分，并计算多轮测试的平均值作为最终得分，避免单次测试结果的偶然性对整体评估的影响。利用这些数据，脚本会生成柱状图和箱线图，展示各个 JDK 在不同工作负载下的性能得分分布。

柱状图展示了各个 JDK 在同一工作负载下的平均得分，而箱线图则展示了得分的分布情况，包括中位数、四分位数和异常值。为了使柱状图突出不同 JDK 之间的差异，图表的 y 轴部分会根据工作负载的得分进行自动调整。

以下是一个示例柱状图与箱线图，展示了不同 JDK 在 compress 工作负载下的性能得分与分布：

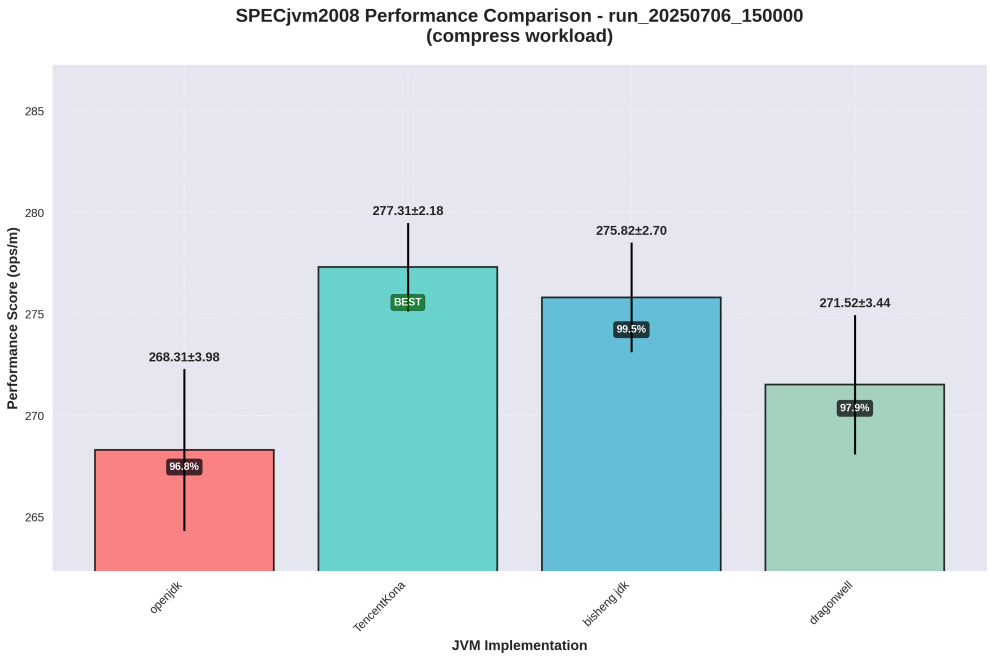

Figure 4: 不同 JDK 的性能得分柱状图

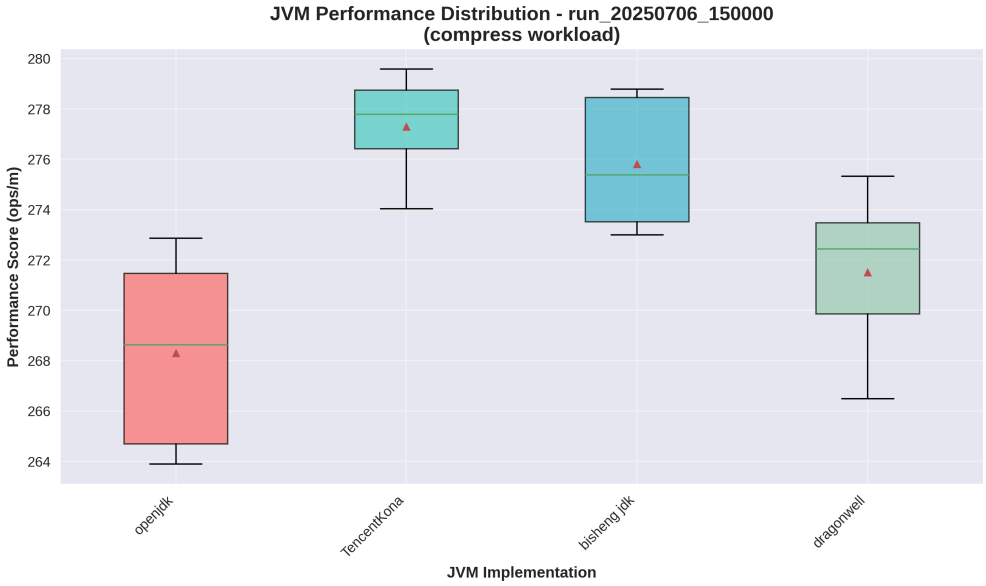

Figure 5: 不同 JDK 的性能得分箱线图

从图中可以看出，不同 JDK 在 compress 工作负载下的性能得分存在明显差异。OpenJDK 在该工作负载下表现最差，腾讯 Kona JDK 则表现最佳，并且在得分的稳定性和一致性方面也优于其他 JDK。

4.2.1 统计分析

为了进一步分析不同 JDK 在各个工作负载下的性能差异是否显著，我主要使用了配对 t 检验统计方法。

在进行该检验前，为了达到该方法的前提条件，我首先对每个 JDK 的性能得分（以下简称样本）进行了正态性检验，确保数据符合正态分布。接着，我对样本进行单因素方法分析（ANOVA），检查样本之间是否存在显著差异。一旦存在，继续使用配对 t 检验来比较不同 JDK 之间的性能差异。由于配对 t 检验需要进行多次成对比较，因此我使用了 Bonferroni 校正方法来控制多重比较引起的第一类错误率。通过这些统计方法，我能够判断不同 JDK 在各个工作负载下的性能差异是否显著，并为进一步的性能优化提供依据。

以上过程均可以通过 Python 脚本实现，脚本会自动读取每个 JDK 在同一工作负载下的性能得分，并进行统计分析。最终结果会以文本形式输出，展示不同 JDK 之间的性能差异及其显著性水平。

继续沿用本节的数据，以下是一个示例统计分析结果，展示了不同 JDK 在 compress 工作负载下的性能差异：

```
工作负载: compress
分析时间: 2025-07-06 21:17:16

基本统计信息:
----------------------------------------
  openjdk: 5 样本, 均值=268.31, 标准差=3.98
  TencentKona: 5 样本, 均值=277.31, 标准差=2.18
  bisheng-jdk: 5 样本, 均值=275.82, 标准差=2.70
  dragonwell: 5 样本, 均值=271.52, 标准差=3.44

正态性检验 (Shapiro-Wilk, α=0.05):
----------------------------------------
  openjdk: p=0.4723, 正态分布
  TencentKona: p=0.7441, 正态分布
  bisheng-jdk: p=0.2624, 正态分布
  dragonwell: p=0.8343, 正态分布

单因素方差分析 (ANOVA):
----------------------------------------
  F统计量: 8.4816
  p值: 0.001334
  结论: 有显著差异
```

Listing 2: 不同 JDK 在 compress 工作负载下的统计分析结果

从统计分析结果可以看出，不同 JDK 在 compress 工作负载下的性能得分存在显著差异。ANOVA 检验的 p 值小于 0.05，表明至少有两个 JDK 之间的性能差异是显著的。接下来，可以使用配对 t 检验进一步比较各个 JDK 之间的具体差异。

从各 JDK 的均值来看，腾讯 Kona JDK 的性能得分最高，其次是毕昇 JDK 和龙井 JDK，而 OpenJDK 的性能得分最低。这些结果表明，腾讯 Kona JDK 在 compress 工作负载下表现最佳。从标准差来看，各 JDK 的得分波动较小，说明在多轮测试中，各 JDK 的性能表现相对稳定。

以下是使用配对 t 检验与 Bonferroni 校正后的结果示例：

```
配对t检验:
----------------------------------------
  openjdk vs TencentKona: t=-8.8505, p=0.000900, 显著
  openjdk vs bisheng-jdk: t=-12.1258, p=0.000265, 显著
  openjdk vs dragonwell: t=-4.1563, p=0.014187, 显著
  TencentKona vs bisheng-jdk: t=2.4519, p=0.070300, 不显著
  TencentKona vs dragonwell: t=10.1789, p=0.000525, 显著
  bisheng-jdk vs dragonwell: t=5.6117, p=0.004955, 显著

Bonferroni多重比较校正:
----------------------------------------
  原始α: 0.050
  比较次数: 6
  校正后α: 0.008333
  校正后结果:
    openjdk vs TencentKona: p=0.000900, 显著
    openjdk vs bisheng-jdk: p=0.000265, 显著
    openjdk vs dragonwell: p=0.014187, 不显著
    TencentKona vs bisheng-jdk: p=0.070300, 不显著
    TencentKona vs dragonwell: p=0.000525, 显著
    bisheng-jdk vs dragonwell: p=0.004955, 显著
```

Listing 3: 不同 JDK 在 compress 负载下的检验结果

从配对 t 检验结果可以看出，OpenJDK 与其他 JDK 之间的性能差异均显著，而腾讯 Kona JDK 与毕昇 JDK 之间的差异不显著。这表明 OpenJDK 在 compress 工作负载下的性能明显低于其他 JDK，而腾讯 Kona JDK 和毕昇 JDK 的性能差异较小。校正后的结果大部分保持了显著性，更加有力地支持了得出的结论。

4.3 问题回答

经过系统地对不同 JDK 在同一工作负载下的性能进行分析后，我对以下问题进行回答

4.3.1 Why is there run to run performance variation?

这种情况由多种原因导致。每次运行时，CPU 核心、内存、缓存等资源存在不一致性，开始运行同一工作负载时计算机这个状态机的起始状态不一致;在多核场景下，不同线程在不同核心上运行，由于系统中不止有一个进程，所以会与其他进程进行竞争;JVM 内部的 JIT 编译的时机不同。JIT 越早识别热点代码并进行编译优化，运行时间就会减少，从而导致性能表现不一致;每次运行时缓存的内部情况不同，运行时 hit 数量有所差异，导致性能表现不一致。

4.3.2 What contributes to run-to-run variation?

每次运行测试时，系统中会有其他进程（系统日志，系统更新等），这些进程的存在会影响操作系统对任务的调度，导致不同运行之间会有不同的调度情况以及上下文切换次数，这些额外的调度与上下文切换会占用运行时间，进而影响性能表现。同时在多核 CPU 上，被分配到同一个核上的不同线程会竞争缓存资源，发生原本属于工作负载的一些缓存内容被替换，导致运行时发生的 cache miss 情况增多，影响性能表现。CPU 具有根据系统负载调整处理器频率的动态频率调整策略，导致不同运行之间 CPU 的频率不完全一致，工作负载运行速度不一致，产生性能表现不同的情况。

4.3.3 How do we validate the factors contributing to run-to-run variation?

可以采用控制变量法，每次只固定一个影响因素。例如使用 `taskset` 绑定固定 CPU 核心运行测试,将其与对照组的数据做统计假设检验，验证该因素是否对运行的性能表现有显著性影响。

4.3.4 What are the pros and cons of using arithmetic mean versus geometric mean in summarizing scores?

在总结性能得分时，算术平均数和几何平均数各有优缺点。算数平均数是所有样本的得分之和除以样本数，它的计算方式简单，容易理解。但不同场景下的分数差异可能会较大，算数平均数会受到极端值的较大影响，导致最终平均值和极端值相近，不能反映较小值的特性，不适用跨越多个数量级的数值之间计算平均数。

几何平均数是所有得分的乘积开 n 次方,在这种计算方式下，各个得分的权重相等，避免了高分测试项占据主导地位，更能反映多个子基准的相对性能变化趋势。缺点是对负数值不适用，反映的是相对水平，计算方式复杂。

4.3.5 Why does SPECjvm2008 use geometric mean? (In fact, it uses hierarchical geometric mean)

在不同的子基准测试中得分差异往往较大，差距有时在多个数量级上。若使用算数平均数，会导致得分基本由高分子基准主导，导致整体性能被误判。因此使用几何平均数能够更好地反映不同基准测试之间的相对性能差异。由于在 SPECjvm2008 中每个种类的工作负载数量不同，为避免同一种类的工作负载在最终的得分中占据过大权重，平等地反映各个应用领域的表现，所以先对每个种类的工作负载求出组内几何平均值作为结果，再将各个组的结果求出几何平均值作为最终得分，也就是分层几何平均。

5. Assignment 3: Software Performance Analysis

在该任务中，我使用了 perf 工具对 Java 程序的性能进行了深入分析。

为了找到 Java 程序的性能瓶颈，我使用了 perf 的 stat 功能观察整体性能指标，包括 CPU 使用率、内存占用、I/O 情况等。接着，我使用 perf 的 record 功能对 Java 程序进行采样，收集函数调用栈信息，并生成火焰图来可视化性能瓶颈。由于 Java 程序的符号信息通常被编译器优化掉了，因此我使用了 perf-map-agent 工具来生成符号映射文件，以便 perf 能够正确解析 Java 函数调用栈。最后，我将 perf script 的输出结果导入数据库中，并进行一定的分析。

5.1 perf 观察总体性能

在使用 perf 工具对 Java 程序进行性能分析时，首先需要观察整体性能指标，以便了解程序的运行情况。通过 perf 的 stat 功能，可以获取以下几个关键指标：CPU 使用率、上下文切换次数、缓存命中率、分支预测命中率和 CPI（每周期指令数）。这些指标可以帮助我们了解程序在运行时的性能表现，并识别潜在的性能瓶颈。

以使用递归方法计算斐波那契数列为例，以下是 perf stat 命令的输出部分结果：

```
Performance counter stats for 'java TestFibonacci':

         90,779.38 msec  task-clock                   #   1.008 CPUs utilized
            11,957       context-switches             # 131.715 /sec
             1,726       cpu-migrations               #  19.013 /sec
            22,906       page-faults                  # 252.326 /sec
   210,622,702,731       cpu_atom/cycles/             #   2.320 GHz                (0.93%)
   321,817,783,998       cpu_core/cycles/             #   3.545 GHz               (98.87%)
   288,753,679,694       cpu_atom/instructions/       #   1.37  insn per cycle     (1.09%)
   948,031,881,267       cpu_core/instructions/       #   4.50  insn per cycle    (98.87%)
    59,233,886,135       cpu_atom/branches/           # 652.504 M/sec             (1.10%)
   195,241,038,466       cpu_core/branches/           #   2.151 G/sec            (98.87%)
       470,867,687       cpu_atom/branch-misses/      #   0.79% of all branches   (1.09%)
       115,698,101       cpu_core/branch-misses/      #   0.20% of all branches  (98.87%)
            TopdownL1 (cpu_core)                    #   1.8 %  tma_backend_bound
                                                    #   1.2 %  tma_bad_speculation
                                                    #  49.3 %  tma_frontend_bound
                                                    #  47.7 %  tma_retiring      (98.87%)
            TopdownL1 (cpu_atom)                    #   6.5 %  tma_bad_speculation
                                                    #  33.3 %  tma_retiring       (1.10%)
                                                    #  22.2 %  tma_backend_bound
                                                    #  22.2 %  tma_backend_bound_aux
                                                    #  37.9 %  tma_frontend_bound (1.12%)

      90.041368662 seconds time elapsed

      90.614939000 seconds user
       0.216947000 seconds sys
```

Listing 4: TestFibonacci 程序的 perf 性能统计结果

5.2 perf 产生火焰图

在获取了整体性能指标后，接下来需要深入分析 Java 程序的性能瓶颈。为此，我使用了 perf 的 record 功能对 Java 程序进行采样，收集函数调用栈信息。

perf record 命令会在指定的 Java 程序运行期间按一定频率采样 CPU 的调用栈信息，并将结果保存到一个 perf.data 文件中。由于 Java 程序执行过程中会被 JIT 编译为本地代码，从而丢失符号信息，为了能够正确解析 Java 函数调用栈，我使用了 perf-map-agent 工具来生成符号映射文件。该工具会在 Java 程序运行时动态地生成符号映射信息，以便 perf 能够正确解析 Java 函数名。然后使用 perf script 命令将 perf.data 文件转换为可读的文本格式。最后调用 FlameGraph 工具生成火焰图，以可视化性能瓶颈。

该过程可通过 shell 脚本自动完成。

以下是对 TestFibonacci 程序进行 perf record 后的火焰图示例：

Figure 6: TestFibonacci 程序的火焰图

从火焰图中可以看出，TestFibonacci 程序中递归函数 `fibonacci` 占据了图中顶层的宽度的大部分，表明该函数在程序运行中占用了大量的 CPU 时间。这是因为递归调用会导致大量的函数调用和返回，增加了 CPU 的负担。如果要进行性能优化，可以考虑使用迭代方法来替代递归方法，减少函数调用的开销。

5.3 数据库分析

在使用 perf 工具对 Java 程序进行性能分析后，我将 perf script 的输出结果导入数据库中，并进行一定的分析。通过对函数调用栈信息的分析，可以识别出程序中的性能瓶颈和热点函数。数据库包括 `perf_samples`、`call_stacks` 以及 `metadata` 表，分别记录采样信息，调用栈信息和元数据。

利用数据库信息可以查询热点函数信息，识别出占用 CPU 时间最多的函数，并分析其调用关系。以下是一个示例查询，展示了 TestFibonacci 程序中占用 CPU 时间最多的函数：

```
=== 数据库基本信息 ===
程序名称: TestFibonacci
采集时间: 60 秒
样本总数: 6861
调用栈记录: 29282

=== 进程信息分析 ===
进程名                    样本数       占比%
----------------------------------------
java                 5948     86.69
GC task thread#      729      10.63
C2 CompilerThre      94       1.37
C1 CompilerThre      33       0.48
VM Periodic Tas      30       0.44
VM Thread            27       0.39

=== Java 热点函数分析 (Top 10) ===
排名   Java方法                                                                    调用次数      占比%
-----------------------------------------------------------------------------------------------
1    TestFibonacci.fibonacci                                          5494     18.76
2    TestFibonacci.performMatrixMultiplication                        387      1.32
3    java/lang/String.split                                           19       0.06
4    TestFibonacci.performStringOperations                            6        0.02
5    java/util/AbstractCollection.toArray                             2        0.01
6    java/lang/String.toUpperCase                                     2        0.01
7    java/util/Arrayist$Subist$1.<init>                               1        0.0
```

Listing 5: TestFibonacci 程序的数据库分析结果

从数据库分析结果可以看出，TestFibonacci 程序中 `fibonacci` 函数占据了大部分的 CPU 时间，这与火焰图的结果一致。通过对调用栈信息的分析，可以进一步了解该函数的调用关系和性能瓶颈。

5.4 问题回答

经过对 Java 程序性能分析的实践，我对以下问题进行回答

5.4.1 Share a case study of using Perf tools to analyze or optimize the performance of a Java application.

在本次任务中，我使用 perf 工具对一个简单的 Java 程序（TestFibonacci）进行了性能分析。该程序使用递归方法计算斐波那契数列，并在运行过程中存在明显的性能瓶颈。通过 perf 工具的分析，我发现 `fibonacci` 函数的调用次数非常频繁，且每次调用都需要进行大量的计算，导致 CPU 使用率飙升。

为了解决这个问题，我尝试对 `fibonacci` 函数进行优化。我将递归实现改为迭代实现，以减少函数调用的开销。经过这些优化后，程序的性能得到了显著提升，CPU 使用率大幅降低。

5.4.2 Explore the benefits of combining different profiling techniques (e.g., perf stat for system-wide metrics, Flame Graphs for code-level visualization, and SQLite analysis for structured data review) to gain a comprehensive understanding of application performance.

结合不同的性能分析技术可以提供更全面的应用程序性能视图。perf stat 提供了系统级别的性能指标，如 CPU 使用率、上下文切换次数等，这些指标可以帮助我们了解程序在运行时的整体性能表现。Flame Graphs 则提供了代码级别的可视化，能够直观地展示函数调用栈信息，识别性能瓶颈和热点函数。而 SQLite 数据库分析则可以对采样数据进行结构化存储和查询，便于深入分析函数调用关系和性能特征，同时能持久化存储性能数据，便于对比分析。