Oracle® Developer Studio 12.5：性能分析器教程

文件号码 E71972
2016 年 6 月

ORACLE

文件号码 E71972

文档可访问性

有关 Oracle 对可访问性的承诺，请访问 Oracle Accessibility Program 网站 http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc。

获得 Oracle 支持

购买了支持服务的 Oracle 客户可通过 My Oracle Support 获得电子支持。有关信息，请访问 http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info；如果您听力受损，请访问 http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs。

目录

使用本文档

- **概述** – 提供了有关对样例程序使用 Oracle Developer Studio 12.5 性能分析器的逐步说明。
- **目标读者** – 应用程序开发人员、开发者、架构师、支持工程师
- **必备知识** – 编程经验、程序/软件开发测试、生成和编译软件产品的能力

产品文档库

有关该产品及相关产品的文档和资源，可从以下网址获得：http://docs.oracle.com/cd/E60778_01。

反馈

可以在 http://www.oracle.com/goto/docfeedback 上提供有关本文档的反馈。

性能分析器教程简介

性能分析器是用于检查 Java、C、C++ 和 Fortran 应用程序的性能的 Oracle Developer Studio 工具。可以使用它了解应用程序性能如何并找出问题区域。这些教程介绍了如何按照逐步说明对样例程序使用性能分析器。

关于性能分析器教程

本文档包含多个教程，说明如何使用性能分析器来分析各种类型的程序。每个教程都提供了对源文件使用性能分析器的步骤（教程中的大多数步骤都包括屏幕抓图）。

所有教程的源代码都包括在单个分发中。有关获取样例源代码的信息，请参见“获取教程的样例代码” [10]。

包括以下教程：

- C 分析简介

  此入门教程使用名为 `lowfruit` 的目标代码，是使用 C 语言编写的。`lowfruit` 程序非常简单，它包括两个编程任务的代码，这两个任务均以高效方式和低效方式实现。此教程说明如何在 C 目标程序上收集性能实验，以及如何在性能分析器中使用各种数据视图。检查每个任务的两种实现，并查看性能分析器如何显示哪个任务是高效的以及哪个任务是低效的。
- Java 分析简介

  此入门教程使用名为 `jlowfruit` 的目标代码，是使用 Java 语言编写的。与在 C 分析教程中使用的代码类似，`jlowfruit` 程序非常简单，它包括两个编程任务的代码，这两个任务均以高效方式和低效方式实现。此教程说明如何在 Java 目标上收集性能实验，以及如何在性能分析器中使用各种数据视图。检查每个任务的两种实现，并查看性能分析器如何显示哪个任务是高效的以及哪个任务是低效的。
- Java 和混合 Java-C++ 分析

  此教程基于名为 `jsynprog` 的 Java 代码，该代码可依次执行多个编程操作。一些操作执行运算，一个操作触发垃圾收集，而多个操作使用动态装入的 C++ 共享对象，从 Java 调用本机代码并再次返回。在此教程中，您可了解各种操作是如何实现的，以及性能分析器如何显示有关程序的性能数据。
- 多线程程序上的硬件计数器分析

此教程基于名为 `mttest` 的多线程程序，该程序运行多个任务、为每个任务衍生线程以及对每个任务使用不同的同步技术。在此教程中，您可了解任务中计算之间的性能差异，以及使用硬件计数器分析来检查和了解两个函数之间的意外性能差异。

- 多线程程序上的同步跟踪

  此教程也基于名为 `mttest` 的多线程程序，该程序运行多个任务、为每个任务衍生线程以及对每个任务使用不同的同步技术。在此教程中，检查同步技术之间的性能差异。

获取教程的样例代码

性能分析器教程中使用的程序包括在分发中，该分发包括用于所有 Oracle Developer Studio 工具的代码。如果之前尚未下载样例代码，请按照以下说明获取它。

1. 访问 Oracle Developer Studio 网页 `http://www.oracle.com/technetwork/server-storage/solarisstudio` 上的 Oracle Developer Studio 12.5 样例应用程序页面。
2. 导航到 Oracle Developer Studio 网页的下载部分。
3. 阅读页面上链接中的许可证，并通过选择 "Accept"（接受）接受它。
4. 下载 zip 文件，方法是单击其链接，并按照下载页面上的说明进行解压缩。

在下载并解压缩样例文件之后，您可以在 `OracleDeveloperStudio12.5-Samples/PerformanceAnalyzer` 目录中找到样例。

请注意，目录包括本文档中未介绍的一些其他样例：`cachetest`、`ksynprog`、`omptest` 和 `synprog`。每个样例子目录都包括一个 `Makefile` 和一个 `README` 文件，其中包括可用于进一步演示性能分析器的说明。

为教程设置环境

在试用教程之前，请确保在路经中具有 Oracle Developer Studio `bin` 目录，且在路经中具有适当的 Java 版本，如《Oracle Developer Studio 12.5：安装指南》 中的 第 5 章, “安装 Oracle Developer Studio 12.5 后”中所述。

`make` 或 `gmake` 命令也必须在您的路经上，以便您可以生成程序。

C 分析简介

本章包含以下主题。

关于 C 分析教程

此教程介绍了用 Oracle Developer Studio 性能分析器进行分析的最简单示例，并演示如何使用性能分析器收集和检查性能实验。在此教程中使用 "Overview"（概述）、"Functions"（函数）视图、"Source"（源）视图和 "Timeline"（时间线）。

程序 `lowfruit` 是执行两个不同任务的简单程序，一个任务用于在循环中进行初始化，另一个任务用于将数值插入到有序列表中。每个任务都执行两次，以低效方式和高效方式执行。

提示 - Java 分析简介教程使用等效的 Java 程序，并显示与性能分析器类似的活动。

在记录的实验中看到的数据将与此处显示的数据不同。用于教程中屏幕抓图的实验是在运行 Oracle Solaris 11.3 的 SPARC T5 系统上记录的。来自运行 Oracle Solaris 或 Linux 的 x86 系统的数据将会有所不同。此外，数据收集本质上是统计性的，随实验的不同而不同，即使运行在同一系统和 OS 上也是如此。

您看到的性能分析器窗口配置可能不会与屏幕抓图完全匹配。通过性能分析器，可以拖动窗口各部分之间的分隔条，折叠各部分以及调整窗口大小。性能分析器记录其配置，并在下次运行时使用相同的配置。在捕获教程所示的屏幕抓图的过程中进行了许多配置更改。

此教程在安装了 Oracle Developer Studio 的系统的本地运行。也可以远程运行，如“使用远程性能分析器” [101] 中所述。

设置 lowfruit 样例代码

开始之前：
查看以下内容以了解有关获取代码和设置您的环境的信息。

- “获取教程的样例代码” [10]
- “为教程设置环境” [10]

1. 使用以下命令将 lowfruit 目录的内容复制到您自己的专用工作区：

   ```
   % cp -r OracleDeveloperStudio12.5-Samples/PerformanceAnalyzer/lowfruit directory
   ```

   其中 *directory* 是您所使用的工作目录。
2. 转到该工作目录。

   ```
   % cd directory/lowfruit
   ```
3. 生成目标可执行文件。

   ```
   % make clobber

   % make
   ```

   注 - 仅当之前在目录中运行了 make 时才需要使用 clobber 子命令，但该子命令在任何情况下都可以放心地使用。

   在运行 make 之后，目录将包含要在教程中使用的目标程序，即一个名为 lowfruit 的可执行文件。

   下一节说明如何使用性能分析器从 lowfruit 程序收集数据以及创建实验。

提示 - 如果您愿意，可以编辑 Makefile 以执行以下操作：使用 GNU 编译器而不是缺省的 Oracle Developer Studio 编译器；以 32 位而不是缺省的 64 位编译；以及添加不同的编译器标志。

使用性能分析器收集数据

本节介绍如何使用性能分析器的 "Profile Application"（分析应用程序）功能收集实验中的数据。

提示 - 如果您不愿意执行这些步骤来了解如何分析应用程序，则可以使用 `lowfruit` 的 `Makefile` 中包括的 `make` 目标来记录实验：

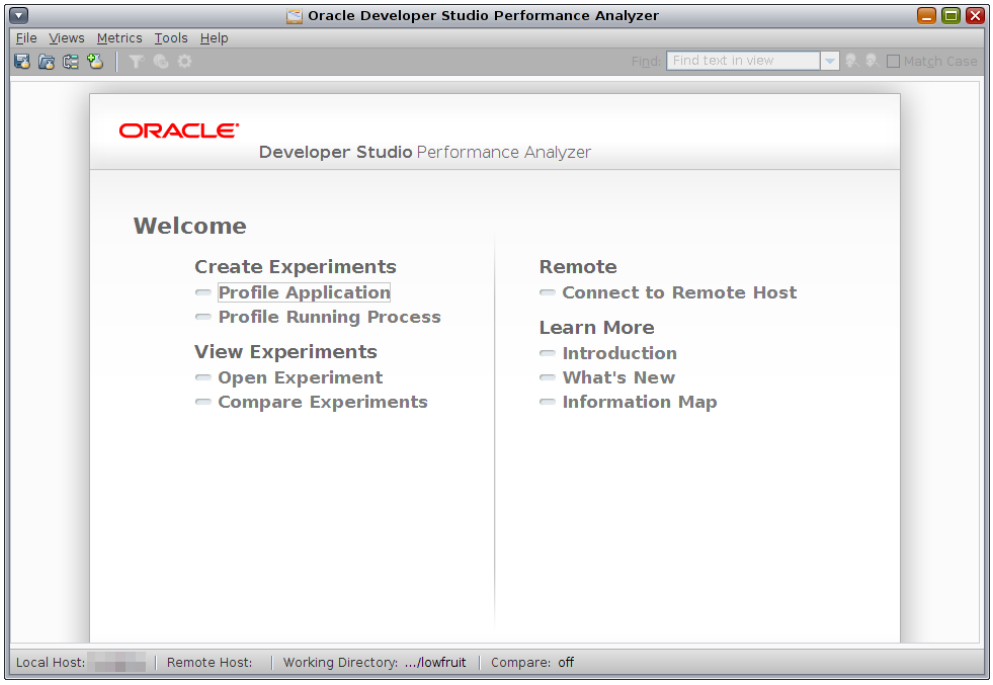
```
make collect
```

`collect` 目标将启动 `collect` 命令并记录实验，就像本节中使用性能分析器创建的那样。然后可以跳到“使用性能分析器检查 `lowfruit` 数据” [16]。

1. 仍在 `lowfruit` 目录中时，启动性能分析器：

   ```
   % analyzer
   ```

   性能分析器将启动并显示 "Welcome"（欢迎）页面。



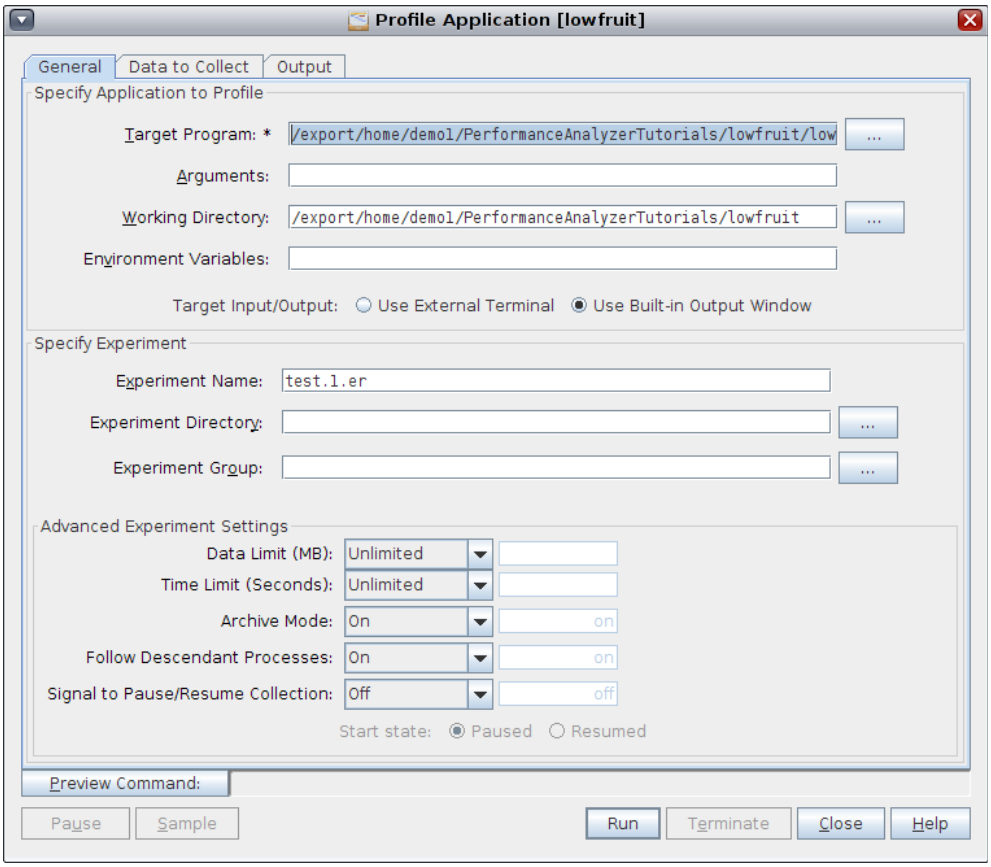
   如果这是首次使用性能分析器，则在 "Open Experiment"（打开实验）项下不显示最近的实验。如果之前已使用过它，则在当前运行性能分析器的系统中可以看到最近打开的实验的列表。

2. 单击 "Welcome"（欢迎）页面中 "Create Experiments"（创建实验）下的 "Profile Application"（分析应用程序）链接。

   "Profile Application"（分析应用程序）对话框将打开，其中 "General"（常规）标签处于选中状态。此页上的选项组织成以下几个区域："Specify Application to Profile"

（指定要分析的应用程序）、"Specify Experiment"（指定实验）以及 "Advanced Experiment"（高级实验设置）。

3. 在 "Target Program"（目标程序）字段中，键入程序名称 `lowfruit`。



提示 - 可以启动性能分析器，并通过已输入的程序名称（使用命令 `analyzer lowfruit` 启动性能分析器时指定目标名称）直接打开此对话框。此方法仅在本地运行性能分析器时才起作用。

---

4. 对于 "Specify Application to Profile"（指定要分析的应用程序）面板底部的 "Target Input/Output"（目标输入/输出）选项，选择 "Use Built-in Output Window"（使用内置输出窗口）。

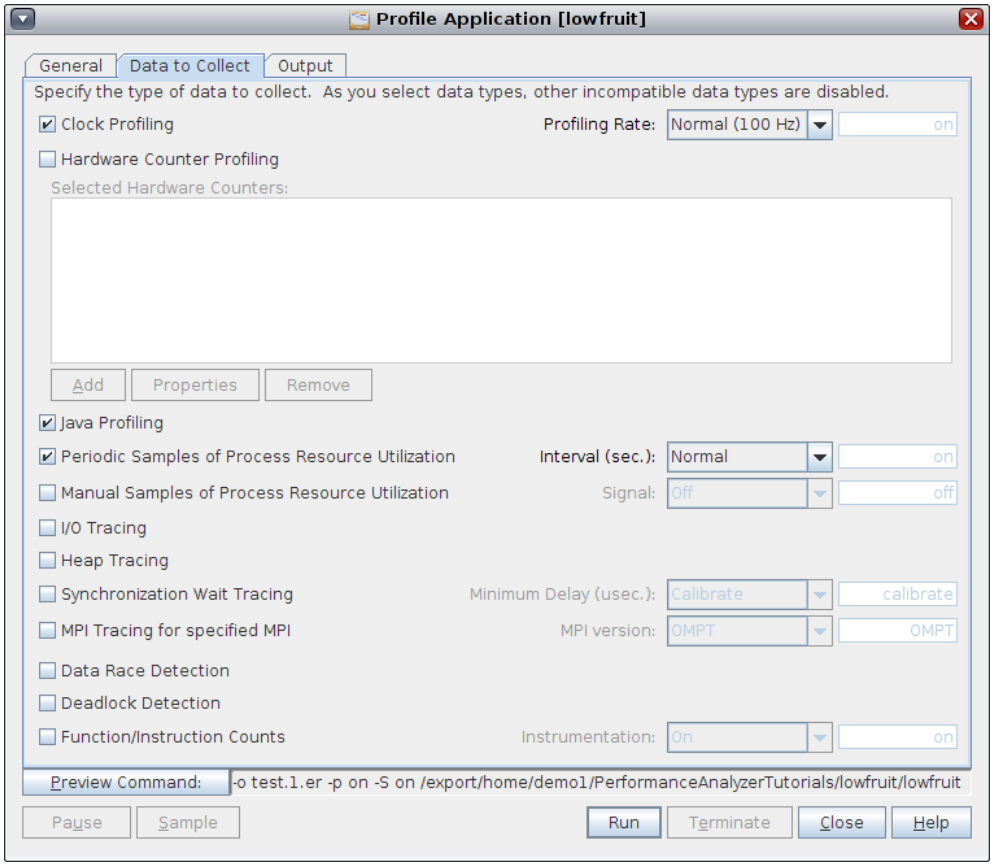
   "Target Input/Output"（目标输入/输出）选项指定目标程序 `stdout` 和 `stderr` 将重定向到的窗口。缺省值是 "Use External Terminal"（使用外部终端），但是在此教程中，"Target Input/Output"（目标输入/输出）选项已更改为 "Use Built-in Output

Window"（使用内置输出窗口），以便在性能分析器窗口中保持所有的活动。使用此选项时，"Profile Application"（分析应用程序）对话框的 "Output"（输出）标签上将显示 `stdout` 和 `stderr`。

如果远程运行，则 "Target Input/Output"（目标输入/输出）选项不存在，因为仅支持内置输出窗口。

5. 对于 "Experiment Name"（实验名称）选项，缺省的实验名称是 `test.1.er`，但是您可以将它更改为其他名称，只要该名称以 `.er` 结尾且尚未使用。
6. 单击 "Data to Collect"（要收集的数据）标签。

   通过 "Data to Collect"（要收集的数据），可以选择要收集的数据类型，以及显示已选择的缺省值。



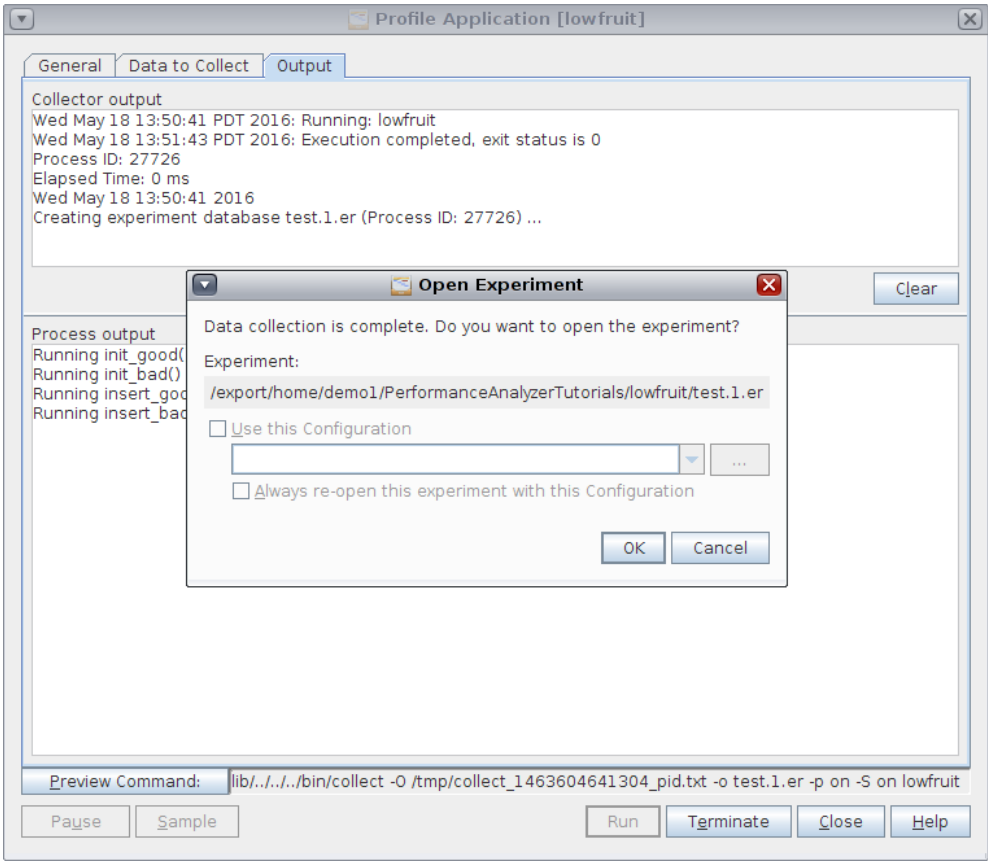
正如在屏幕抓图中所见，缺省情况下启用 Java 分析，但是对于非 Java 目标（如 `lowfruit`）则忽略它。

也可以单击 "Preview Command"（预览命令）按钮，并查看开始分析时将运行的 `collect` 命令。

7. 单击工具栏上的 "Run"（运行）按钮。

   "Profile Application"（分析应用程序）对话框显示 "Output"（输出）标签，并显示它在 "Process Output"（进程输出）面板中运行时的程序输出。

   程序完成后，将显示一个对话框，询问您是否要打开刚记录的实验。



8. 在对话框中单击 OK。

   实验将打开。下一节说明如何检查数据。

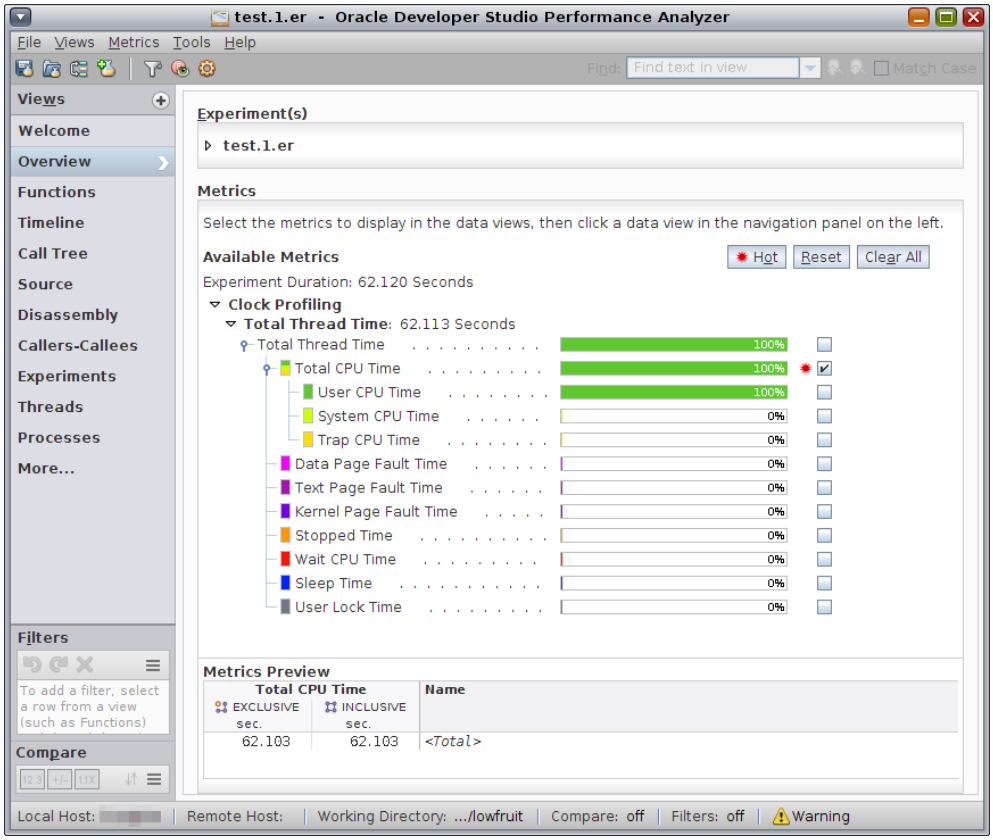
使用性能分析器检查 `lowfruit` 数据

本节介绍如何了解通过 `lowfruit` 样例代码创建的实验中的数据。

1. 如果在上一节中创建的实验尚未打开，则可以从 lowfruit 目录启动性能分析器并装入实验，如下所示：

```
% analyzer test.1.er
```

实验打开时，性能分析器将显示 "Overview"（概述）屏幕。



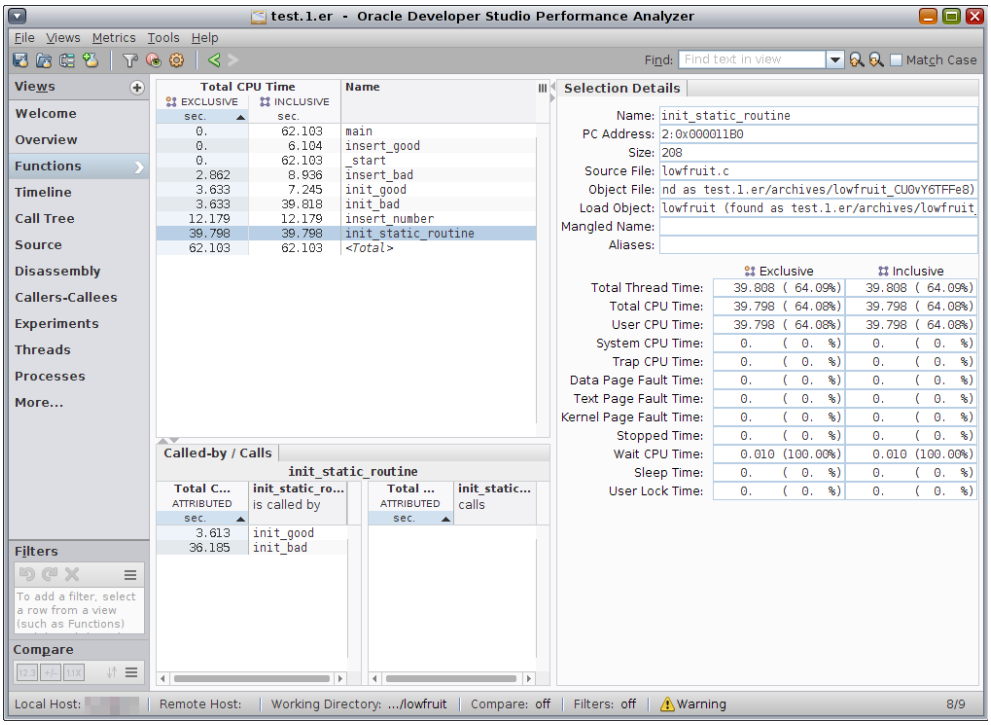
在此实验中，"Overview"（概述）本质上显示 100% 用户 CPU 时间。程序是单线程的，且一个线程是 CPU 绑定的。实验记录在 Oracle Solaris 系统上，"Overview"（概述）显示记录的十二个度量，但是缺省情况下仅启用 "Total CPU Time"（CPU 总时间）。

具有带颜色指示符的度量是由 Oracle Solaris 定义的十个微状态所用的时间。这些度量包括 "User CPU Time"（用户 CPU 时间）、"System CPU Time"（系统 CPU 时间）和 "Trap CPU Time"（自陷 CPU 时间）（合在一起等于 "Total CPU Time"（CPU 总时间）），以及各种等待时间。"Total Thread Time"（总线程时间）是所有微状态的总和。

在 Linux 计算机上，仅记录 "Total CPU Time"（CPU 总时间），因为 Linux 不支持微状态记帐。

缺省情况下会预览 "Inclusive Total CPU Time"（包含总 CPU 时间）和 "Exclusive Total CPU Time"（独占总 CPU 时间）。任何度量的包含都是指该函数或方法中的度量值，其中包括在所有函数或其所调用方法中累计的度量。独占仅指在该函数或方法内累计的度量。

2. 单击左侧 "Views"（视图）导航栏中的 "Functions"（函数）视图，或者从菜单栏使用 "Views"（视图）-> "Functions"（函数）选择它。



"Functions"（函数）视图显示应用程序中函数的列表，以及每个函数的性能度量。列表最初按在每个函数中所用的 "Exclusive Total CPU Time"（独占总 CPU 时间）排序。列表包括目标应用程序中的所有函数以及程序使用的任何共享对象。缺省情况下，最顶端的函数（即耗费资源最多的函数）处于选定状态。

右侧的 "Selection Details"（选择详细信息）窗口显示选定函数的所有已记录度量。

函数列表下的 "Called-by/Calls"（调用方/调用）面板提供有关选定函数的更多信息，并分为两个列表。"Called-by"（调用方）列表显示选定函数的调用方，度量值显示归属于其调用方的函数总度量。"Calls"（调用）列表显示选定函数的被调用方，并显示其被调用方的包含度量如何有助于选定函数的总度量。如果在 "Called-

by/Calls"（调用方/调用）面板中双击任一列表中的函数，则在主 "Functions"（函数）视图中该函数将变为选定的函数。

3. 通过选择各种函数进行实验，以了解 "Functions"（函数）视图中的窗口如何随选择的变化而更新。

   "Selection Details"（选择详细信息）窗口显示大多数的函数来自 `lowfruit` 可执行文件，如 "Load Object"（装入对象）字段中所示。

   也可以通过单击列标题进行实验，以将排序从 "Exclusive Total CPU Time"（独占总 CPU 时间）更改为 "Inclusive Total CPU Time"（包含总 CPU 时间），或者按 "Name"（名称）排序。

4. 在 "Functions"（函数）视图中，对初始化任务的两个版本 `init_bad（）` 和 `init_good（）` 进行比较。

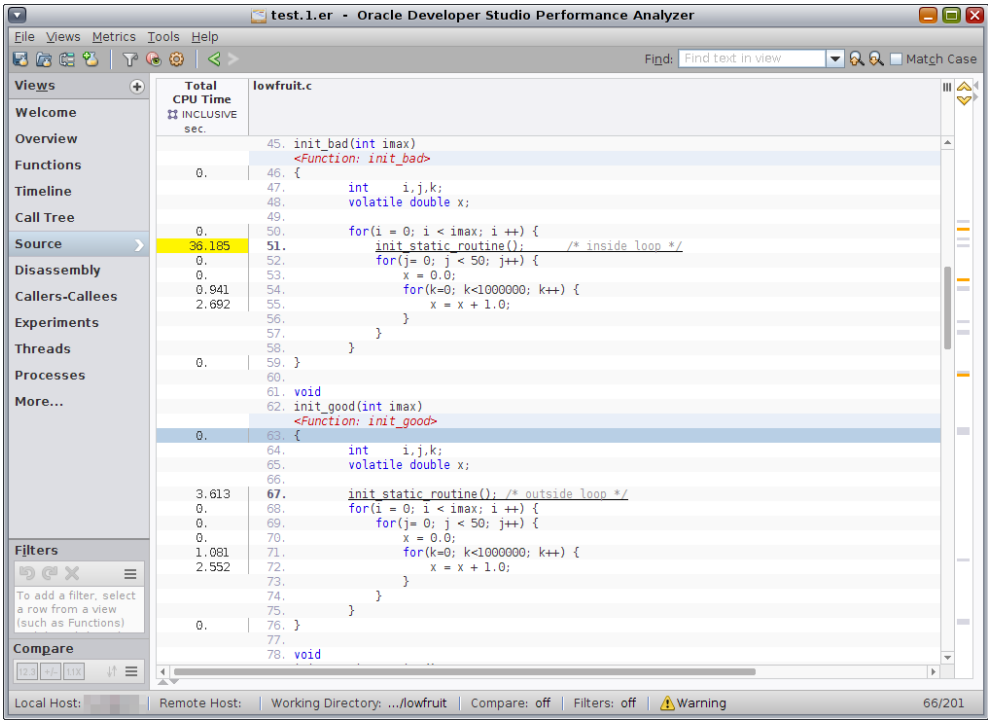
   可以看到，这两个函数具有大致相同的 "Exclusive Total CPU Time"（独占总 CPU 时间），但是包含时间截然不同。`init_bad（）` 函数的运行速度较慢，因为它在被调用方中花费了时间。这两个函数调用同一被调用方，但是它们在该例程中所用的时间截然不同。通过检查这两个例程的源代码，您可以找出原因。

5. 选择函数 `init_good（）`，然后单击 "Source"（源）视图，或者从菜单栏中选择 "Views"（视图）-> "Source"（源）。

6. 调整窗口以便为代码提供更多空间：单击上边缘的向下箭头折叠 "Called-by/Calls"（调用方/调用）面板，单击侧边缘的向右箭头折叠 "Selection Details"（选择详细信息）面板。

注 - 在教程的其余部分，您可能需要重新展开和重新折叠这些面板。

您应该向上滚动一点以便同时看到 `init_bad（）` 和 `init_good（）` 的源代码。"Source"（源）视图看起来应该与以下屏幕抓图类似。



请注意，对 `init_static_routine`（） 的调用在 `init_good`（） 中的循环之外，而 `init_bad`（） 对 `init_static_routine`（） 的调用则在循环之内。与好版本相比，差版本所用时间大约长十倍（与循环计数相对应）。

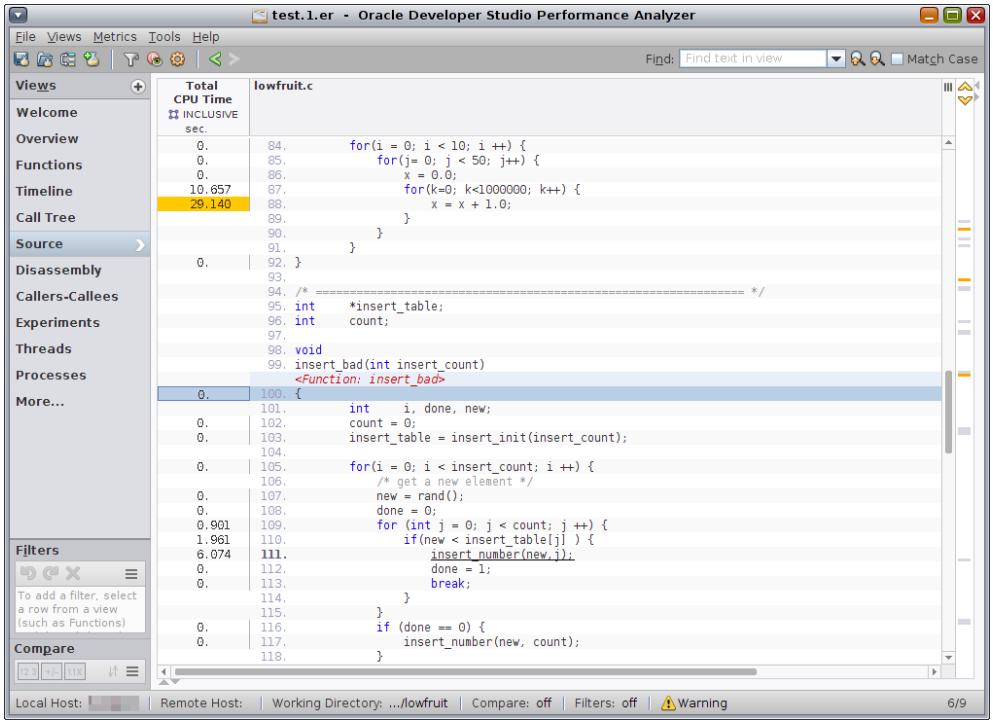
此示例不像看起来的那样笨拙。它基于真实的代码，可生成一个表，其中每行都有一个图标。虽然很容易看出在此示例中初始化不应在循环之内，但是在真实的代码中，初始化嵌入在库例程中且不易看出。

用于实现该代码的工具包提供两个库调用 (API)。第一个 API 将图标添加到表行，第二个 API 将图标向量添加到整个表。虽然使用第一个 API 更易于编码，但是每次添加图标时，工具包都要重新计算所有行的高度，以便为整个表设置正确的值。代码使用另一个 API 一次性添加所有图标时，高度的重新计算仅执行一次。

7. 现在，返回到 "Functions"（函数）视图，并查看插入任务的两个版本 `insert_bad`（） 和 `insert_good`（）。

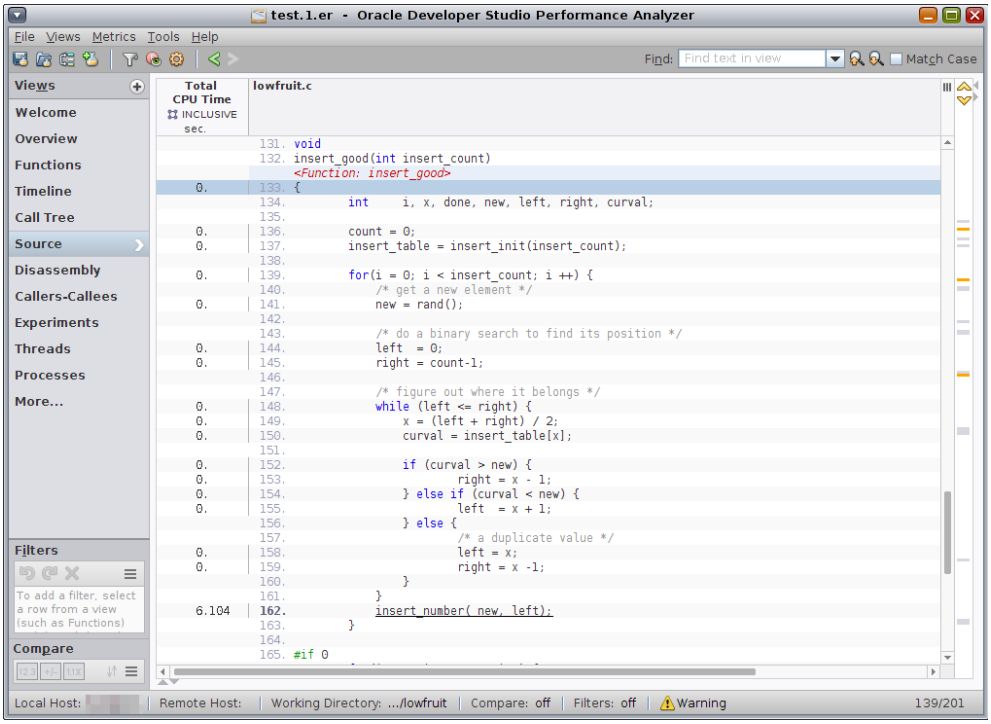
   请注意，"Exclusive Total CPU time"（独占总 CPU 时间）对 `insert_bad`（） 很重要，但是对 `insert_good`（） 则可以忽略。每个版本的包含时间和独占时间（表示被调用以将每个条目插入到列表中的函数 `insert_number`（） 中的时间）之间的差异是相同的。通过检查源代码，可以找出原因。

8. 选择 `insert_bad`（） 并切换到 "Source"（源）视图：



请注意时间（不包括对 `insert_number（）` 的调用）花费在循环查找中，通过线性搜索找到正确的位置以插入新数字。

9. 现在向下滚动以查看 `insert_good（）`。



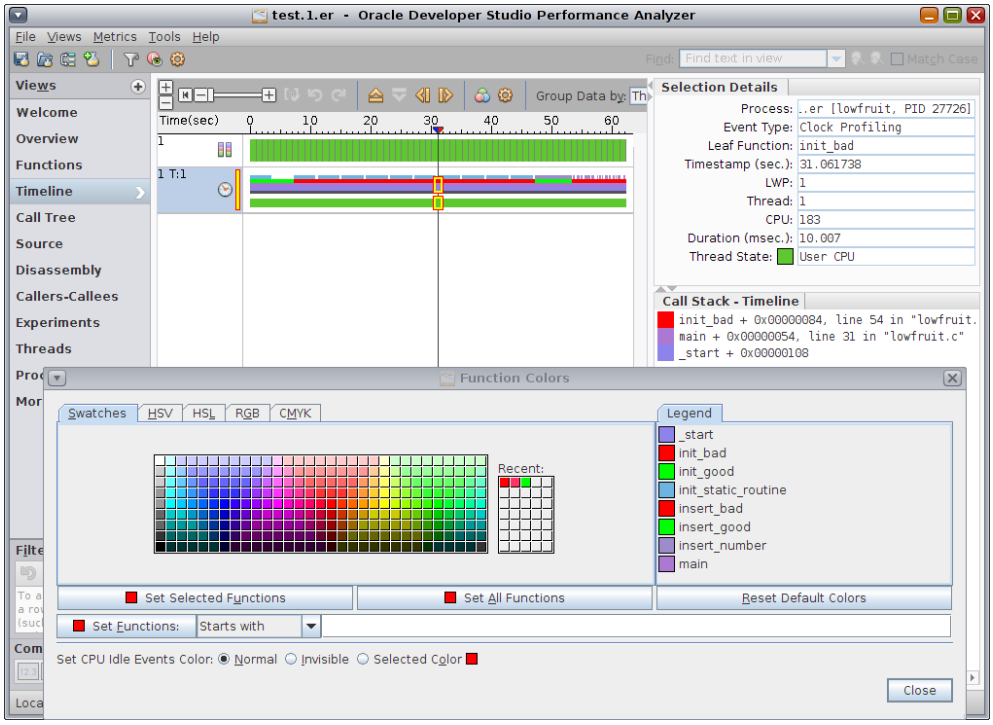
请注意代码更为复杂，因为它执行二进制搜索，以查找要插入的正确位置，但是所用的总时间（不包括对 `insert_number`（） 的调用）比 `insert_bad`（） 中少得多。此示例说明二进制搜索的效率比线性搜索更高。

您也可以在 "Timeline"（时间线）视图中以图形方式查看例程中的差异。

10. 单击 "Timeline"（时间线）视图，或者从菜单栏中选择 "Views"（视图）-> "Timeline"（时间线）。

    会将分析数据记录为一系列事件，每个事件表示每个线程的分析时钟的每一计时周期。"Timeline"（时间线）视图显示每个单独的事件，以及记录在该事件中的调用堆栈。调用堆栈以调用堆栈中的帧列表形式显示，其中叶 PC（在事件发生的瞬间随即执行的指令）处于顶部，其次为调用它的调用点，等等。对于程序的主线程，调用堆栈的顶部始终为 `_start`。

11. 在 "Timeline"（时间线）工具栏中，单击 "Call Stack Function Colors"（调用堆栈函数颜色）图标  以对函数着色，或者从菜单栏中选择 "Tools"（工具）-> "Function Colors"（函数颜色），然后将看到如下所示的对话框。



函数颜色已更改，以便为屏幕抓图更清晰地区分函数的好版本和差版本。`init_bad`() 和 `insert_bad`() 函数现在都呈红色，而 `init_good`() 和 `insert_good`() 现在都呈亮绿色。

12. 要使 "Timeline"（时间线）视图的外观类似，请在 "Function Colors"（函数颜色）对话框中执行以下操作：
    - 向下滚动 "Legend"（图例）中的方法列表以查找 `init_bad`() 方法。
    - 选择 `init_bad`() 方法，单击 "Swatches"（色板）中的红色方块，然后单击 "Set Selected Functions"（设置所选函数）按钮。
    - 选择 `insert_bad`() 方法，单击 "Swatches"（色板）中的红色方块，然后单击 "Set Selected Functions"（设置所选函数）按钮。
    - 选择 `init_good`() 方法，单击 "Swatches"（色板）中的绿色方块，然后单击 "Set Selected Functions"（设置所选函数）按钮。
    - 选择 `insert_good`() 方法，单击 "Swatches"（色板）中的绿色方块，然后单击 "Set Selected Functions"（设置所选函数）按钮。
13. 查看时间线的顶部条。

    时间线的顶部条是 "CPU Utilization Samples"（CPU 利用率抽样）条，将鼠标光标移动到第一列上时，可以在工具提示中看到它。"CPU Utilization Samples"（CPU

利用率抽样）条的每个段表示一秒的间隔，显示在该秒执行期间目标的资源使用率。

在此示例中，所有段都呈绿色，因为所有的间隔都用在累计用户 CPU 时间上。"Selection Details"（选择详细信息）窗口显示颜色到微状态的映射，尽管它在屏幕抓图中不可见。

14. 查看时间线的第二个条。

    第二个条是 "Clock Profiling Call Stacks"（时钟分析调用堆栈）条，标记有 "1 T:1"，这表示进程 1 和线程 1（示例中的唯一线程）。"Clock Profiling Call Stacks"（时钟分析调用堆栈）条显示程序执行期间发生的事件的两个数据条。靠上的条显示调用堆栈的颜色编码表示形式，靠下的条显示每个事件的线程状态。此示例中的状态始终为 "User CPU Time"（用户 CPU 时间），因此它看起来是一条纯绿色的线。

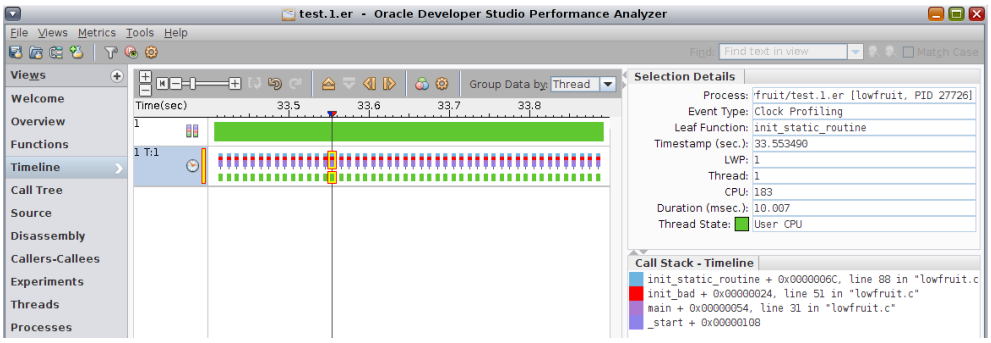
    如果单击该 "Clock Profiling Call Stacks"（时钟分析调用堆栈）条中的任何位置，则选择最近的事件，并在 "Selection Details"（选择详细信息）窗口中显示该事件的详细信息。从调用堆栈的图案中，可以看到在屏幕抓图中以亮绿色显示的 `init_good（）` 和 `insert_good（）` 例程中的时间比以红色显示的 `init_bad（）` 和 `insert_bad（）` 例程中的对应时间要短得多。

15. 选择与时间线中的好例程和差例程相对应的区域中的事件，并在 "Selection Details"（选择详细信息）窗口下的 "Call Stack - Timeline"（调用堆栈 - 时间线）窗口中查看调用堆栈。

    可以在 "Call Stack"（调用堆栈）窗口中选择任何帧，然后在 "Views"（视图）导航栏上选择 "Source"（源）视图，并转到该源代码行的源代码。也可以双击调用堆栈中的帧以转到 "Source"（源）视图，或者右键单击调用堆栈中的帧并从弹出菜单中进行选择。

16. 通过时间线顶部的滑块、使用 + 键或者通过用鼠标双击，均可以放大事件。

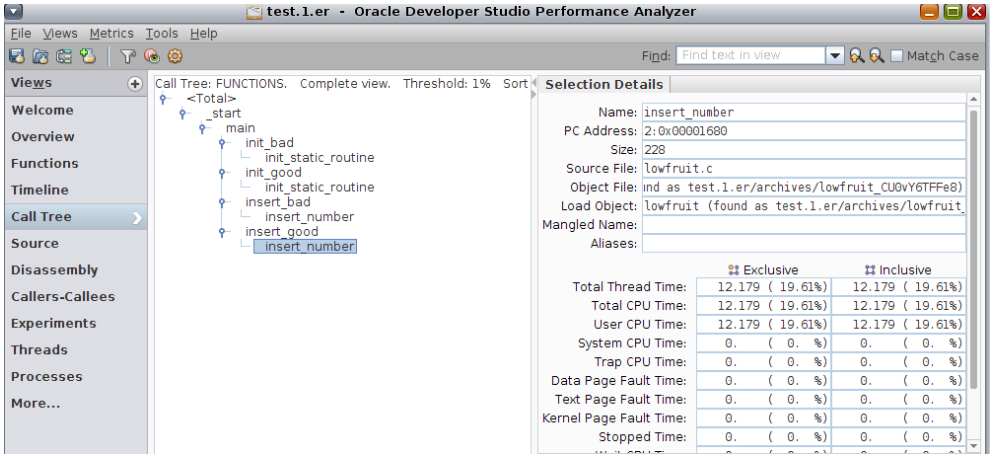
    如果放大得足够大，则可以看到显示的数据不是连续的，而是由离散的事件组成，每个分析计时周期（在此示例中约为 10 ms）都有一个事件。



按 F1 键可查看帮助以了解有关 "Timeline"（时间线）视图的更多信息。

17. 单击 "Call Tree"（调用树）视图或者选择 "Views"（视图）-> "Call Tree"（调用树），以查看程序的结构。

"Call Tree"（调用树）视图显示程序的动态调用图，而 "Selection Details"（选择详细信息）面板会显示性能信息。



性能分析器具有数据的许多其他视图，如 "Caller-Callees"（调用方-被调用方）视图（通过该视图可以在程序结构中导航）和 "Experiments"（实验）视图（显示已记录实验的详细信息）。对于此简单示例，"Threads"（线程）和 "Processes"（进程）视图没有什么要关注的内容。

通过单击 "Views"（视图）列表上的 + 按钮，可以将其他视图添加到导航栏。如果您是汇编语言程序员，则可能希望查看 "Disassembly"（反汇编）。尝试了解其他视图。

性能分析器还具有非常强大的过滤功能。可以按时间、线程、函数、源代码行、指令、调用堆栈片段及其任意组合进行过滤。使用过滤功能超出了此教程的范围，因为样例代码如此简单，不需要进行过滤。

Java 分析简介

本章包含以下主题。

关于 Java 分析教程

此教程介绍了用 Oracle Developer Studio 性能分析器进行分析的最简单示例，并演示如何使用性能分析器收集和检查性能实验。在本教程中将使用 "Overview"（概述）、"Functions"（函数）视图、"Timeline"（时间线）视图和 "Call Tree"（调用树）视图。

程序 `jlowfruit` 是执行两个不同任务的简单程序，一个任务用于在循环中进行初始化，另一个任务用于将数值插入到有序列表中。每个任务都执行两次，以低效方式和高效方式执行。

提示 - C 分析简介教程使用等效 C 程序并显示与性能分析器相似的活动。

在记录的实验中看到的数据将与此处显示的数据不同。用于教程中屏幕抓图的实验是在运行 Oracle Solaris 11.3 的 SPARC T5 系统上记录的。来自运行 Oracle Solaris 或 Linux 的 x86 系统的数据将会有所不同。此外，数据收集本质上是统计性的，随实验的不同而不同，即使运行在同一系统和 OS 上也是如此。

您看到的性能分析器窗口配置可能不会与屏幕抓图完全匹配。通过性能分析器，可以拖动窗口各部分之间的分隔条，折叠各部分以及调整窗口大小。性能分析器记录其配置，并在下次运行时使用相同的配置。在捕获教程所示的屏幕抓图的过程中进行了许多配置更改。

此教程在安装了 Oracle Developer Studio 的系统的本地运行。也可以远程运行，如“使用远程性能分析器” [101] 中所述。

设置 jlowfruit 样例代码

开始之前：
查看以下内容以了解有关获取代码和设置您的环境的信息。

- “获取教程的样例代码” [10]
- “为教程设置环境” [10]

1. 使用以下命令将 jlowfruit 目录的内容复制到您自己的专用工作区：

   ```
   % cp -r OracleDeveloperStudio12.5-Samples/PerformanceAnalyzer/jlowfruit directory
   ```

   其中 *mydirectory* 是您所使用的工作目录。
2. 转到该工作目录副本。

   ```
   % cd directory/jlowfruit
   ```
3. 生成目标可执行文件。

   ```
   % make clobber

   % make
   ```

   注 - 仅当之前在目录中运行了 make 时才需要使用 clobber 子命令，但该子命令在任何情况下都可以放心地使用。

   运行 make 后，目录将包含教程中要使用的目标应用程序，即名为 jlowfruit.class 的 Java 类文件。

提示 - 如果编译样例时遇到问题，请使用以下命令检查 javac 版本：

```
% javac -version
```

如果输出并未报告版本至少为 1.7 的 javac，则需要将 PATH 更新为 JDK 7 或更高版本。

下一节说明如何使用性能分析器从 jlowfruit 程序收集数据以及创建实验。

使用性能分析器从 jlowfruit 中收集数据

本节介绍如何使用性能分析器的 "Profile Application"（分析应用程序）功能收集 Java 应用程序中的实验中的数据。

提示 - 如果您不愿意执行这些步骤来了解如何从性能分析器来分析应用程序，则可以使用 jlowfruit 的 Makefile 中包括的 make 目标来记录实验：

```
% make collect
```

collect 目标将启动 collect 命令并记录实验，就像本节中使用性能分析器创建的那样。然后可以跳到“使用性能分析器检查 jlowfruit 数据” [32]。

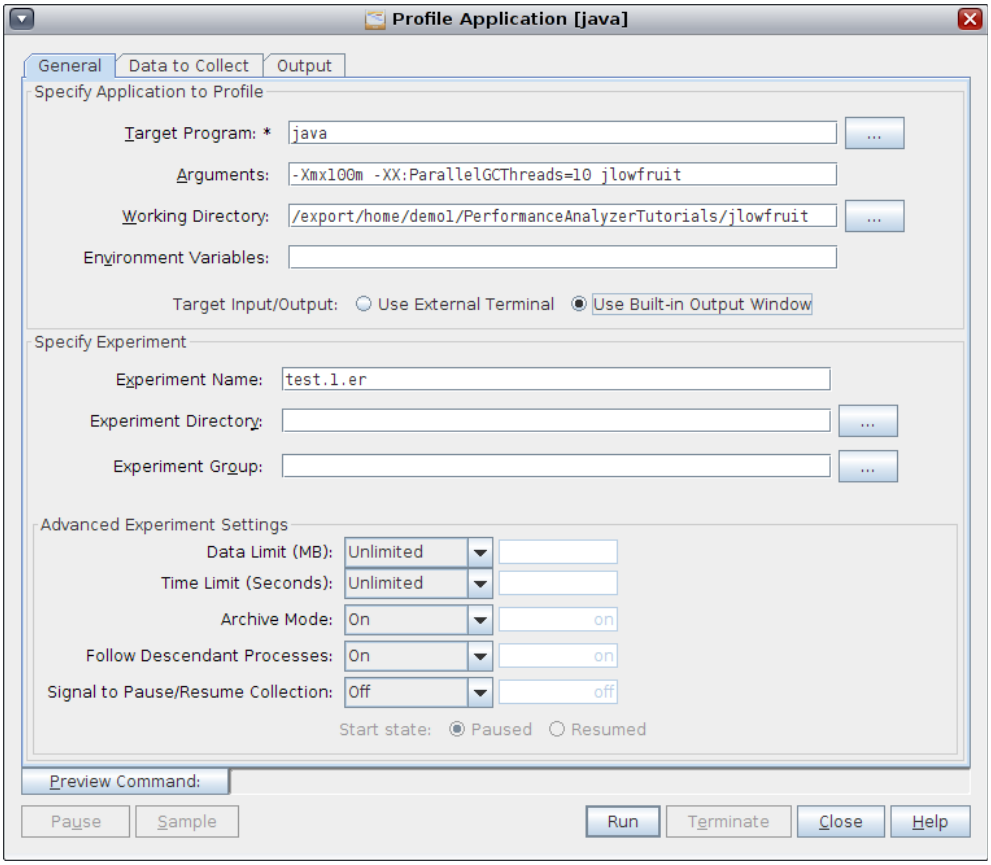
1. 仍在 jlowfruit 目录中时，使用目标 java 及其参数启动性能分析器：

   ```
   % analyzer java -Xmx100m -XX:ParallelGCThreads=10 jlowfruit
   ```

   "Profile Application"（分析应用程序）对话框将打开，其中 "General"（常规）选项卡处于选中状态，同时已经使用随 analyzer 命令提供的信息填充了多个选项。

   "Target Program"（目标程序）设置为 java，"Arguments"（参数）设置为

   ```
   -Xmx100m -XX:ParallelGCThreads=10 jlowfruit
   ```



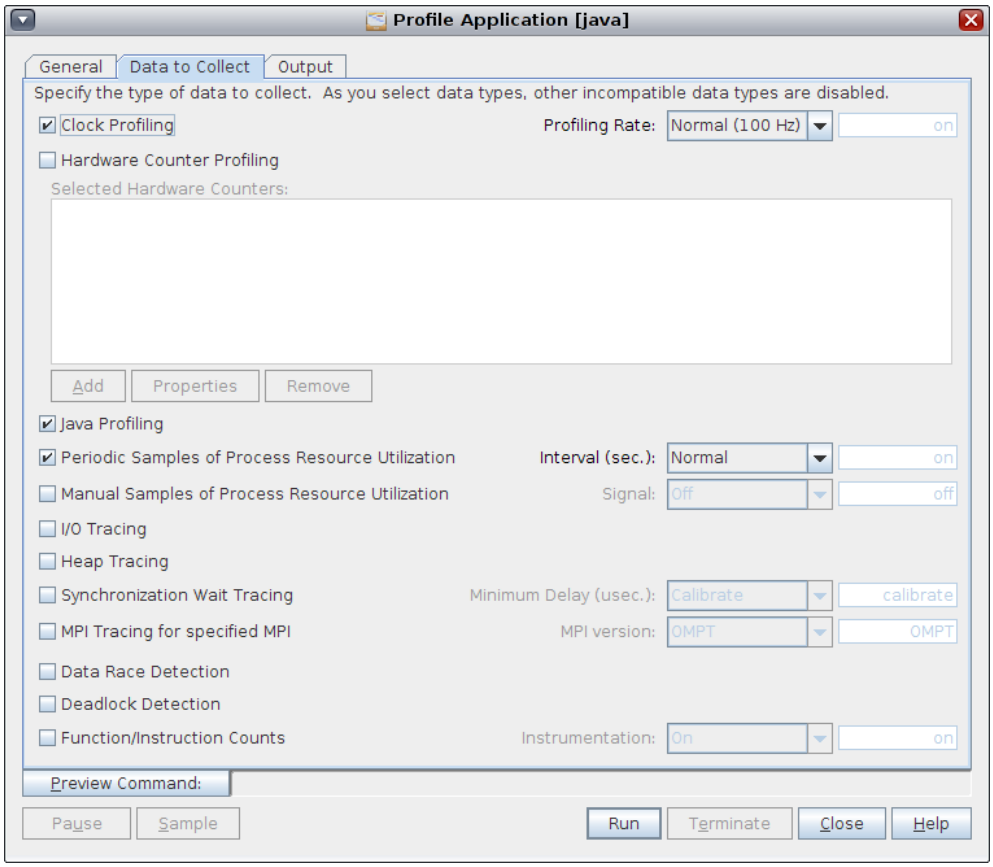
2. 对于 "Target Input/Output"（目标输入/输出）选项，选择 "Use Built-in Output Window"（使用内置输出窗口）。

   "Target Input/Output"（目标输入/输出）选项指定目标程序 `stdout` 和 `stderr` 将重定向到的窗口。缺省值为 "Use External Terminal"（使用外部终端），但是在本教程中您应该将 "Target Input/Output"（目标输入/输出）选项更改为 "Use Built-in Output Window"（使用内置输出窗口）以保留性能分析器窗口中的所有活动。使用此选项时，"Profile Application"（分析应用程序）对话框的 "Output"（输出）标签上将显示 `stdout` 和 `stderr`。

   如果远程运行，则 "Target Input/Output"（目标输入/输出）选项不存在，因为仅支持内置输出窗口。

3. 对于 "Experiment Name"（实验名称）选项，缺省的实验名称是 `test.1.er`，但是您可以将它更改为其他名称，只要该名称以 `.er` 结尾且尚未使用。
4. 单击 "Data to Collect"（要收集的数据）标签。

使用 "Data to Collect"（要收集的数据）标签，可以选择要收集的数据类型并显示已经选择的缺省值。



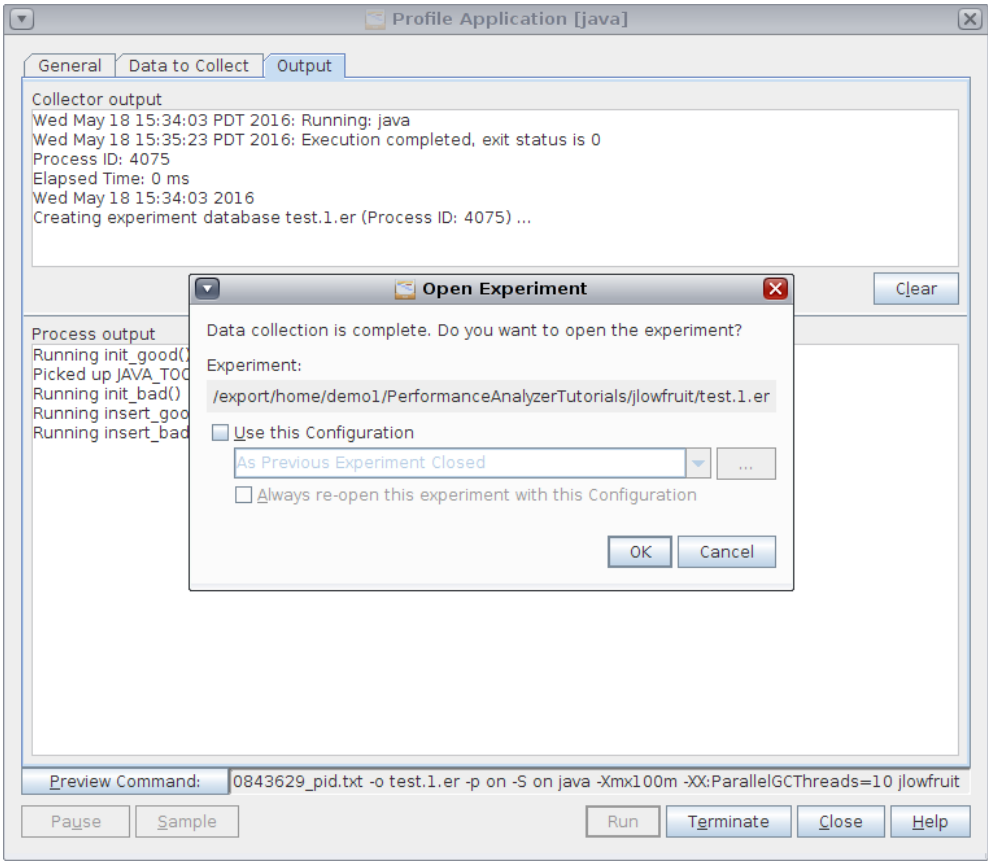
如屏幕快照中所示，缺省情况下启用 Java 分析。

也可以单击 "Preview Command"（预览命令）按钮，并查看开始分析时将运行的 `collect` 命令。

5. 单击工具栏上的 "Run"（运行）按钮。

   "Profile Application"（分析应用程序）对话框显示 "Output"（输出）标签，并在程序运行时在 "Process Output"（进程输出）面板中显示程序输出。

   程序完成后，将显示一个对话框，询问您是否要打开刚记录的实验。



6. 在对话框中单击 OK。

   实验将打开。下一节说明如何检查数据。

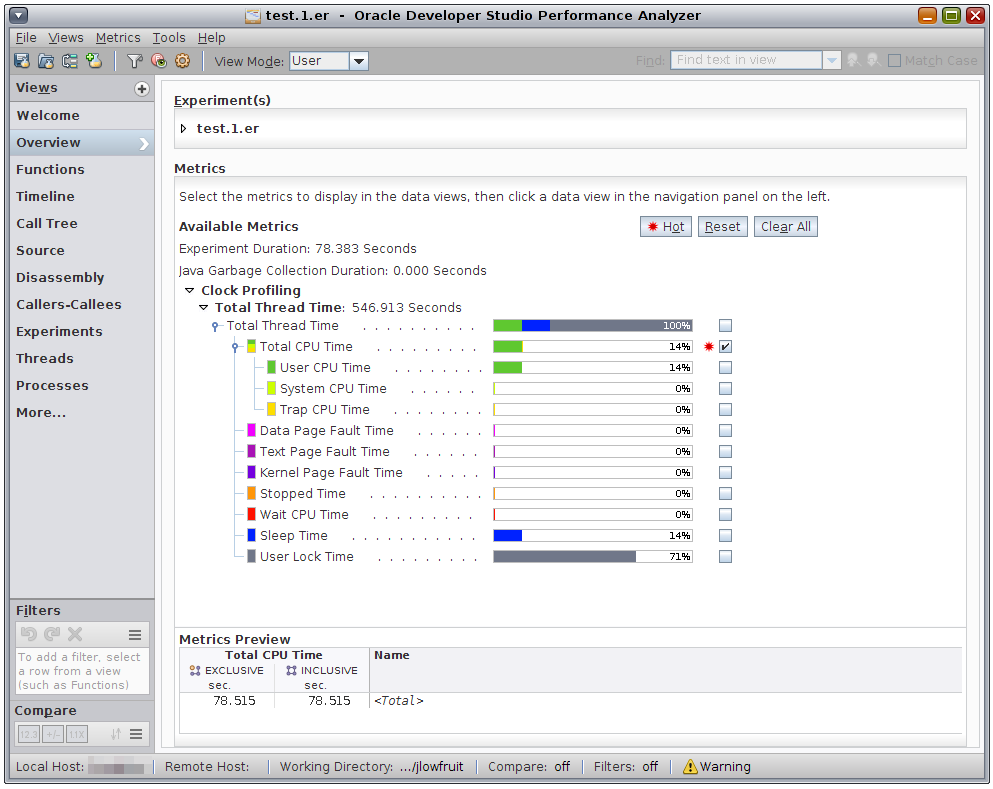
使用性能分析器检查 jlowfruit 数据

本节介绍如何了解通过 jlowfruit 样例代码创建的实验中的数据。

1. 如果在上一节中创建的实验尚未打开，则可以从 jlowfruit 目录启动性能分析器并装入实验，如下所示：

   ```
   % analyzer test.1.er
   ```

   实验打开时，性能分析器将显示 "Overview"（概述）页面。
2. 请注意，"Overview"（概述）页面显示度量值摘要并允许您选择度量。



在此实验中，"Overview"（概述）显示大约 14% 的 "Total CPU Time"（CPU 总时间）（其是所有 "User CPU Time"（用户 CPU 时间）），加上大约 14% 的 "Sleep Time"（休眠时间）和 71% 的 "User Lock Time"（用户锁定时间）。用户 Java 代码 `jlowfruit` 是单线程的并且该线程是计算密集型的，但是所有 Java 程序使用多个线程，包括许多系统线程。这些线程的数量取决于选择的 JVM 选项，包括 "Garbage Collector"（垃圾收集器）参数以及运行程序的计算机的大小。

实验记录在 Oracle Solaris 系统上，"Overview"（概述）显示记录的十二个度量，但是缺省情况下仅启用 "Total CPU Time"（CPU 总时间）。

具有带颜色指示符的度量是由 Oracle Solaris 定义的十个微状态所用的时间。这些度量包括 "User CPU Time"（用户 CPU 时间）、"System CPU Time"（系统 CPU 时间）和 "Trap CPU Time"（自陷 CPU 时间）（合在一起等于 "Total CPU Time"（CPU 总时间）），以及各种等待时间。"Total Thread Time"（总线程时间）是所有微状态的总和。

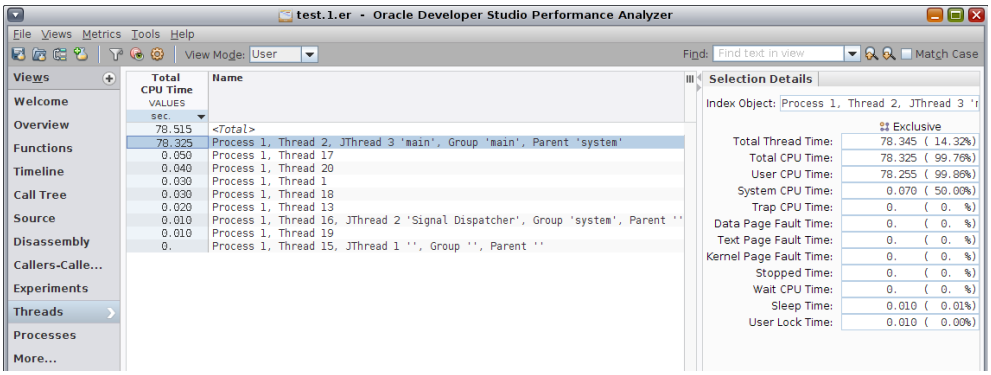
在 Linux 计算机上，仅记录 "Total CPU Time"（CPU 总时间），因为 Linux 不支持微状态记帐。

缺省情况下会预览 "Inclusive Total CPU Time"（包含总 CPU 时间）和 "Exclusive Total CPU Time"（独占总 CPU 时间）。任何度量的包含都是指该函数或方法中的度量值，其中包括在所有函数或其所调用方法中累计的度量。独占仅指在该函数或方法内累计的度量。

3. 单击 "Hot"（常用）按钮可以选择具有较高值的度量，从而在数据视图中显示这些度量。

   将更新底部的 "Metrics Preview"（度量预览）面板来显示在提供表格式数据的数据视图中显示度量的方式。接下来，您将查看哪些线程负责哪些度量。

4. 现在，通过以下方法切换到 "Threads"（线程）视图：在 "Views"（视图）导航面板中单击线程名称，或者从菜单栏中选择 "Views"（视图）-> "Threads"（线程）。



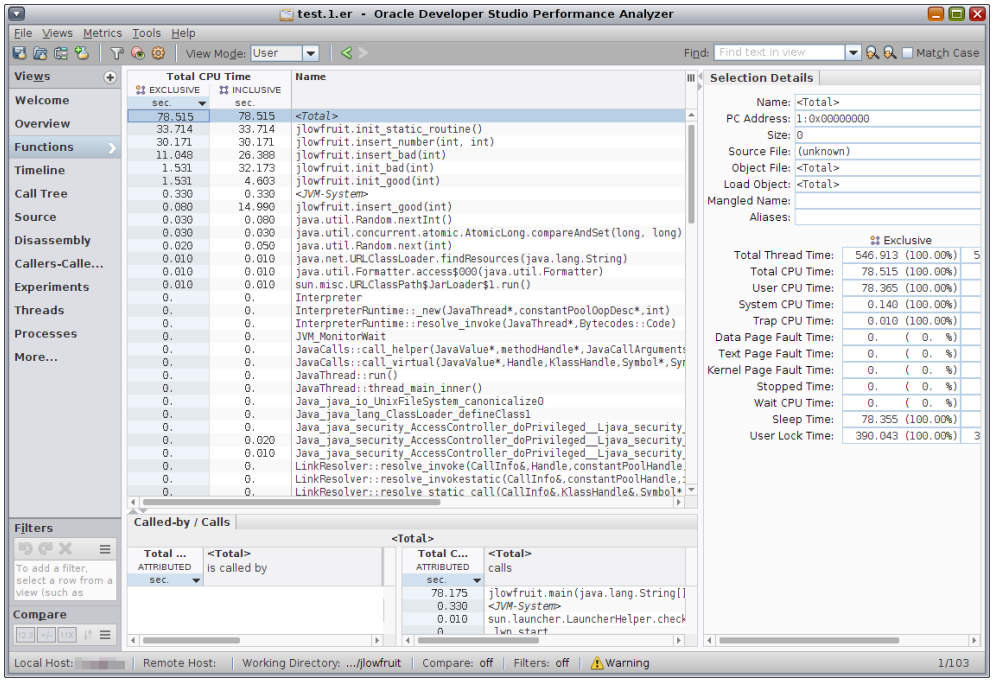
具有几乎所有 "Total CPU Time"（CPU 总时间）的线程是 "Thread 2"（线程 2），其是此示例应用程序中的唯一用户 Java 线程。

"Thread 15"（线程 15）最可能是用户线程，即使它实际上是由 JVM 在内部创建的。它仅在启动过程中处于活动状态，因此累计活动时间很短。在您的实验中，可能创建了与线程 15 相似的另一个线程。

"Thread 1"（线程 1）在其整个时间内处于休眠状态。

其余线程花费其时间等待锁，这是 JVM 在内部同步自身的方式。这些线程包括用于 HotSpot 编译和垃圾收集的那些线程。本教程不讲述 JVM 系统的行为，但在另一教程 Java 和混合 Java-C++ 分析中讲述。

5. 单击 "Views"（视图）导航面板中的 "Functions"（函数）视图，或从菜单栏中选择 "Views"（视图）-> "Functions"（函数）。



"Functions"（函数）视图显示应用程序中函数的列表，以及每个函数的性能度量。列表最初按在每个函数中所用的 "Exclusive Total CPU Time"（独占总 CPU 时间）排序。"Functions"（函数）视图中还有许多来自 JVM 的函数，但是它们具有相对较低的度量。列表包括目标应用程序中的所有函数以及程序使用的任何共享对象。缺省情况下，最顶端的函数（即耗费资源最多的函数）处于选定状态。

右侧的 "Selection Details"（选择详细信息）窗口显示选定函数的所有已记录度量。

函数列表下的 "Called-by/Calls"（调用方/调用）面板提供有关选定函数的更多信息，并分为两个列表。"Called-by"（调用方）列表显示选定函数的调用方，度量值显示归属于其调用方的函数总度量。"Calls"（调用）列表显示选定函数的被调用方，并显示其被调用方的包含度量如何有助于选定函数的总度量。如果在 "Called-by/Calls"（调用方/调用）面板中双击任一列表中的函数，则在主 "Functions"（函数）视图中该函数将变为选定的函数。

6. 通过选择各种函数进行实验，以了解 "Functions"（函数）视图中的 "Called-by/Calls"（调用方/调用）面板和 "Selection Details"（选择详细信息）窗口如何随选择的变化而更新。

   "Selection Details"（选择详细信息）窗口显示大多数的函数来自 `jlowfruit.class`，如 "Load Object"（装入对象）字段中所示。

也可以通过单击列标题进行实验，以将排序从 "Exclusive Total CPU Time"（独占总 CPU 时间）更改为 "Inclusive Total CPU Time"（包含总 CPU 时间），或者按 "Name"（名称）排序。

7. 在 "Functions"（函数）视图中，对初始化任务的两个版本 `jlowfruit.init_bad（）` 和 `jlowfruit.init_good（）` 进行比较。

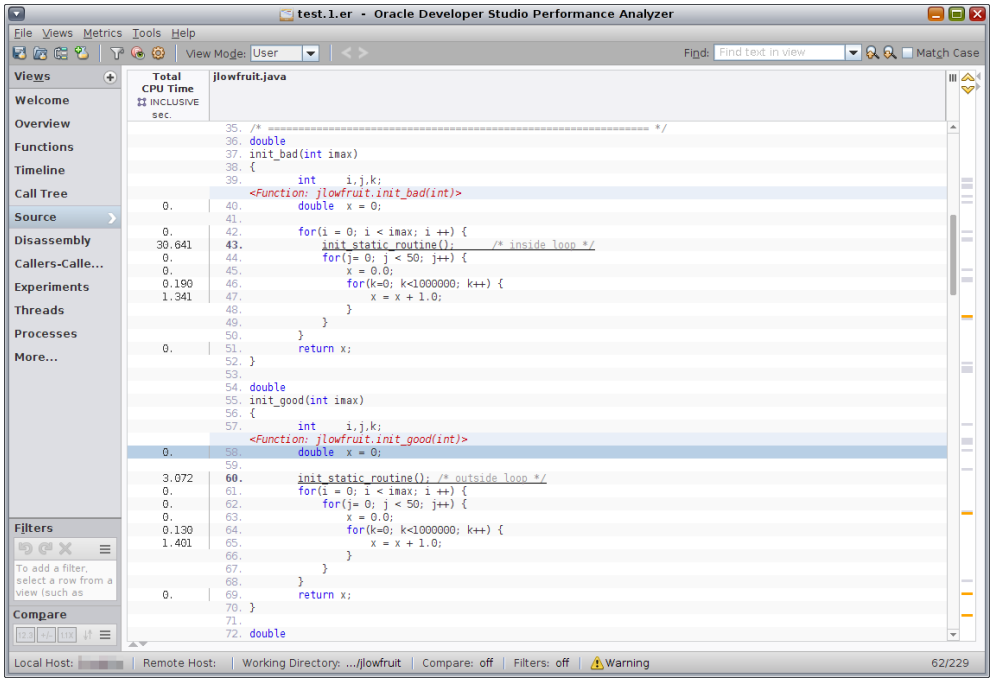
   可以看到，这两个函数具有大致相同的 "Exclusive Total CPU Time"（独占总 CPU 时间），但是包含时间截然不同。`jlowfruit.init_bad（）` 函数的运行速度较慢，因为它在被调用方中花费了时间。这两个函数调用同一被调用方，但是它们在该例程中所用的时间截然不同。通过检查这两个例程的源代码，您可以找出原因。

8. 选择函数 `jlowfruit.init_good（）`，然后单击 "Source"（源）视图，或者从菜单栏中选择 "Views"（视图）-> "Source"（源）。

9. 调整窗口以便为代码提供更多空间：单击上边缘的向下箭头折叠 "Called-by/Calls"（调用方/调用）面板，单击侧边缘的向右箭头折叠 "Selection Details"（选择详细信息）面板。

注 - 在教程的其余部分，您可能需要重新展开和重新折叠这些面板。

您应该向上滚动一点以便同时看到 `jlowfruit.init_bad（）` 和 `jlowfruit.init_good（）` 的源代码。"Source"（源）视图看起来应该与以下屏幕抓图类似。



请注意，对 `jlowfruit.init_static_routine()` 的调用在 `jlowfruit.init_good()` 中的循环之外，而 `jlowfruit.init_bad()` 对 `jlowfruit.init_static_routine()` 的调用则在循环之内。与好版本相比，差版本所用时间大约长十倍（与循环计数相对应）。

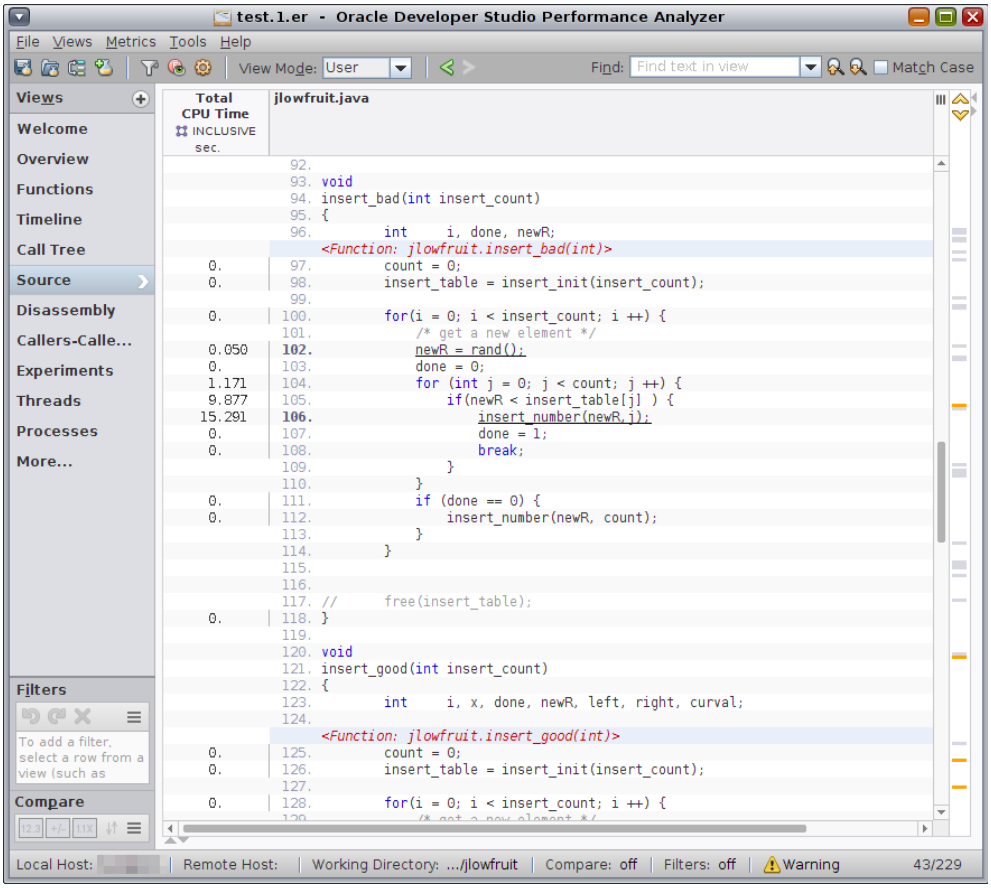
此示例不像看起来的那样笨拙。它基于真实的代码，可生成一个表，其中每行都有一个图标。虽然很容易看出在此示例中初始化不应在循环之内，但是在真实的代码中，初始化嵌入在库例程中且不易看出。

用于实现该代码的工具包提供两个库调用 (API)。第一个 API 将图标添加到表行，第二个 API 将图标向量添加到整个表。虽然使用第一个 API 更易于编码，但是每次添加图标时，工具包都要重新计算所有行的高度，以便为整个表设置正确的值。代码使用另一个 API 一次性添加所有图标时，高度的重新计算仅执行一次。

10. 现在，返回到 "Functions"（函数）视图，并查看插入任务的两个版本 `jlowfruit.insert_bad()` 和 `jlowfruit.insert_good()`。

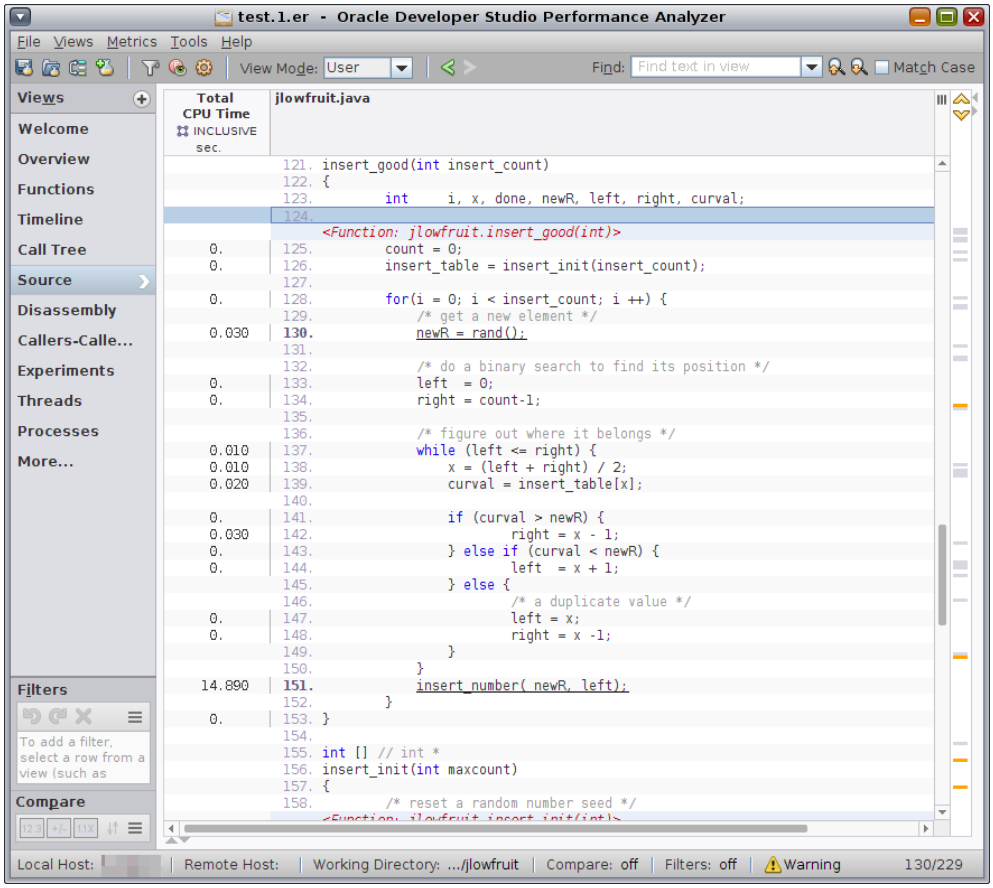
    请注意，"Exclusive Total CPU time"（独占总 CPU 时间）对 `jlowfruit.insert_bad()` 很重要，但是对 `jlowfruit.insert_good()` 则可以忽略。每个版本的包含时间和独占时间（表示被调用以将每个条目插入到列表中的函数 `jlowfruit.insert_number()` 中的时间）之间的差异是相同的。通过检查源代码，可以找出原因。

11. 选择 `jlowfruit.insert_bad()` 并切换到 "Source"（源）视图：



请注意时间（不包括对 `jlowfruit.insert_number（）` 的调用）花费在循环查找中，通过线性搜索找到正确的位置以插入新数字。

12. 现在向下滚动以查看 `jlowfruit.insert_good（）`。



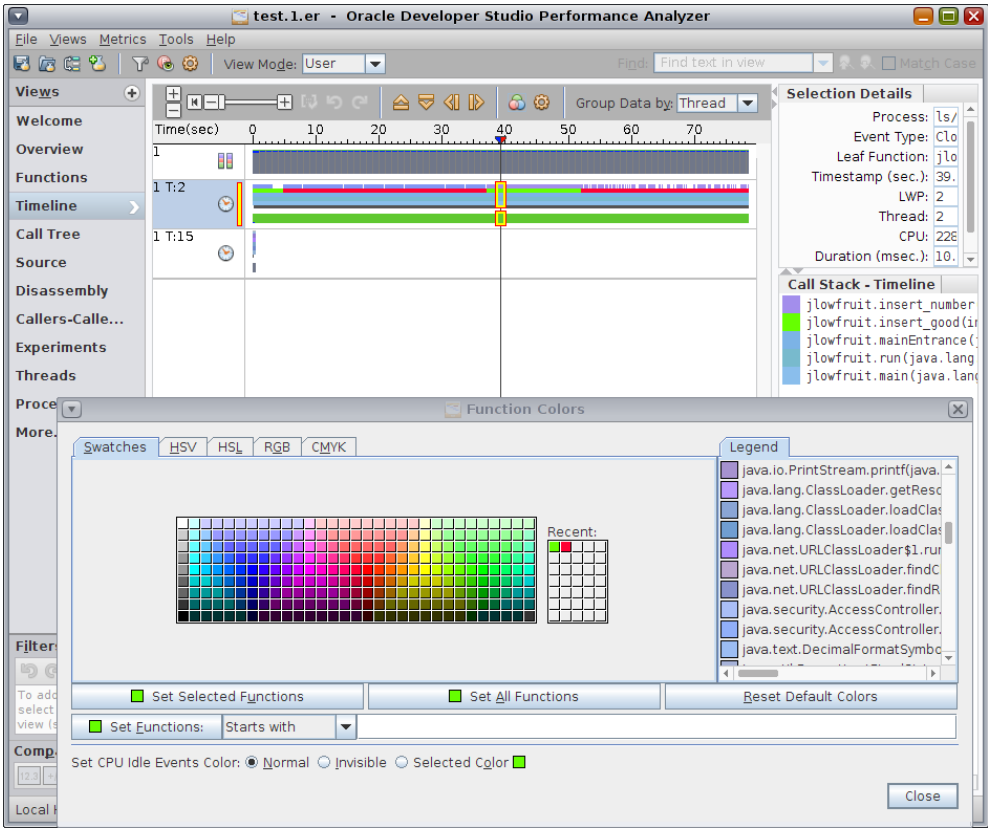
请注意代码更为复杂，因为它执行二进制搜索，以查找要插入的正确位置，但是所用的总时间（不包括对 `jlowfruit.insert_number（）` 的调用）比 `jlowfruit.insert_bad（）` 中少得多。此示例说明二进制搜索的效率比线性搜索更高。

您也可以在 "Timeline"（时间线）视图中以图形方式查看例程中的差异。

13. 单击 "Timeline"（时间线）视图，或者从菜单栏中选择 "Views"（视图）-> "Timeline"（时间线）。

    会将分析数据记录为一系列事件，每个事件表示每个线程的分析时钟的每一计时周期。"Timeline"（时间线）视图显示各个事件以及该事件中记录的调用堆栈。调用堆栈以调用堆栈中的帧列表形式显示，其中叶 PC（在事件发生的瞬间随即执行的指令）处于顶部，其次为调用它的调用点，等等。对于程序的主线程，调用堆栈的顶部始终为 `main`。

14. 在 "Timeline"（时间线）工具栏中，单击 "Call Stack Function Colors"（调用堆栈函数颜色）图标 以对函数着色，或者从菜单栏中选择 "Tools"（工具）-> "Function Colors"（函数颜色），然后将看到如下所示的对话框。



函数颜色已更改，以便为屏幕抓图更清晰地区分函数的好版本和差版本。`jlowfruit.init_bad()` 和 `jlowfruit.insert_bad()` 函数现在都呈红色，而 `jlowfruit.init_good()` 和 `jlowfruit.insert_good()` 现在都呈亮绿色。

15. 要使 "Timeline"（时间线）视图的外观类似，请在 "Function Colors"（函数颜色）对话框中执行以下操作：
    - 向下滚动 "Legend"（图例）中 java 方法的列表以查找 `jlowfruit.init_bad()` 方法。
    - 选择 `jlowfruit.init_bad()` 方法，单击 "Swatches"（色板）中的红色方块，然后单击 "Set Selected Functions"（设置所选函数）按钮。
    - 选择 `jlowfruit.insert_bad()` 方法，单击 "Swatches"（色板）中的红色方块，然后单击 "Set Selected Functions"（设置所选函数）按钮。

- 选择 `jlowfruit.init_good（）` 方法，单击 "Swatches"（色板）中的绿色方块，然后单击 "Set Selected Functions"（设置所选函数）按钮。
- 选择 `jlowfruit.insert_good（）` 方法，单击 "Swatches"（色板）中的绿色方块，然后单击 "Set Selected Functions"（设置所选函数）按钮。

16. 查看时间线的顶部条。

    时间线的顶部条是 "CPU Utilization Samples"（CPU 利用率抽样）条，将鼠标光标移动到第一列上时，可以在工具提示中看到它。"CPU Utilization Samples"（CPU 利用率抽样）条的每个段表示一秒的间隔，显示在该秒执行期间目标的资源使用率。

    在此示例中，段大部分为灰色并有一些绿色，反映出仅一小部分 "Total Time"（总时间）用于积累 "User CPU Time"（用户 CPU 时间）。"Selection Details"（选择详细信息）窗口显示颜色到微状态的映射，尽管它在屏幕抓图中不可见。

17. 查看时间线的第二个条。

    第二个条是 "Clock Profiling Call Stacks"（时钟分析调用堆栈）条，标记有 "1 T:2"，这表示进程 1 和线程 2（示例中的主用户线程）。"Clock Profiling Call Stacks"（时钟分析调用堆栈）条显示程序执行期间发生的事件的两个数据条。靠上的条显示调用堆栈的颜色编码表示形式，靠下的条显示每个事件的线程状态。此示例中的状态始终为 "User CPU Time"（用户 CPU 时间），因此它看起来是一条纯绿色的线。

    您应该会看到标记有不同线程编号的一个或两个其他条，但是它们在运行开始时将仅具有几个事件。

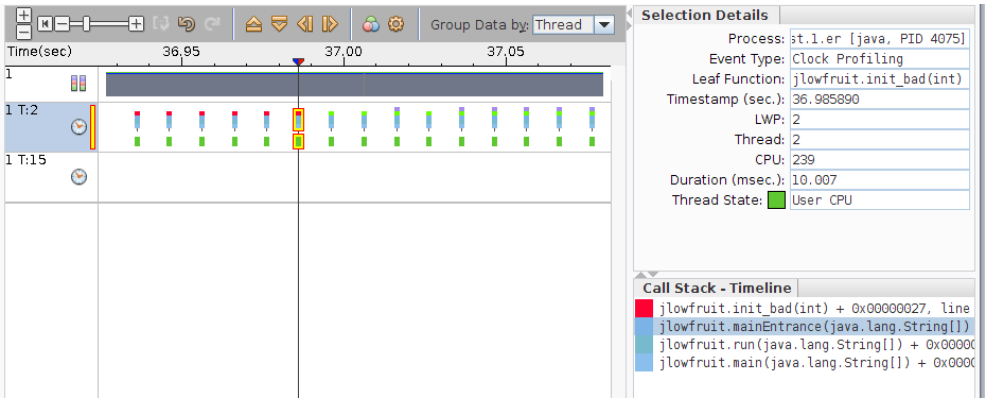
    如果单击该 "Clock Profiling Call Stacks"（时钟分析调用堆栈）条中的任何位置，则选择最近的事件，并在 "Selection Details"（选择详细信息）窗口中显示该事件的详细信息。从调用堆栈的图案中，可以看到在屏幕抓图中以亮绿色显示的 `jlowfruit.init_good（）` 和 `jlowfruit.insert_good（）` 例程中的时间比以红色显示的 `jlowfruit.init_bad（）` 和 `jlowfruit.insert_bad（）` 例程中的对应时间要短得多。

18. 选择与时间线中的好例程和差例程相对应的区域中的事件，并在 "Selection Details"（选择详细信息）窗口下的 "Call Stack - Timeline"（调用堆栈 - 时间线）窗口中查看调用堆栈。

    可以在 "Call Stack"（调用堆栈）窗口中选择任何帧，然后在 "Views"（视图）导航栏上选择 "Source"（源）视图，并转到该源代码行的源代码。也可以双击调用堆栈中的帧以转到 "Source"（源）视图，或者右键单击调用堆栈中的帧并从弹出菜单中进行选择。

19. 通过时间线顶部的滑块、使用 + 键或者通过用鼠标双击，均可以放大事件。

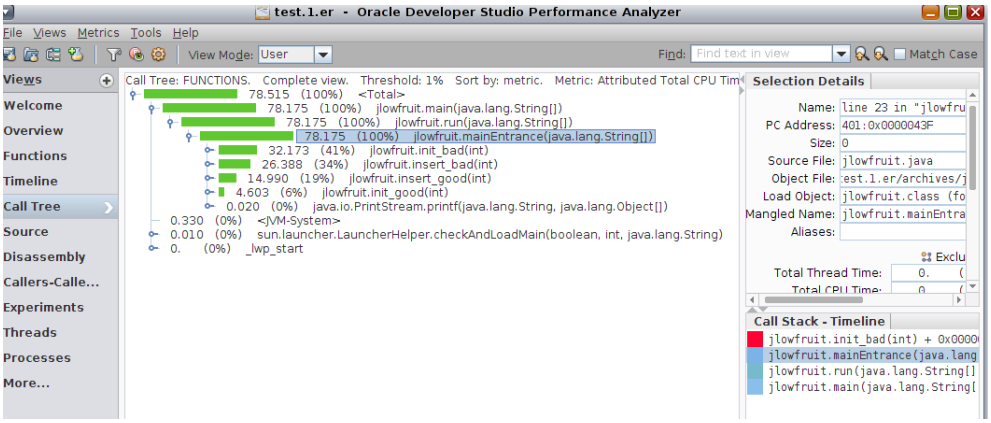
    如果放大得足够大，则可以看到显示的数据不是连续的，而是由离散的事件组成，每个分析计时周期（在此示例中约为 10 ms）都有一个事件。



按 F1 键可查看帮助以了解有关 "Timeline"（时间线）视图的更多信息。

20. 单击 "Call Tree"（调用树）视图或者选择 "Views"（视图）-> "Call Tree"（调用树），以查看程序的结构。

    "Call Tree"（调用树）视图显示程序的动态调用图，注释有性能信息。



Java 和混合 Java-C++ 分析

本章包含以下主题。

关于 Java-C++ 分析教程

本教程说明 Oracle Developer Studio 性能分析器的 Java 分析功能。它介绍了如何使用样例代码在性能分析器中执行以下操作：

- 检查各种数据视图中的性能数据，包括 "Overview"（概述）页面以及 "Threads"（线程）、"Functions"（函数）和 "Timeline"（时间线）视图。
- 查看 Java 代码和 C++ 代码的 "Source"（源）和 "Disassembly"（反汇编）。
- 了解 "User Mode"（用户模式）、"Expert Mode"（专家模式）和 "Machine Mode"（计算机模式）之间的差异。
- 深入了解执行程序的 JVM 的行为并查看任何 HotSpot 编译的方法的已生成本机代码。
- 了解如何通过用户代码调用垃圾收集器以及如何触发 HotSpot 编译器。

`jsynprog` 是 Java 程序，具有 Java 程序的许多典型子任务。该程序还装入 C++ 共享对象并从该对象调用各种例程，以显示从 Java 代码到动态装入的 C++ 库中的本机代码的无缝转换以及反向转换。

`jsynprog.main` 是主方法，调用不同类中的函数。它通过 Java 本地接口 (Java Native Interface, JNI) 调用来使用 `gethrtime` 和 `gethrvtime` 以对其自己的行为计时，并写入具有其自己的计时的记帐文件以及将消息写入 `stdout`。

`jsynprog.main` 具有许多方法：

- Routine.memalloc 执行内存分配并触发垃圾收集
- Routine.add_int 执行整型加法
- Routine.add_double 执行双精度（浮点）加法
- Sub_Routine.add_int 是派生调用，覆盖 Routine.add_int
- Routine.has_inner_class 定义内部类并使用该类
- Routine.recurse 显示直接递归
- Routine.recursedeep 执行深度递归，显示工具如何处理截断的堆栈
- Routine.bounce 显示间接递归，其中 bounce 调用 bounce_b，后者又会再回调 bounce
- Routine.array_op 执行数组运算
- Routine.vector_op 执行向量运算
- Routine.sys_op 使用系统类中的方法
- jsynprog.jni_JavaJavaC：Java 方法调用其他 Java 方法，而后者调用 C 函数
- jsynprog.JavaCJava：Java 方法调用 C 函数，而该函数又会调用某个 Java 方法
- jsynprog.JavaCC：Java 调用 C 函数，而该函数会调用另一个 C 函数

这些方法中的一些方法从另一些方法进行调用，所以它们并不是全部表示顶级任务。

在记录的实验中看到的数据将与此处显示的数据不同。用于教程中屏幕抓图的实验是在运行 Oracle Solaris 11.3 的 SPARC T5 系统上记录的。来自运行 Oracle Solaris 或 Linux 的 x86 系统的数据将会有所不同。此外，数据收集本质上是统计性的，随实验的不同而不同，即使运行在同一系统和 OS 上也是如此。

您看到的性能分析器窗口配置可能不会与屏幕抓图完全匹配。通过性能分析器，可以拖动窗口各部分之间的分隔条，折叠各部分以及调整窗口大小。性能分析器记录其配置，并在下次运行时使用相同的配置。在捕获教程所示的屏幕抓图的过程中进行了许多配置更改。

设置 jsynprog 样例代码

开始之前：

查看以下内容以了解有关获取代码和设置您的环境的信息。

- “获取教程的样例代码” [10]
- “为教程设置环境” [10]

您可能需要先浏览 Java 分析简介中的入门教程以熟悉性能分析器。

1. 使用以下命令将 jsynprog 目录的内容复制到您自己的专用工作区：

   ```
   % cp -r OracleDeveloperStudio12.5-Samples/PerformanceAnalyzer/jsynprog directory
   ```

   其中 *directory* 是您所使用的工作目录。

2. 转到该工作目录副本。

```
% cd directory/jsynprog
```

3. 生成目标可执行文件。

```
% make clobber

% make
```

注 - 仅当之前在目录中运行了 make 时才需要使用 clobber 子命令，但该子命令在任何情况下都可以放心地使用。

在运行 make 之后，目录将包含教程中要使用的目标应用程序，即名为 jsynprog.class 的 Java 类文件和名为 libcloop.so 的共享对象，该对象包含将从 Java 程序动态装入和调用的 C++ 代码。

提示 - 如果您愿意，可以编辑 Makefile 以执行以下操作：使用 GNU 编译器而不是缺省的 Oracle Developer Studio 编译器；以 32 位而不是缺省的 64 位编译；以及添加不同的编译器标志。

通过 jsynprog 收集数据

收集数据的最简单方式是在 jsynprog 目录中运行以下命令：

```
% make collect
```

Makefile 的 collect 目标将启动 collect 命令并记录实验。缺省情况下，实验名为 test.1.er。

缺省情况下，collect 目标为 JVM 指定选项 -J "-Xmx100m -XX:ParallelGCThreads=10" 并收集时钟分析数据。

或者，可以使用性能分析器的 "Profile Application"（分析应用程序）对话框记录数据。按照 Java 简介教程中的过程“使用性能分析器从 jlowfruit 中收集数据” [28] 进行操作并在 "Arguments"（参数）字段中指定 jsynprog，而不是 jlowfruit。

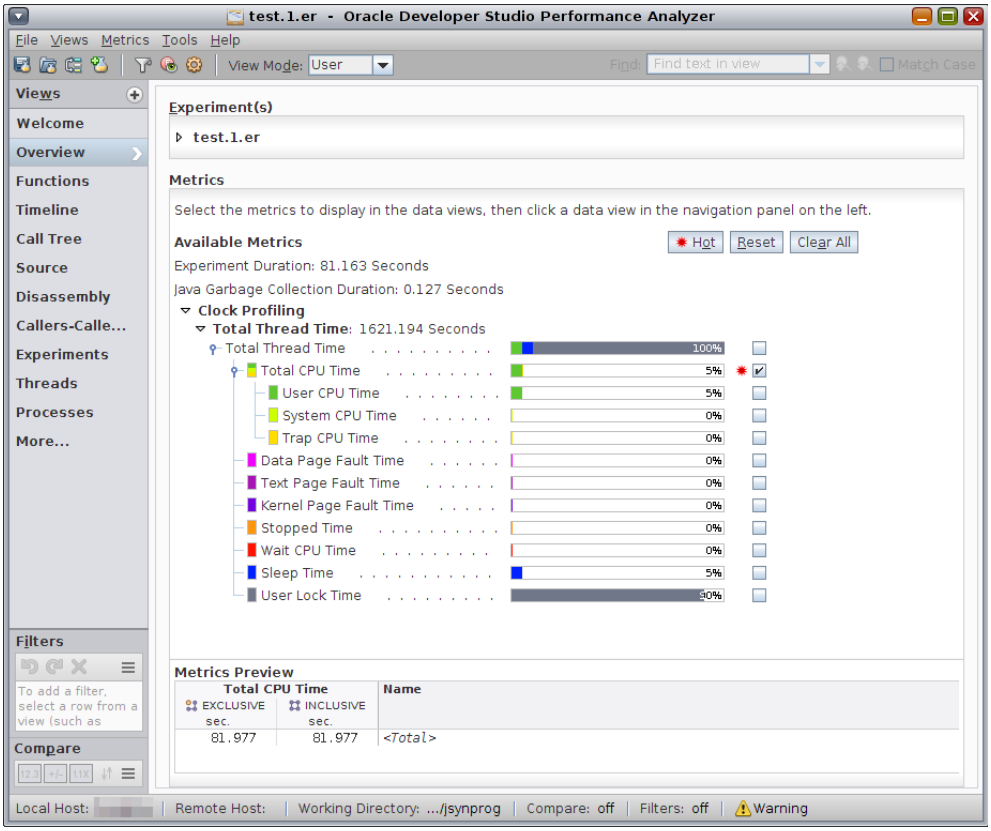
检查 jsynprog 数据

该过程假定您已经按照上一节中所述创建了实验。

1. 从 jsynprog 目录启动性能分析器并按如下所示装入实验，如果实验名不是 test.1.er，则指定您的实验名称。

   ```
   % analyzer test.1.er
   ```

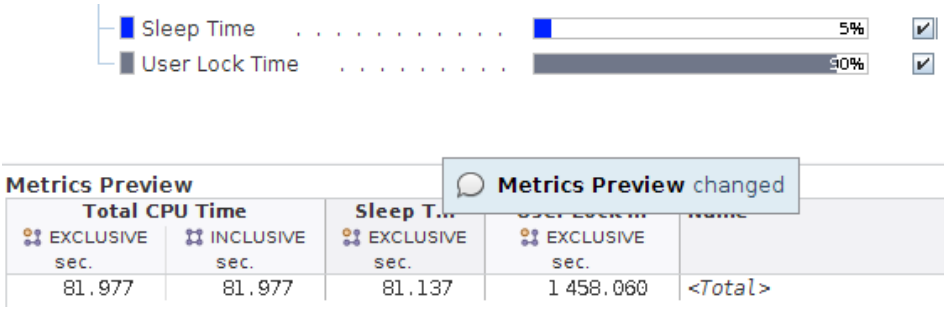
   实验打开时，性能分析器将显示 "Overview"（概述）页面。



   请注意，性能分析器的工具栏现在具有查看模式选择器，该选择器最初设置为 "User Mode"（用户模式），显示程序的用户模型。

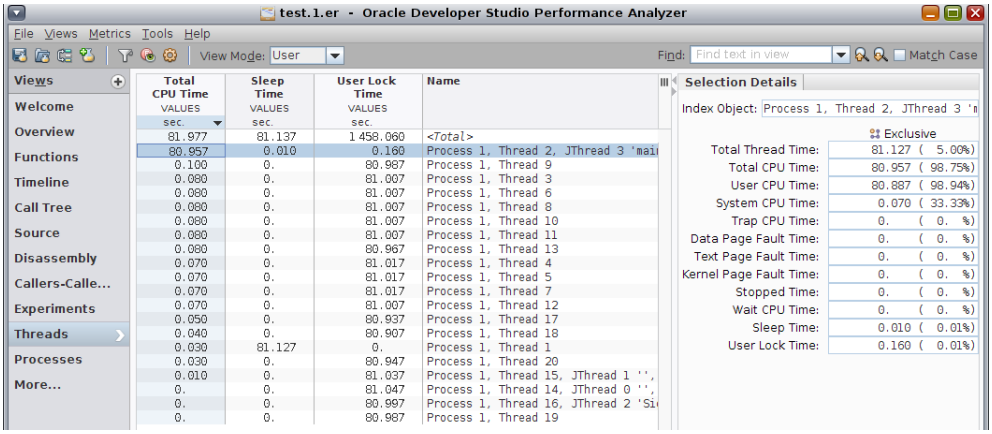
   "Overview"（概述）显示实验运行了大约 81 秒，但是使用的总时间多于 1600 秒，表示进程中平均存在 20 个线程。

2. 选中 "Sleep Time"（休眠时间）和 "User Lock Time"（用户锁定时间）度量的复选框以将其添加到数据视图。



请注意，"Metrics Preview"（度量预览）将更新以显示添加了这些度量后数据视图的样子。

3. 选择导航面板中的 "Threads"（线程）视图，您将看到线程的数据：



仅 "Thread 2"（线程 2）积累了大量 "Total CPU"（总 CPU）时间。其他每个线程针对 "Total CPU"（总 CPU）时间仅具有几个分析事件。

4. 选择 "Threads"（线程）视图中的任何线程并在右侧的 "Selection Details"（选择详细信息）窗口中查看该线程的所有信息。

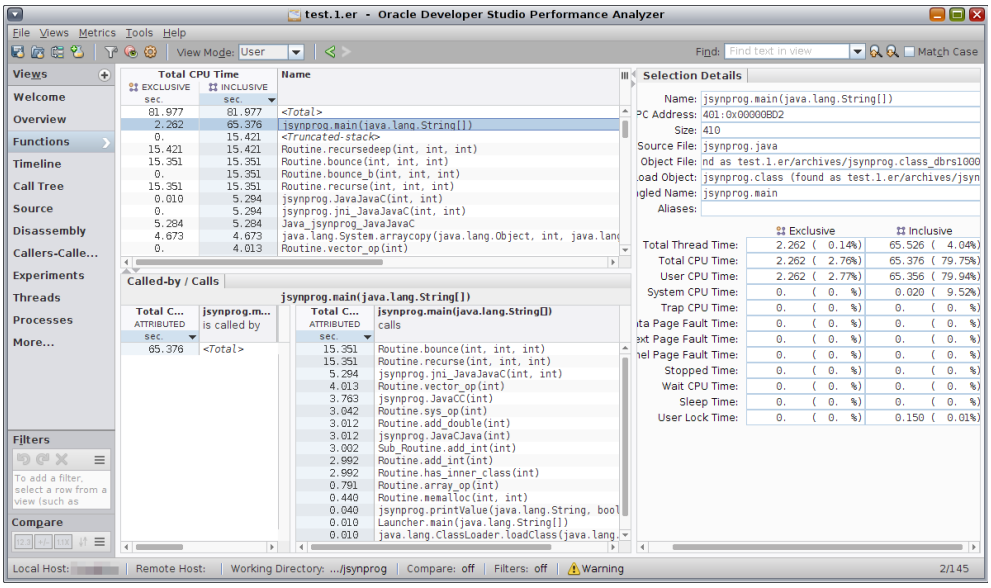
   您应该看到除了 "Thread 1"（线程 1）和 "Thread 2"（线程 2）外，几乎所有线程都在 "User Lock"（用户锁定）状态下花费了其所有时间。这显示了 JVM 如何在内部同步自身。"Thread 1"（线程 1）启动用户 Java 代码，然后休眠，直到其完成。

5. 返回 "Overview"（概述）并取消选中 "Sleep Time"（休眠时间）和 "User Lock Time"（用户锁定时间）。

6. 选择导航面板中的 "Functions"（函数）视图，然后单击列标题以便按 "Exclusive Total CPU Time"（独占总 CPU 时间）、"Inclusive Total CPU Time"（包含总 CPU 时间）或 "Name"（名称）排序。

   可以按降序或升序排序。

   保持列表按 "Inclusive Total CPU Time"（包含总 CPU 时间）降序排序，然后选择最顶部的函数 `jsynprog.main（）`。该例程是 JVM 调用的初始例程以启动执行。



   请注意，"Functions"（函数）视图底部的 "Called-by/Calls"（调用方/调用）面板显示 `jsynprog.main（）` 函数被 `<Total>` 调用，表示其位于堆栈顶部。

   该面板的 "Calls"（调用）侧显示 `jsynprog.main（）` 调用各种不同例程，“关于 Java-C++ 分析教程” [43] 中所示的直接从主例程调用的每个子任务使用其中一个例程。该列表还包括几个其他例程。

检查混合 Java 和 C++ 代码

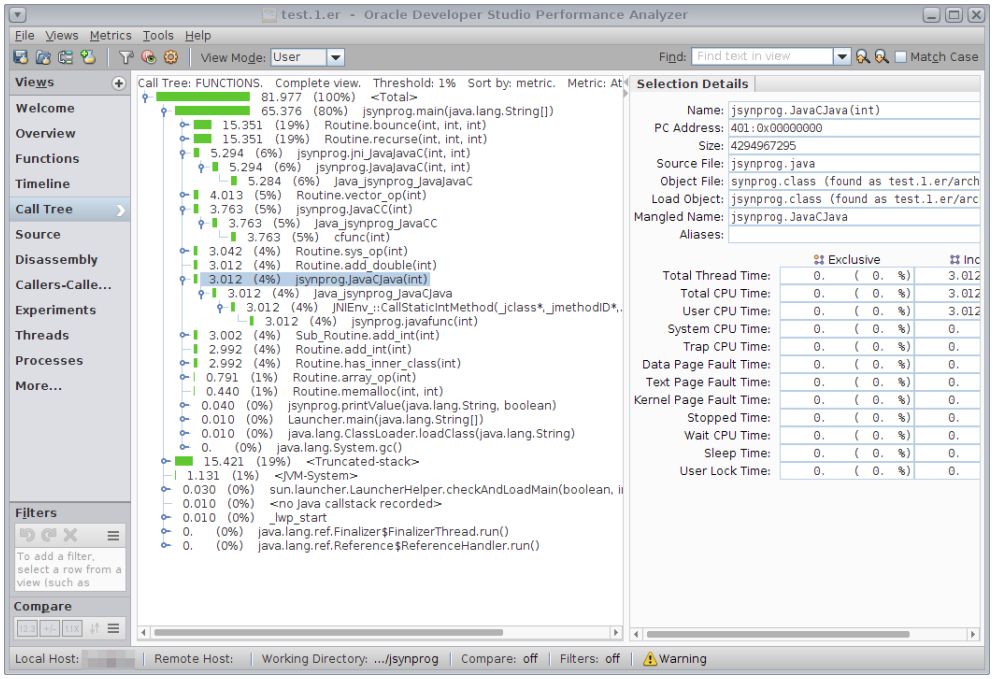
本节讲述 "Call Tree"（调用树）视图和 "Source"（源）视图，并介绍如何查看 Java 与 C++ 的调用相互之间的关系。本节还介绍如何向导航面板添加 "Disassembly"（反汇编）视图。

1. 依次选择 "Function"（函数）视图中列表顶部的每个函数，然后在 "Selection Details"（选择详细信息）窗口中检查详细信息。

请注意，对于一些函数，"Source File"（源文件）被报告为 `jsynprog.java`，而对于一些其他函数则报告为 `cloop.cc`。这是因为 `jsynprog` 程序装入了名为 `libcloop.so` 的 C++ 共享对象，其是从 `cloop.cc` C++ 源文件构建的。性能分析器无缝报告从 Java 到 C++ 的调用，反之亦然。

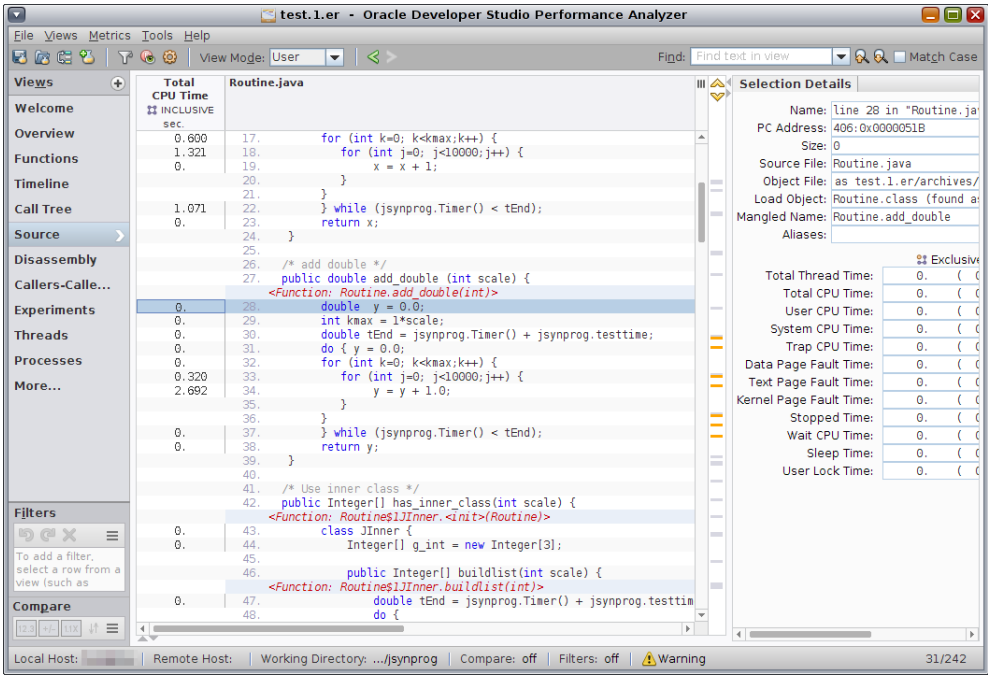
2. 选择导航面板中的 "Call Tree"（调用树）。

   "Call Tree"（调用树）视图以图形方式显示如何在 Java 和 C++ 之间进行这些调用。

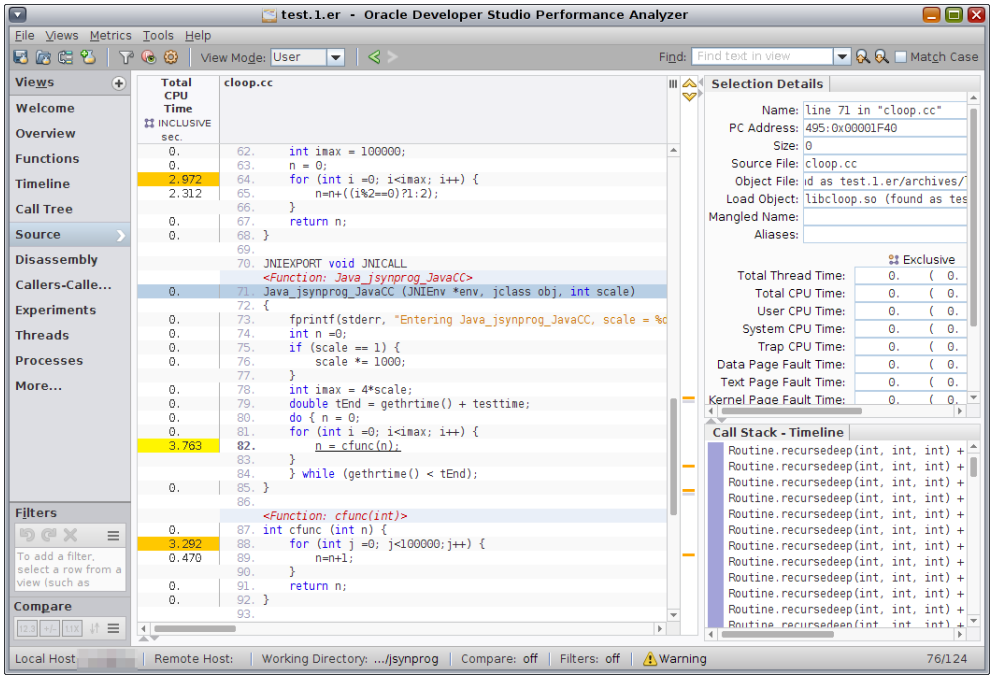


3. 在 "Call Tree"（调用树）视图中，执行以下操作来查看从 Java 到 C++ 以及再回到 Java 的调用：
   - 展开引用名称中包含 "C" 的各种函数的行。
   - 选择 `jsynprog.JavaCC（）` 对应的行。该函数来自 Java 代码，但是它调用到来自 C++ 代码的 `Java_jsynprog_JavaCC（）` 中。
   - 选择 `jsynprog.JavaCJava（）` 对应的行。该函数也来自 Java 代码，但是调用属于 C++ 代码的 `Java_jsynprog_JavaCJava（）`。该函数调用到 `JNIEnv_::CallStaticIntMethod（）` 的 C++ 方法中，JNIEnv_::CallStaticIntMethod 回调到 Java 来调用方法 `jsynprog.javafunc（）`。
4. 选择 Java 或 C++ 的方法并切换到 "Source"（源）视图，来查看以适当语言显示的源代码以及性能度量。

   选择 Java 方法后的 "Source"（源）视图的示例如下所示。

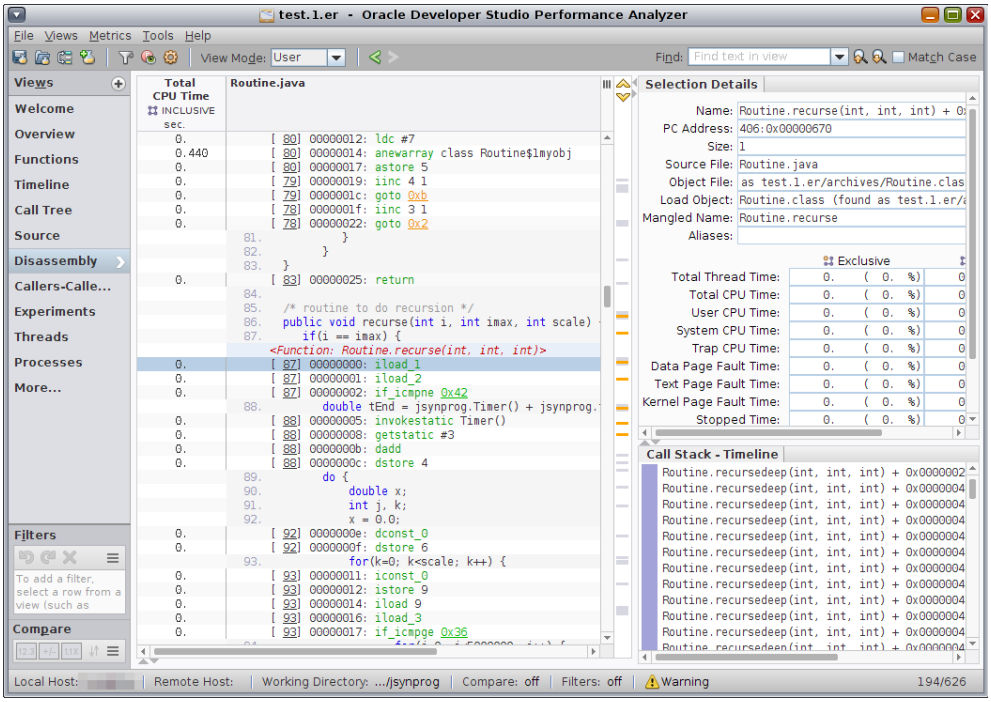


选择 C++ 方法后的 "Source"（源）视图的示例如下所示。

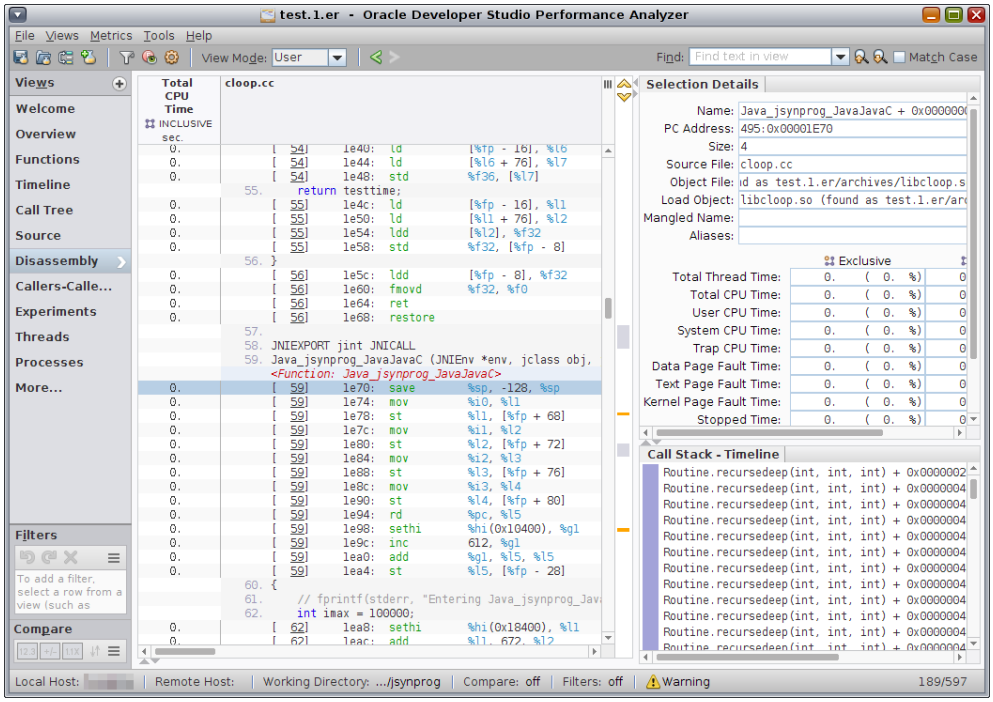


5. 如果尚未在导航面板中看到 "Disassembly"（反汇编）标签，要添加该视图，请单击导航面板顶部 "Views"（视图）标签旁边的 + 按钮，然后选中 "Disassembly"（反汇编）复选框。

   此时将显示最后选择的函数的 "Disassembly"（反汇编）视图。对于 Java 函数，" Disassembly"（反汇编）视图显示 Java 字节代码，如下面的屏幕抓图中所示。



对于 C++ 函数，"Disassembly"（反汇编）视图显示本机计算机代码，如下面的屏幕抓图中所示。



下一节进一步使用 "Disassembly"（反汇编）视图。

了解 JVM 行为

本节介绍如何通过使用过滤器、"Expert Mode"（专家模式）和 "Machine Mode"（计算机模式）检查 JVM 中出现的内容。

1. 选择 "Functions"（函数）视图并查找名为 `<JVM-System>` 的例程。

   如果您键入 **`<JVM`** 并按 Enter 键，则可以使用工具栏中的 "Find"（查找）工具非常快速地查找该例程。

   在此实验中，`<JVM-System>` 花费了大约一秒的 "Total CPU Time"（CPU 总时间）。`<JVM-System>` 函数中的时间表示 JVM 的工作，而不是用户代码的工作。
2. 右键单击 `<JVM-System>` 并选择 "Add Filter: Include only stacks containing the selected functions"（添加过滤器：仅包括含有所选函数的堆栈）。

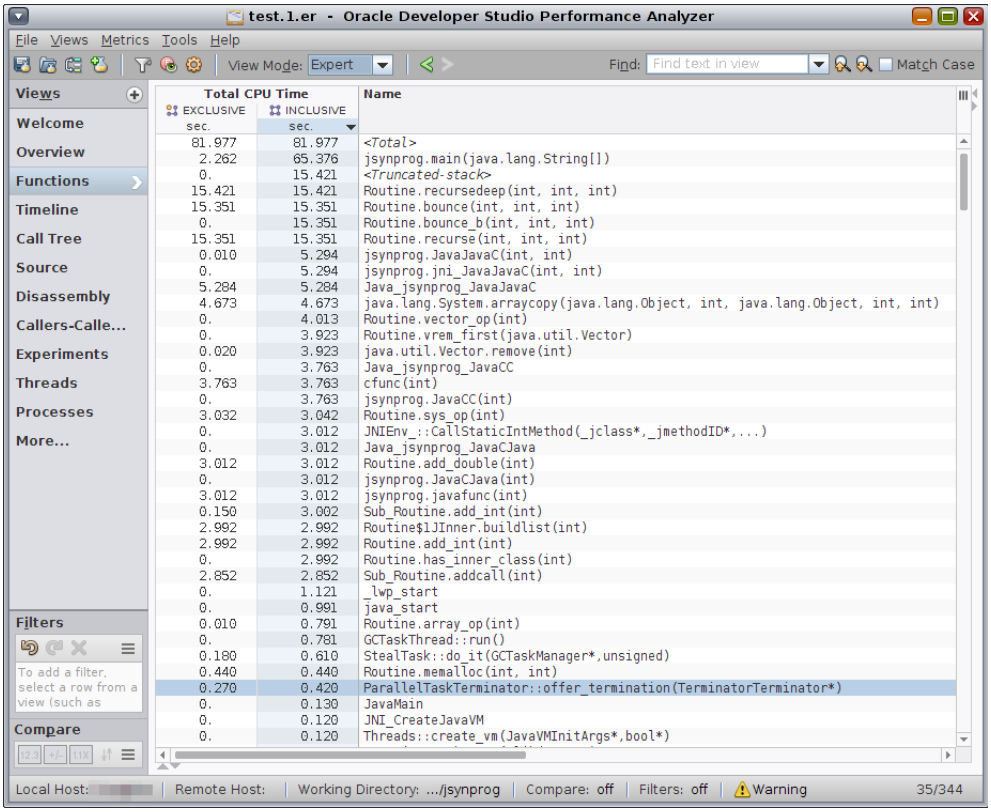
   请注意，导航面板下面的过滤器面板以前显示 "No Active Filters"（没有活动过滤器），现在显示 "1 Active Filter"（1 个活动过滤器）以及您添加的过滤器的名称。"Functions"（函数）视图将刷新，从而仅剩余 `<JVM-System>`。

3. 在性能分析器工具栏中，将查看模式选择器从 "User Mode"（用户模式）更改为 "Expert Mode"（专家模式）。

   "Functions"（函数）视图将刷新以显示由 <JVM-System> 时间表示的许多函数。函数 <JVM-System> 自身不再可见。

4. 通过单击 "Active Filters"（活动过滤器）面板中的 X 来删除过滤器。

   "Functions"（函数）视图将刷新以再次显示用户函数，但是在 <JVM-System> 函数不可见时，<JVM-System> 表示的函数也仍可见。

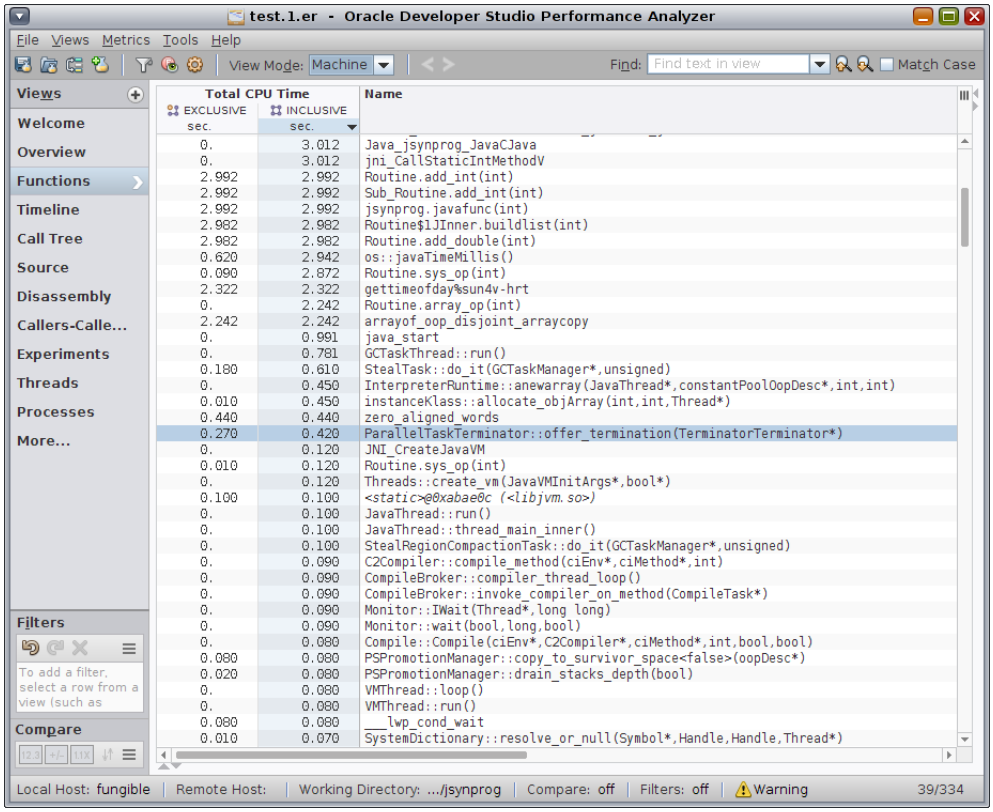


   请注意，您不需要执行过滤来展开 <JVM-System>。该过程包括过滤以便更容易地显示 "User Mode"（用户模式）与 "Expert Mode"（专家模式）之间的差异。

   总结："User Mode"（用户模式）显示所有用户函数，但是将 JVM 中花费的所有时间聚集到 <JVM-System> 中，而 "Expert Mode"（专家模式）展开该 <JVM-System> 聚集。

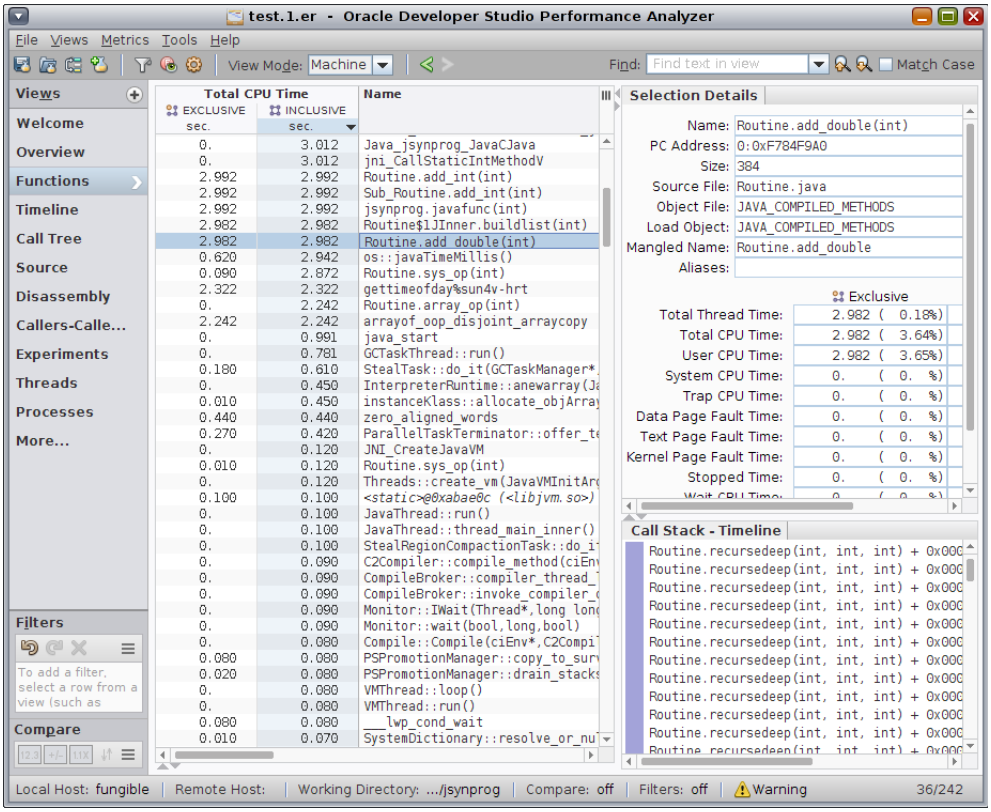
   接下来，您可以了解 "Machine Mode"（计算机模式）。

5. 在查看模式列表中选择 "Machine Mode"（计算机模式）。



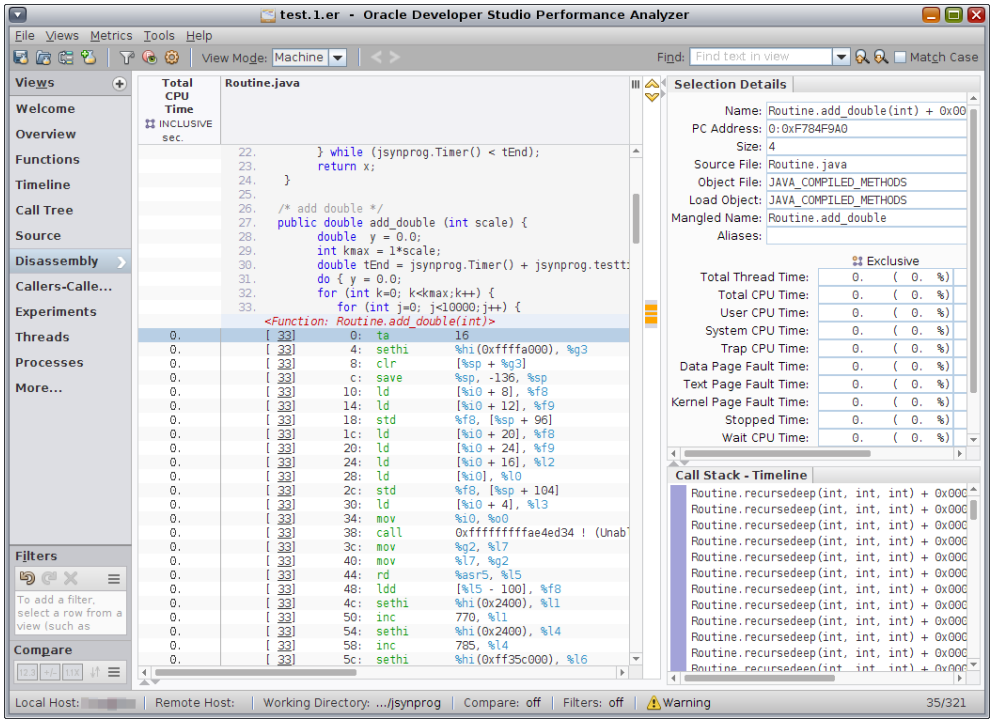
在 "Machine Mode"（计算机模式）中，解释的任何用户方法不按名称显示在 "Functions"（函数）视图中。解释的方法中花费的时间将聚集到 `Interpreter` 条目中，其表示 JVM 中以解释方式执行 Java 字节代码的部分。

但是，在 "Machine Mode"（计算机模式）下，"Functions"（函数）视图显示 HotSpot 编译的任何用户方法。如果选择 `Routine.add_int（）` 等编译的方法，" Selection Details"（选择详细信息）窗口将该方法的 Java 源文件显示为 "Source File"（源文件），但是 "Object File"（对象文件）和 "Load Object"（装入对象）显示为 `JAVA_COMPILED_METHODS`。



6. 仍在 "Machine Mode"（计算机模式）下时，如果在 "Functions"（函数）视图中选择了编译的方法则切换到 "Disassembly"（反汇编）视图。

   "Disassembly"（反汇编）视图显示 HotSpot 编译器生成的计算机代码。可以查看代码上面列标题中的 "Source File"（源文件）、"Object File"（对象文件）和 "Load Object"（装入对象）名称。



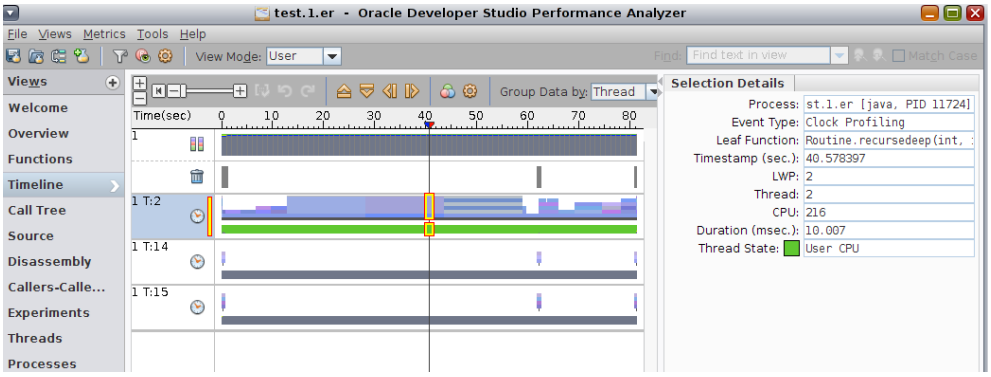
大多数可见行上显示的 "Total CPU Time"（CPU 总时间）为零，因为该函数中的大多数工作将在代码中进一步执行。

继续下一节。

了解 Java 垃圾收集器行为

该过程介绍如何使用 "Timeline"（时间线）视图以及查看模式设置对 "Timeline"（时间线）的影响，同时检查触发 Java 垃圾收集的活动。

1. 将查看模式设置为 "User Mode"（用户模式）并在导航面板中选择 "Timeline"（时间线）视图，从而显示此混合 Java/本机应用程序 `jsynprog` 的执行详细信息。



您应该看到顶部的 "CPU Utilization Samples"（CPU 利用率抽样）条以及三个线程的分析数据。在屏幕抓图中，您可以看到进程 1 以及线程 2、14、15 的数据。您看到的编号和线程数可能取决于 OS、系统以及使用的 Java 版本。

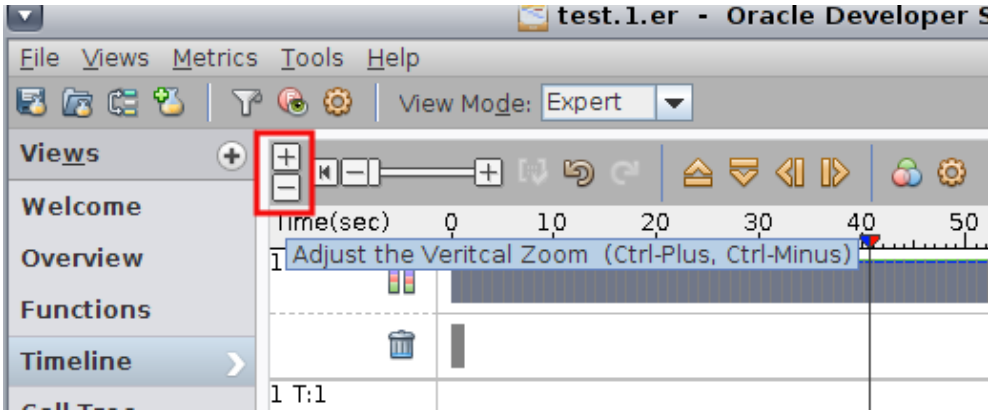
仅第一个线程（示例中标记为 T:2 的 "Thread 2"（线程 2））将其微状态显示为 "User CPU"（用户 CPU）。其他两个线程花费其所有时间来等待 "User Lock"（用户锁定），这是 JVM 同步的一部分。

2. 将查看模式设置为 "Expert Mode"（专家模式）。

   "Timeline"（时间线）视图现在应该显示更多线程，尽管用户线程 T:2 看起来几乎未更改。

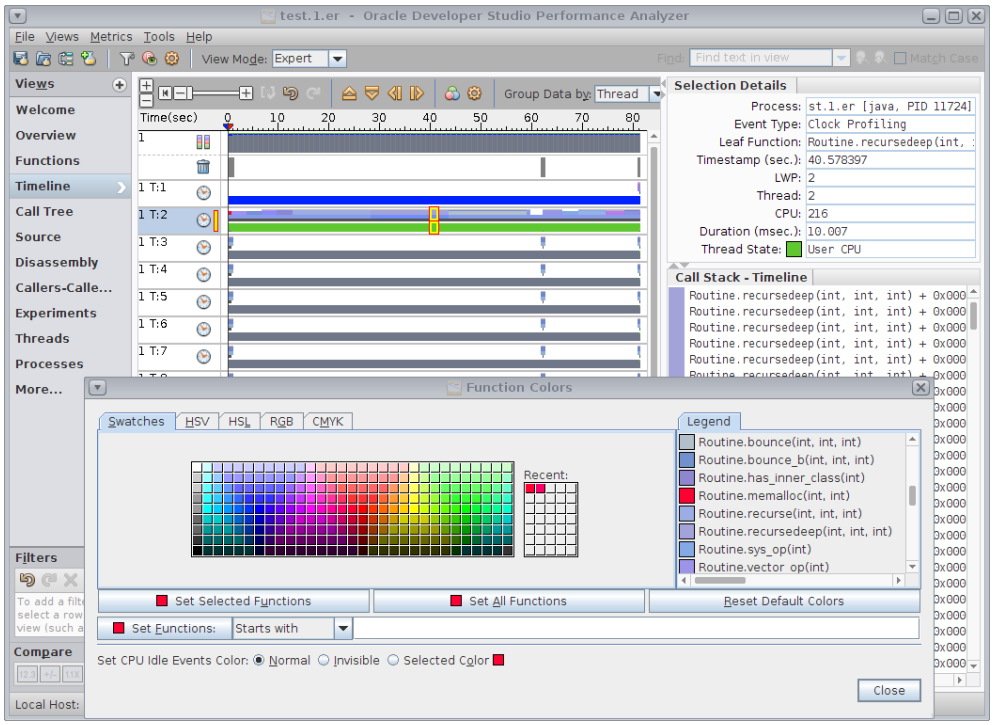
3. 使用时间线顶部的垂直缩放控件来调整缩放，从而您可以看到所有线程。

   在以下屏幕抓图中，垂直缩放控件如红色框所示。单击减号按钮来减少线程行的高度，直到您可以看到所有二十个线程。



4. 单击 "Timeline"（时间线）工具栏中的 "Call Stack Function Colors"（调用堆栈函数颜色）图标 将函数 `Routine.memalloc()` 的颜色设置为红色。

   在 "Function Colors"（函数颜色）对话框中，选择 "Legend"（图例）中的 `Routine.memalloc()` 函数，单击 "Swatches"（色板）中的红色框，然后单击 "Set Selected Functions"（设置所选函数）。

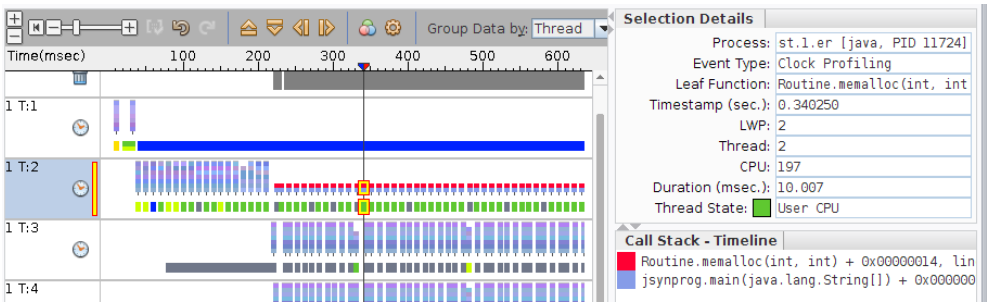


   请注意，"Thread 2"（线程 2）在其堆栈顶部具有一个红条。该区域表示 `Routine.memalloc()` 例程处于运行状态的时间部分。

   您可能需要垂直缩小以查看调用堆栈的更多帧，并水平放大到关注的时间区域。

5. 使用 "Timeline"（时间线）工具栏中的水平滑块进行足够放大以查看线程 T:2 中的单个事件。

   还可以通过双击或按键盘上的 + 键进行缩放。



时间线的每行实际包括三个数据条。顶部条表示该事件的调用堆栈。中间条显示黑色勾选标记，其中事件一起过于紧密发生而无法显示所有事件。换句话说，看到勾选标记时，您将知道该空间有多个事件。

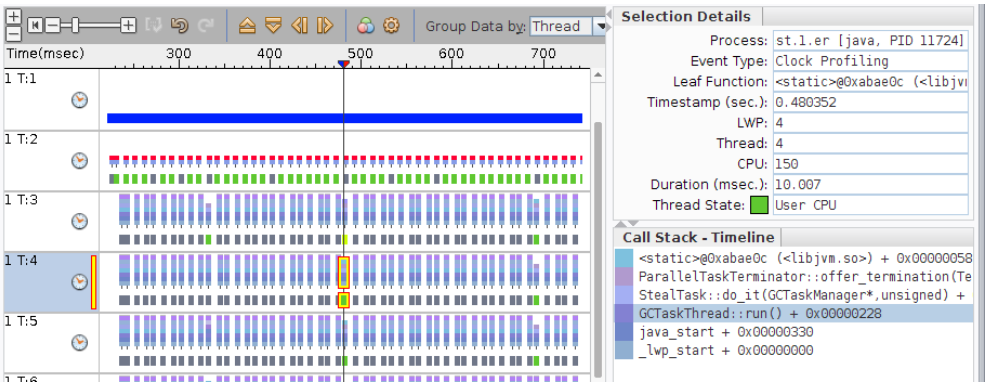
下面的条指示事件状态。对于 T:2，下面的条为绿色，指示正在使用 "User CPU Time"（用户 CPU 时间）。对于线程 3 到 12，下面的条为灰色，指示 "User Lock Time"（用户锁定时间）。

然而请注意，当用户线程 T:2 处于 `Routine.memalloc`（显示为红色的例程）中时，线程 3 到线程 12 将具有许多事件聚集在一起同时发生。

6. 通过执行以下操作，放大到 `Routine.memalloc` 区域并过滤以仅包括该区域：
   - 单击靠近上面具有红色函数调用的 `Routine.memalloc` 区域开始处的 T:2 条。
   - 单击并将鼠标拖到靠近该区域末尾，调用堆栈顶部的红色在该处结束。
   - 右键单击并选择 "Zoom"（缩放）-> "To Selected Time Range"（至所选的时间范围）。
   - 在仍选中该范围的情况下，右键单击并选择 "Add Filter: Include only events intersecting selected time range"（添加过滤器：仅包含贯穿所选时间范围的事件）。

   缩放后，您可以看到线程 3-12 中有一些绿色的事件状态，指示 "User CPU"（用户 CPU）时间，并且甚至有几个红色的事件状态，指示 "Wait CPU Time"（等待 CPU 时间）。
7. 单击线程 3-12 上的任何事件，您在 "Call Stack"（调用堆栈）面板中看到每个线程的事件在堆栈中包括 `GCTaskThread::run（）`。

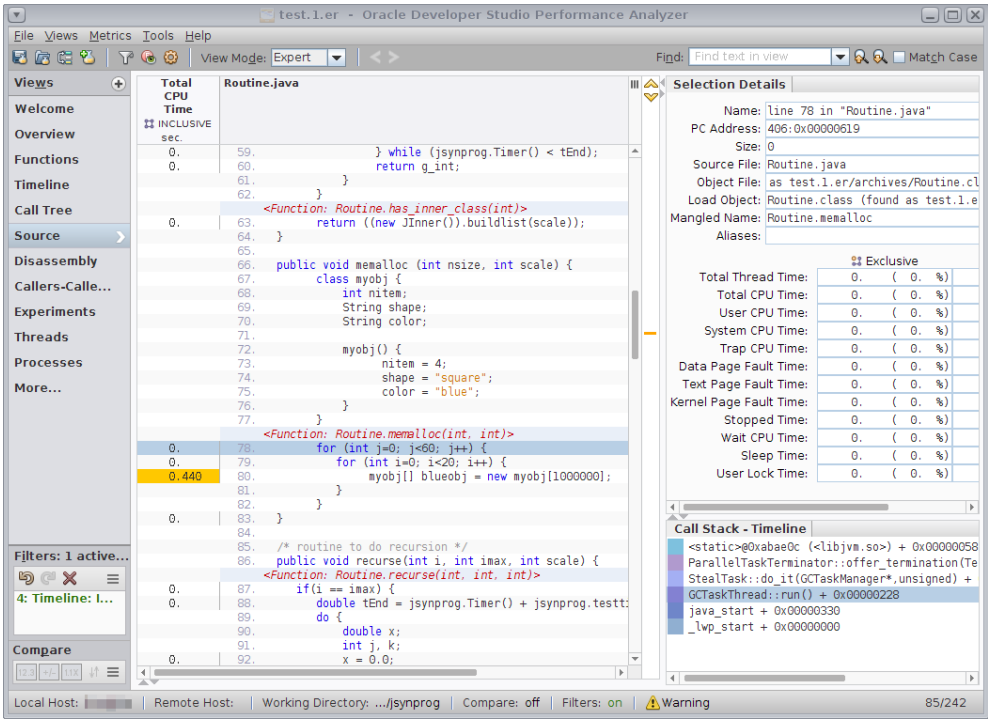
   那些线程表示 JVM 用于运行垃圾收集的线程。GC 线程不会占用大量 "User CPU Time"（用户 CPU 时间），而是仅在用户线程处于 `Routine.memalloc` 中时运行。



8. 返回 "Functions"（函数）视图并单击 "Incl.Total CPU"（包含总 CPU）列标题以按包含 "Total CPU Time"（总 CPU 时间）进行排序。

   您应该看到其中一个顶部函数为 `GCTaskThread::run（）` 函数。这样您可以知道用户任务 `Routine.memalloc` 正在以某种方式触发垃圾收集。

9. 选择 `Routine.memalloc` 函数并切换到 "Source"（源）视图。



从源代码的此片段，可以容易地看到为何在触发垃圾收集。该代码分配一个包含一百万个对象的数组并在相同位置存储这些对象的指针，每个对象都进行循环。这将呈现未使用的旧对象，从而其变为垃圾。

继续下一节。

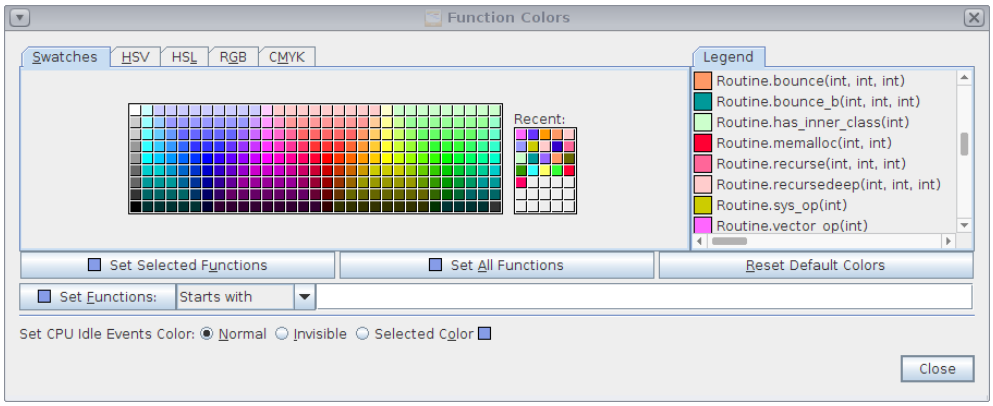
了解 Java HotSpot 编译器行为

该过程接续上一节，介绍如何使用 "Timeline"（时间线）和 "Threads"（线程）视图来过滤和查找负责 HotSpot 编译的线程。

1. 选择 "Timeline"（时间线）视图并通过单击 "Active Filters"（活动过滤器）面板中的 X 来删除过滤器，然后通过按键盘上的 0 将水平缩放重置为缺省值。

   还可以单击 "Timeline"（时间线）工具栏中水平滑块前面的 |< 按钮，或者在 "Timeline"（时间线）中右键单击并选择 "Reset"（重置）。
2. 再次打开 "Function Colors"（函数颜色）对话框并为每个 `Routine.*` 函数选取不同颜色。

在 "Timeline"（时间线）视图中，颜色更改出现在线程 2 的调用堆栈中。



3. 查看您看到线程 2 中颜色更改的时间段中 "Timeline"（时间线）的所有线程。

   您应该看到有一些线程具有几乎与线程 2 中的颜色更改同时发生的事件的图案。在此示例中，它们是线程 17、18 和 19。

4. 转至 "Threads"（线程）视图并选择线程 2 以及您的实验中在线程 2 显示对 `Routine.*` 函数的调用的时间段内显示活动的线程。

   您可能发现更容易通过单击 "Name"（名称）列标题来首先按名称排序。然后在单击线程时按 Ctrl 键来选择多个线程。

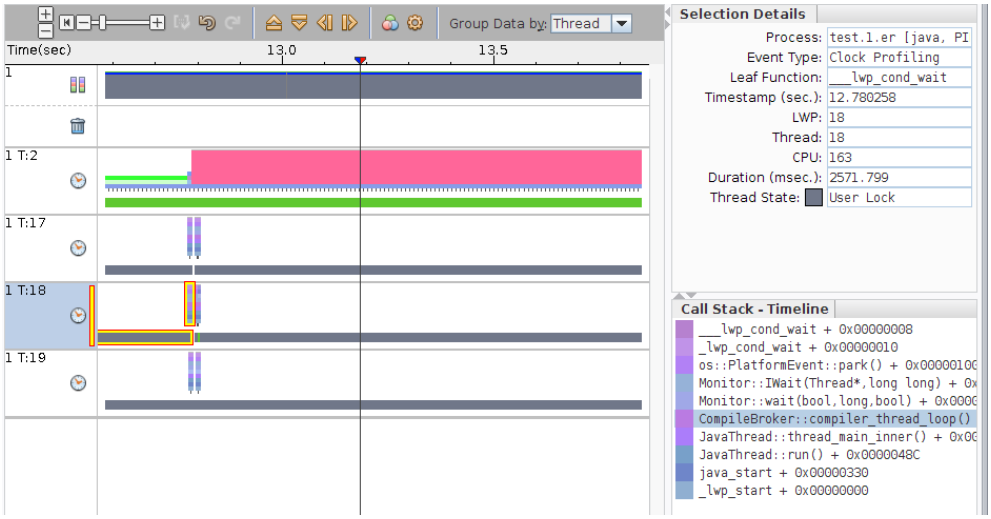
   在此示例中，选择线程 2、17、18、19。

Total CPU Time VALUES sec.	Name
81.977	*<Total>*
0.030	Process 1, Thread 1
80.957	Process 1, Thread 2, JThread 3 'main', Group 'main', Parent 'system'
0.080	Process 1, Thread 3
0.070	Process 1, Thread 4
0.070	Process 1, Thread 5
0.080	Process 1, Thread 6
0.070	Process 1, Thread 7
0.080	Process 1, Thread 8
0.100	Process 1, Thread 9
0.080	Process 1, Thread 10
0.080	Process 1, Thread 11
0.070	Process 1, Thread 12
0.080	Process 1, Thread 13
0.	Process 1, Thread 14, JThread 0 '', Group '', Parent ''
0.010	Process 1, Thread 15, JThread 1 '', Group '', Parent ''
0.	Process 1, Thread 16, JThread 2 'Signal Dispatcher', Group 'system', Parent ''
0.050	Process 1, Thread 17
0.040	Process 1, Thread 18
0.	Process 1, Thread 19
0.030	Process 1, Thread 20

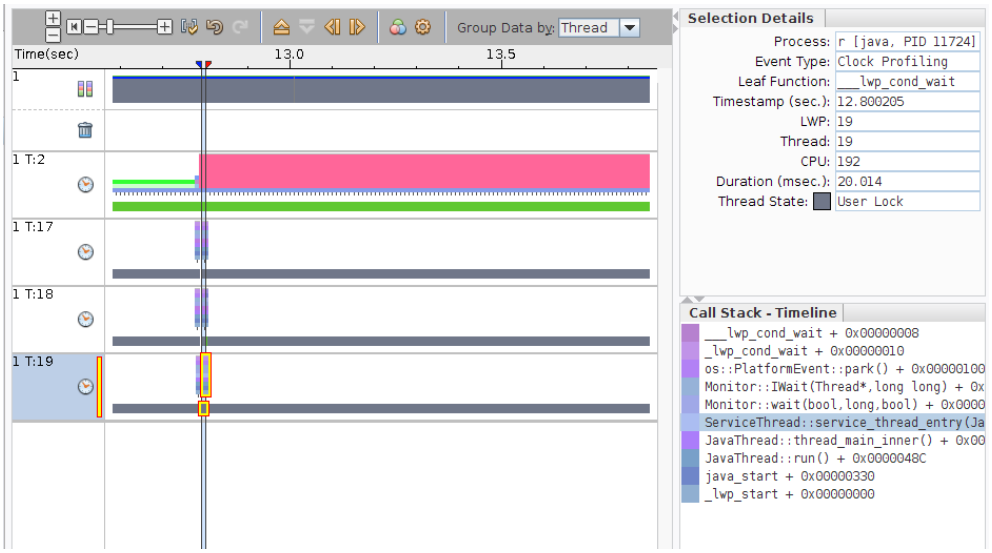
5. 单击工具栏中的按钮  ，然后选择 "Add Filter: Include only events with selected items"（添加过滤器：仅包括含有所选项的事件）。

   这将设置过滤器以仅包括那些线程上的事件。还可以在 "Threads"（线程）视图中右键单击，然后选择过滤器。
6. 返回 "Timeline"（时间线）视图并重置水平缩放以易于看到图案。
7. 单击线程 17 和 18 中的事件。

   请注意，"Call Stack"（调用堆栈）面板显示 `CompileBroker::compiler_thread_loop()`。那些线程是用于 HotSpot 编译器的线程。



"Thread 19"（线程 19）显示调用堆栈并且其中具有 `ServiceThread::service_thread_entry()`。



这些线程上发生多个事件的原因是只要用户代码调用新的方法并且在其中花费了相当时间，就会触发 HotSpot 编译器来生成该方法的计算机代码。HotSpot 编译器足

够快，从而运行该编译器的线程不会消耗非常多的 "User CPU Time"（用户 CPU 时间）。

如何触发 HotSpot 编译器的详细信息不在本教程范围内。

多线程程序上的硬件计数器分析

本章包含以下主题。

关于硬件计数器分析教程

本教程介绍了如何在名为 mttest 的多线程程序上使用性能分析器来收集和了解时钟分析和硬件计数器分析数据。

您将了解 "Overview"（概述）页面和更改显示哪些度量，检查 "Functions"（函数）视图、"Callers-Callees"（调用方-被调用方）视图、"Source"（源）和 "Disassembly"（反汇编）视图，以及应用过滤器。

您将首先了解时钟分析数据，然后了解 HW 计数器分析数据以及 "Instructions Executed"（执行的指令）（在所有受支持系统上可用的计数器）。接下来您将了解 "Instructions Executed"（执行的指令）和 "CPU Cycles"（CPU 周期）（在大多数而不是所有受支持的系统上可用）以及 "D-cache Misses"（数据高速缓存未命中次数）（在一些受支持的系统上可用）。

如果在具有 "D-cache Misses"（数据高速缓存未命中次数） (dcm) 的精确硬件计数器的系统上运行，则还将学习如何使用 IndexObject 和 MemoryObject 视图，以及如何检测高速缓存行的伪共享。

程序 mttest 是一个简单程序，对伪数据应用各种同步选项。该程序实现多项不同任务，每个任务都使用一种基本算法：

- 将多个工作块排队，缺省情况下为四个。每个工作块都是结构 Workblk 的一个实例。

- 衍生多个线程以处理工作，缺省情况下也是四个。向每个线程传递其专用工作块。
- 在每个任务中，使用特定的同步基元控制对工作块的访问。
- 在同步之后，处理块的工作。

在记录的实验中看到的数据将与此处显示的数据不同。用于本教程中屏幕抓图的实验是在运行 Oracle Solaris 11.3 的 SPARC T5 系统上记录的。来自运行 Oracle Solaris 或 Linux 的 x86 系统的数据将会有所不同。此外，数据收集本质上是统计性的，随实验的不同而不同，即使运行在同一系统和 OS 上也是如此。

您看到的性能分析器窗口配置可能不会与屏幕抓图完全匹配。通过性能分析器，可以拖动窗口各部分之间的分隔条，折叠各部分以及调整窗口大小。性能分析器记录其配置，并在下次运行时使用相同的配置。在捕获教程所示的屏幕抓图的过程中进行了许多配置更改。

设置 mttest 样例代码

开始之前：

查看以下内容以了解有关获取代码和设置您的环境的信息。

- “获取教程的样例代码” [10]
- “为教程设置环境” [10]

您可能需要先浏览 C 分析简介中的入门教程以熟悉性能分析器。

1. 使用以下命令将 mttest 目录的内容复制到您自己的专用工作区：

   ```
   % cp -r OracleDeveloperStudio12.5-Samples/PerformanceAnalyzer/mttest directory
   ```

   将 *directory* 替换为正在使用的工作目录。
2. 转到该工作目录副本。

   ```
   % cd directory/mttest
   ```
3. 生成目标可执行文件。

   ```
   % make clobber

   % make
   ```

   注 - 仅当之前在目录中运行了 make 时才需要使用 clobber 子命令，但该子命令在任何情况下都可以放心地使用。

   在运行 make 之后，目录将包含要在教程中使用的目标应用程序，即一个名为 mttest 的 C 程序。

提示 - 如果您愿意，可以编辑 Makefile 以执行以下操作：使用 GNU 编译器而不是缺省的 Oracle Developer Studio 编译器；以 32 位而不是缺省的 64 位编译；以及添加不同的编译器标志。

从硬件计数器分析教程的 mttest 收集数据

收集数据的最简单方式是在 mttest 目录中运行以下命令：

```
% make hwcperf
```

Makefile 的 hwcperf 目标将启动 collect 命令并记录实验。

注 - 相比前一个入门教程，此教程可能要花很长时间编译和收集数据。

缺省情况下实验将被命名为 test.1.er，它包含以下三个计数器的时钟分析数据和硬件计数器分析数据：inst（指令）、cycles（周期）和 dcm（数据高速缓存未命中次数）。

如果系统不支持 cycles 计数器或 dcm 计数器，则 collect 命令将失败。在这种情况下，请编辑 Makefile 将 # 号移动到相应的行，以启用仅指定您系统支持的那些计数器的 HWC_OPT 变量。实验将不具有已忽略的那些计数器中的数据。

提示 - 可以使用命令 collect -h 确定您的系统支持哪些计数器。有关硬件计数器的信息，请参见《Oracle Developer Studio 12.5：性能分析器》 中的 “硬件计数器列表”。

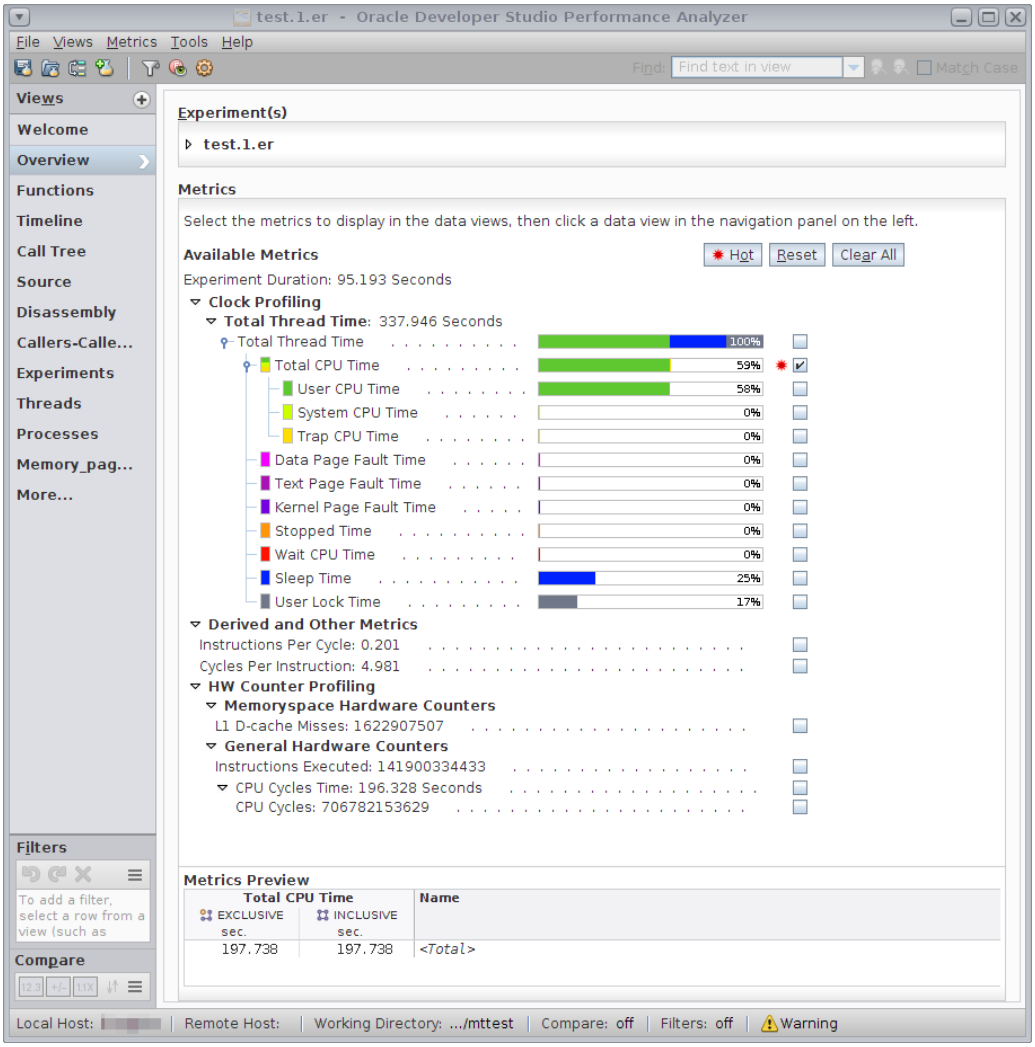
检查 mttest 的硬件计数器分析实验

本节介绍如何了解通过上一节中 mttest 样例代码创建的实验中的数据。

从 mttest 目录启动性能分析器并装入实验，如下所示：

```
% analyzer test.1.er
```

实验打开时，性能分析器将显示 "Overview"（概述）页面。



首先显示时钟分析度量并包括带颜色的条。大多数线程时间花费在 "User CPU Time"（用户 CPU 时间）中。一些时间花费在 "Sleep Time'（休眠时间）或 "User Lock Time"（用户锁定时间）中。

如果同时记录了 `cycles` 和 `insts` 计数器，则将显示 "Derived and Other Metrics"（派生的度量和其他度量）组。派生的度量表示这两个计数器中的度量比率。高的 "Instructions Per Cycle"（每个周期的指令数）值或低的 "Cycles Per Instruction"（每个指令的周期数）值表示效率相对较高的代码。相反，低的 "Instructions Per Cycle"（每个周期的指令数）值或高的 "Cycles Per Instruction"（每个指令的周期数）值则表示效率相对较低的代码。

在此实验中，"HW Counter Profiling"（HW 计数器分析）组显示了两个子组，即 "Memoryspace Hardware Counters"（内存空间硬件计数器）和 "General Hardware Counters"（常规硬件计数器）。"Instructions Executed"（执行的指令）计数器 (`insts`) 在 "General Hardware Counters"（常规硬件计数器）下列出。如果您收集的数据包括 `cycles` 计数器，则在 "General Hardware Counters"（常规硬件计数器）下还将列出 "CPU Cycles"（CPU 周期）。如果数据是在具有 *precise* `dcm` 计数器的计算机上收集的，则 "L1 D-cache Misses"（L1 数据高速缓存未命中次数）将在 "Memoryspace Hardware Counters"（内存空间硬件计数器）下列出。如果 `dcm` 计数器可用但不是精确计数器，则 "L1 D-cache Misses"（L1 数据高速缓存未命中次数）将在 "General Hardware Counters"（常规硬件计数器）下列出。精确计数器是在执行导致溢出的指令时产生溢出中断的计数器。幅度可变的“失控”越过导致溢出的指令时提供非精确计数器。即使非精确计数器与内存相关，它也不能用于内存空间分析。有关内存空间分析的更多信息，请参见《Oracle Developer Studio 12.5：性能分析器》 中的 “数据空间分析和内存空间分析”。

如果您的系统不支持 `dcm`，且您编辑了 `Makefile` 以删除 `-h dcm`，则会看到 "Instructions Executed"（执行的指令）和 "CPU Cycles"（CPU 周期）计数器。如果编辑了 `Makefile` 以同时删除 `-h dcm` 和 `-h cycles`，则只能看到 "Instructions Executed"（执行的指令）计数器。

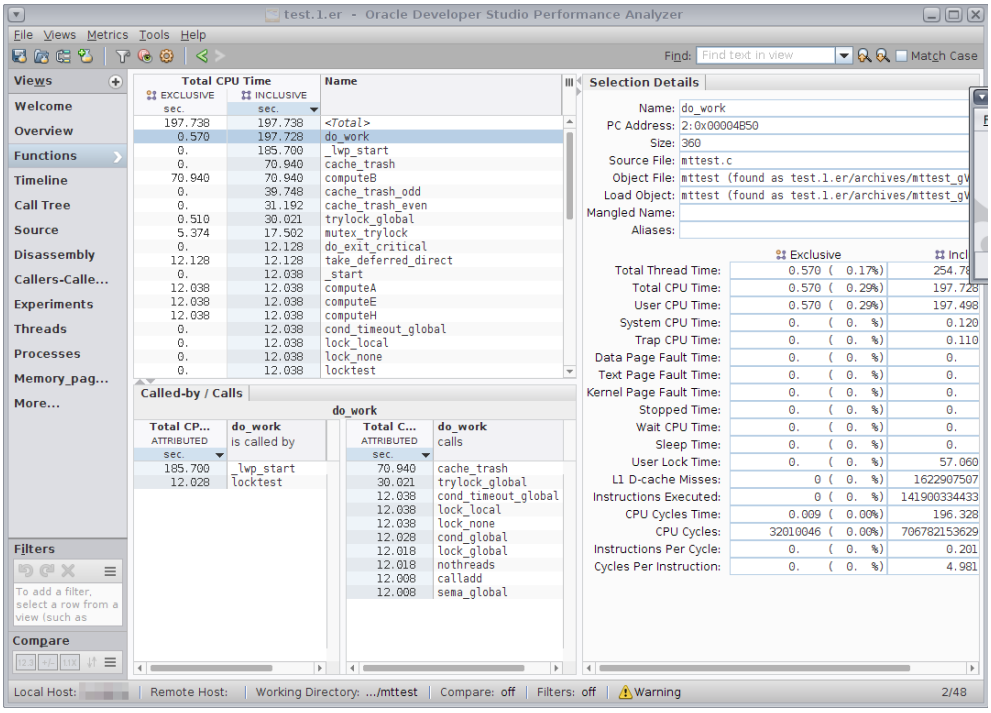
您将在本教程接下来的几节中了解这些度量及其解释。

了解时钟分析数据

本节使用 "Overview"（概述）页面和带 "Called-by/Calls"（调用方/调用）面板的 "Functions"（函数）视图来说明时钟分析数据。

1. 在 "Overview"（概述）页面中，取消选中三个 HW 计数器度量的复选框，仅使 "Total CPU Time"（CPU 总时间）复选框保持选中状态。
2. 转至 "Functions"（函数）视图，单击 "Inclusive Total CPU Time"（包含总 CPU 时间）列标题一次，以便根据“包含总 CPU 时间”进行排序。

   函数 `do_work（）` 现在应该处于列表的顶部。



3. 选择 `do_work()` 函数并查看 "Functions"（函数）视图底部的 "Called-by/Calls"（调用方/调用）面板。

   请注意 `do_work()` 是从两个位置调用的，并且它调用十个函数。

   `do_work()` 调用的十个函数表示十个不同的任务，每个任务都具有程序执行的不同同步方法。在从 `mttest` 创建的一些实验中，您可能会看到第十一个函数，该函数使用相对较少的时间获取其他任务的工作块。屏幕抓图中未显示此函数。

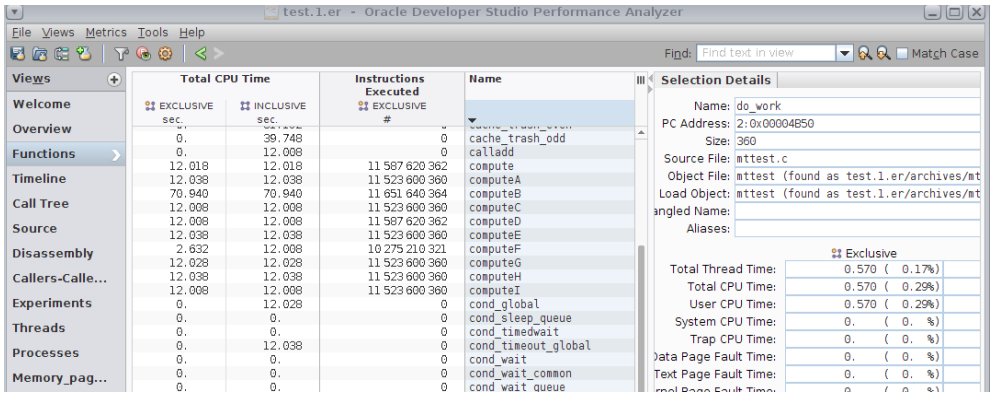
   通常情况下，创建处理数据的线程时会调用 `do_work()`，并在从 `_lwp_start()` 调用时显示它。在一种情况下，从 `locktest()` 被调用后，`do_work()` 调用名为 `nothreads()` 的单线程任务。

   在面板的 "Calls"（调用）侧，请注意，除了前两个被调用方外，所有被调用方都显示大约相同的 "Attributed Total CPU"（归属总 CPU）时间量（约为 12 秒）。

了解硬件计数器指令分析度量

本节介绍如何使用常规硬件计数器查看对函数执行了多少指令。

1. 选择 "Overview"（概述）页面，然后启用 "General Hardware Counters"（常规硬件计数器）下名为 "Instructions Executed"（执行的指令）的 "HW Counter Profiling"（HW 计数器分析）度量。
2. 返回到 "Functions"（函数）视图，然后单击 "Name"（名称）列标题以按字母顺序排序。
3. 向下滚动以查找函数 `compute`（）、`computeA`（）、`computeB`（）等。



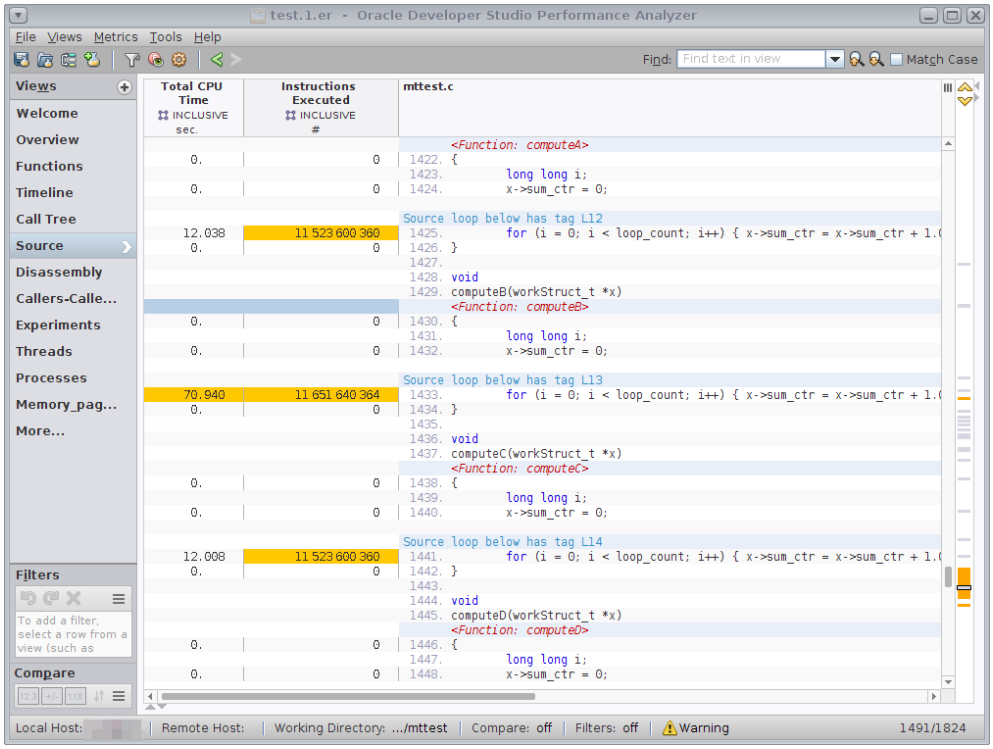
请注意，除 `computeB`（） 和 `computeF`（） 之外的所有函数都具有大致相同的 "Exclusive Total CPU time"（独占总 CPU 时间）和 "Exclusive Instructions Executed"（独占 执行的指令）值。

4. 选择 `computeF`（） 并切换到 "Source"（源）视图。通过双击 `computeF`（），可以一步执行此操作。

computeF（） 中的计算内核是不同的，因为它调用函数 addone（） 来加一，而其他 compute*（） 函数则直接执行加法。这就解释了为什么其性能与其他函数的不同。

5. 在 "Source"（源）视图中上下滚动以查看所有的 compute*（） 函数。

   请注意，所有的 compute*（） 函数（包括 computeB（））都显示数量大致相等的已执行指令。而 computeB（） 显示截然不同的 CPU 时间成本。



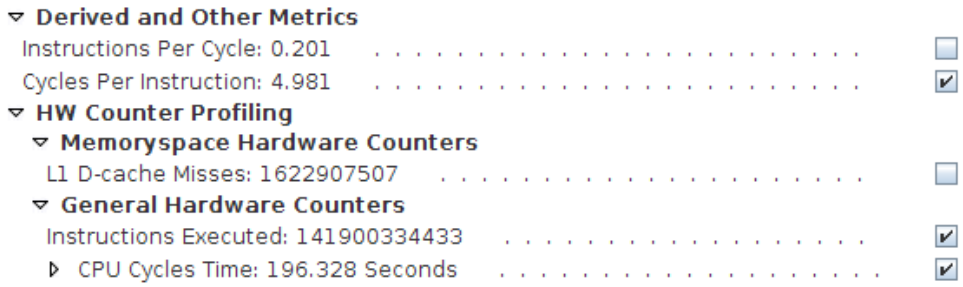
下一节帮助说明为什么 `computeB（）` 的总 CPU 时间要长得多。

了解硬件计数器 CPU 周期分析度量

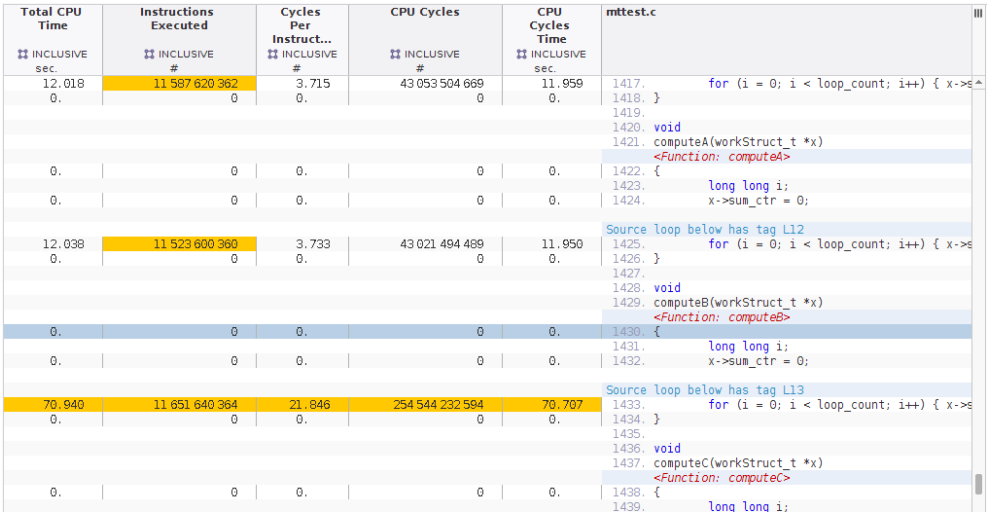
教程的此部分需要一个包含来自 `cycles` 计数器的数据的实验。如果您的系统不支持此计数器，则在本节中无法使用您的实验。请跳到下一节“了解高速缓存争用和高速缓存分析度量” [77]。

1. 选择 "Overview"（概述）页面，然后启用派生的度量 "Cycles Per Instruction"（每个指令的周期数）和 "General Hardware Counter"（常规硬件计数器）度量、"CPU Cycles Time"（CPU 周期时间）。

   应该保持选中 "Total CPU Time"（CPU 总时间）和 "Instructions Executed"（执行的指令）。



2. 返回到 `computeB（）` 的 "Source"（源）视图。



请注意 "Incl. CPU Cycles"（包含 CPU 周期）时间和 "Incl. Total CPU Time"（包含总 CPU 时间）在每个 `compute*（）` 函数中大致相等。.这表示时钟分析和 CPU 周期硬件计数器分析正在获取类似的数据。

在屏幕抓图中，"Incl. CPU Cycles"（包含 CPU 周期）和 "Incl. Total CPU Time"（包含总 CPU 时间）对于每个 `compute*（）` 函数（`computeB（）` 除外）都约为 12 秒。在实验中您还应该看到，"Incl. Cycles Per Instruction (CPI)"（包含 每个指令的周期数 (CPI)）对于 `computeB（）` 要比对于其他 `compute*（）` 函数高得多。这表示需要更多的 CPU 周期来执行相同数量的指令，因此 `computeB（）` 比其他函数的效率低。

目前为止您看到的数据显示了 `computeB（）` 函数与其他函数之间的差异，但是未显示它们可能不同的原因。本教程的下一部分将说明为什么 `computeB（）` 是不同的。

了解高速缓存争用和高速缓存分析度量

本节和教程的其余部分需要一个包含来自精确 dcm 硬件计数器的数据的实验。如果您的系统不支持精确 dcm 计数器，则本教程的剩余部分不适用于在系统上记录的实验。

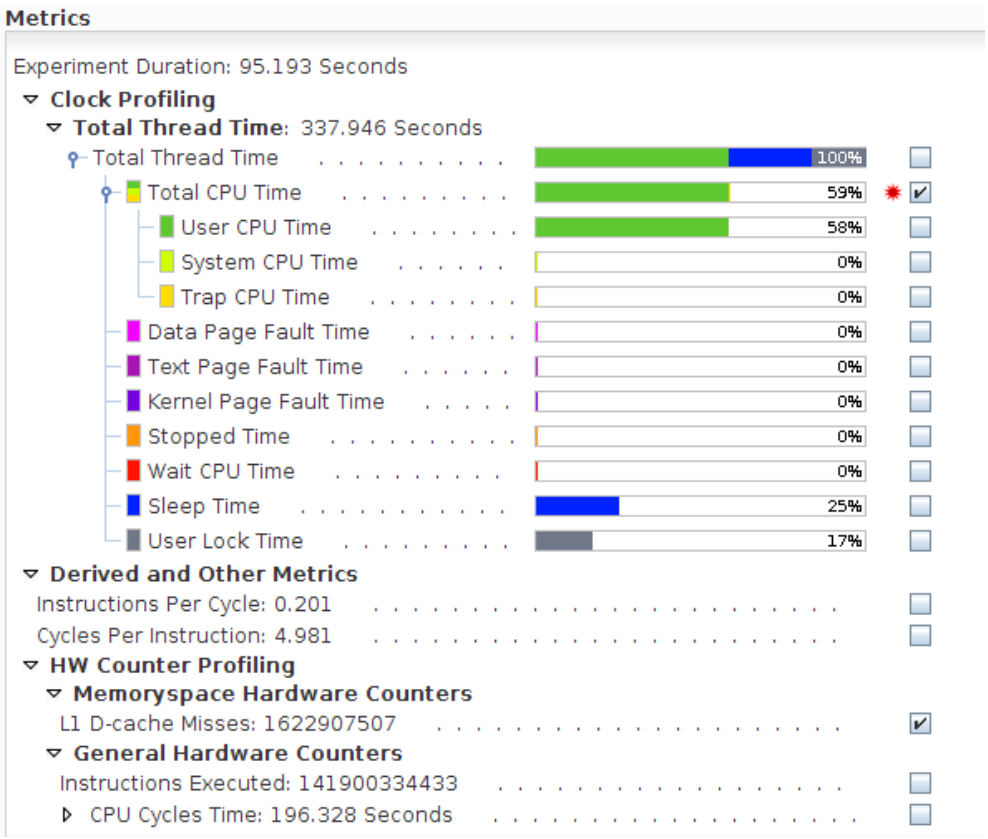
dcm 计数器对高速缓存未命中次数进行计数，高速缓存未命中次数是引用不在高速缓存中的内存地址的装入和存储次数。

地址可能由于以下任一原因而不在高速缓存中：

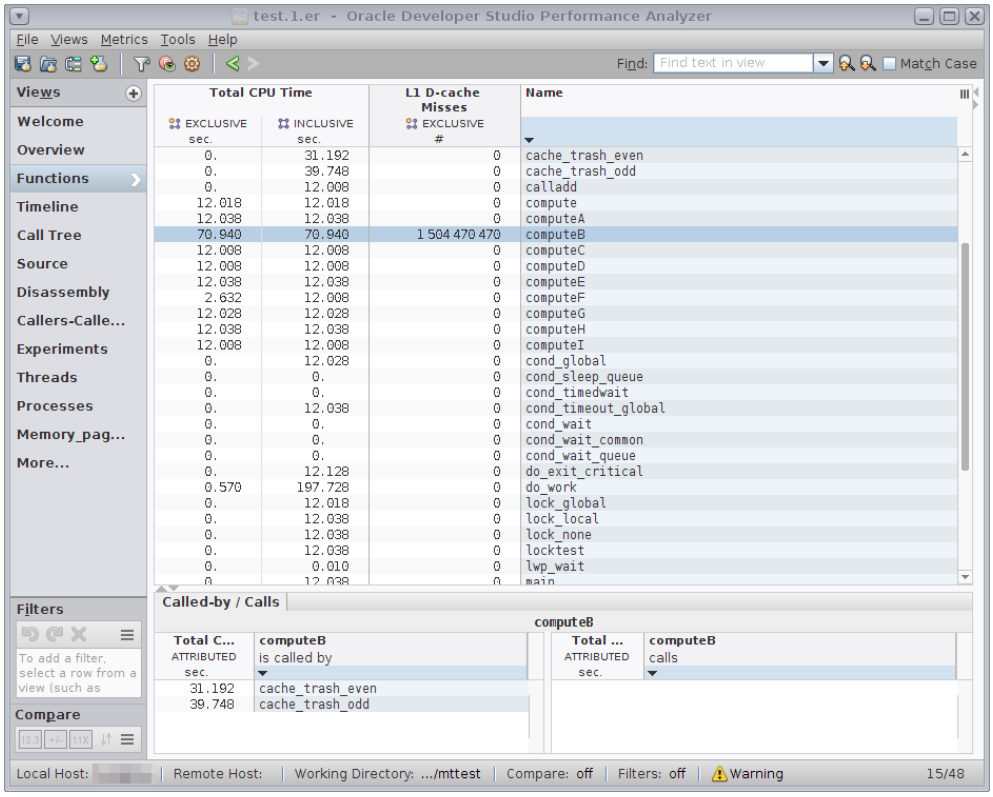
- 因为当前指令是对该 CPU 中该内存位置的首次引用。更准确地说，它是对共享高速缓存行的任何内存地址的首次引用。
- 因为线程已引用如此多的其他内存地址，导致当前地址已从高速缓存刷新。这是容量未命中的情况。
- 因为线程已引用映射到同一高速缓存行的其他内存地址，导致当前地址被刷新。这是冲突未命中的情况。
- 因为其他线程已写入高速缓存行中的地址，导致当前线程的高速缓存行被刷新。这是共享未命中的情况，并可能是以下两种共享未命中之一：
  - 真共享，其中其他线程已写入当前线程所引用的同一地址。真共享导致的高速缓存未命中是无法避免的。
  - 伪共享，其中其他线程已写入与当前线程所引用的地址不同的地址。高速缓存未命中由于伪共享而发生，因为高速缓存硬件在高速缓存行粒度上运作，而不是在数据字粒度上运作。通过更改相关数据结构，以便在每个线程中引用的不同地址处于不同的高速缓存行上，可以避免伪共享。

以下过程检查对函数 computeB（） 具有影响的伪共享情况。

1. 返回到 "Overview"（概述），然后启用 "L1 D-cache Misses"（L1 数据高速缓存未命中次数）的度量，并禁用 "Cycles Per Instruction"（每个指令的周期数）的度量。



2. 切换回 "Functions"（函数）视图并查看 `compute*（）` 例程。

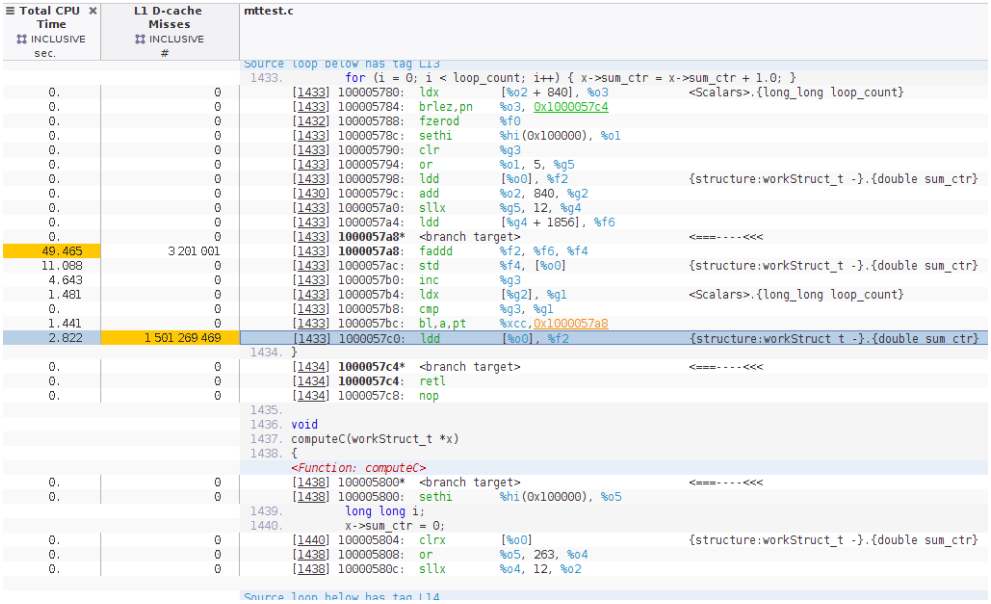


回想一下，所有的 `compute*`（） 函数都显示了大致相同的指令计数，而 `computeB`（） 显示了更高的 "Total CPU Time"（CPU 总时间），并且是具有 "Exclusive L1 D-cache Misses"（独占 L1 数据高速缓存未命中次数）的重要计数的唯一函数。

3. 返回到 "Source"（源）视图，请注意在 `computeB`（） 中高速缓存未命中次数处于单行循环中。
4. 如果尚未在导航面板中看到 "Disassembly"（反汇编）标签，要添加该视图，请单击导航面板顶部 "Views"（视图）标签旁边的 + 按钮，然后选中 "Disassembly"（反汇编）复选框。

   滚动 "Disassembly"（反汇编）视图，直到您看到 "L1 D-Cache Misses"（L1 数据高速缓存未命中次数）很高的装入指令对应的行。

提示 - 视图（如 "Disassembly"（反汇编））的右边缘包括快捷方式，可以单击它们以跳转到具有高度量的行或热线。尝试单击边缘顶部的 "Next Hot Line"（下一个热线）向下箭头或 "Non-Zero Metrics"（非零度量）标记，以快速跳转到具有值得注意的度量值的行。



在 SPARC 系统上，如果使用 `-xhwcprof` 进行编译，则使用表明指令正在引用 `workStruct_t` 数据结构中的双字 `sum_ctr` 的结构信息对装入和存储进行注释。您还会看到其地址与下一行相同的行，它们将 `<branch target>` 作为其指令。这样的行表示下一个地址是分支的目标，这意味着代码可能已到达指示为热点的指令，而不曾执行 `<branch target>` 之上的指令。

在 x86 系统上，不对装入和存储进行注释，也不显示 `<branch target>` 行，因为 x86 不支持 `-xhwcprof`。

5. 在 "Functions"（函数）和 "Disassembly"（反汇编）视图之间来回切换，选择各种 `compute*`（） 函数。

   请注意，对于所有的 `compute*`（） 函数，"Instructions Executed"（执行的指令）计数很高的指令将引用相同的结构字段。

现在您已看到，`computeB`（） 所用的时间比其他函数长得多，即使它执行相同数量的指令也是如此，而且它是获取高速缓存未命中次数的唯一函数。高速缓存未命中次数是造

成执行指令的周期数增加的原因，因为与具有高速缓存命中的装入相比，完成具有高速缓存未命中的装入所用的周期要多得多。

对于除 computeB（） 之外的所有 compute*（） 函数，结构 workStruct_t 中每个线程的参数所指向的双字字段 sum_ctr 包含在该线程的 Workblk 内。虽然 Workblk 结构是连续分配的，但是它们大到足以使每个结构中的双字距离太远而无法共享高速缓存行。

对于 computeB（），线程中的 workStruct_t 参数是该结构的连续实例，它的长度仅为一个双字 long。因此，由不同线程使用的双字将共享高速缓存行，这会导致一个线程中的任何存储使其他线程中的高速缓存行无效。这就是高速缓存未命中计数如此高的原因所在，重新填充高速缓存行的延迟是总 CPU 时间和 CPU 周期度量如此高的原因所在。

在此示例中，线程存储的数据字未重叠，尽管它们共享高速缓存行。此性能问题称为“伪共享”。如果线程引用相同的数据字，则将是真共享。到目前为止查看的数据并不区分伪共享和真共享。

在本教程的最后一节中，将说明伪共享和真共享之间的差异。

检测伪共享

教程的此部分仅适用于 L1 数据高速缓存未命中 dcm 计数器精确的系统。这样的系统包括 SPARC-T4、SPARC-T5、SPARC-M5 和 SPARC-M6 等。如果您的实验是在没有精确 dcm 计数器的系统上记录的，则本节不适用。

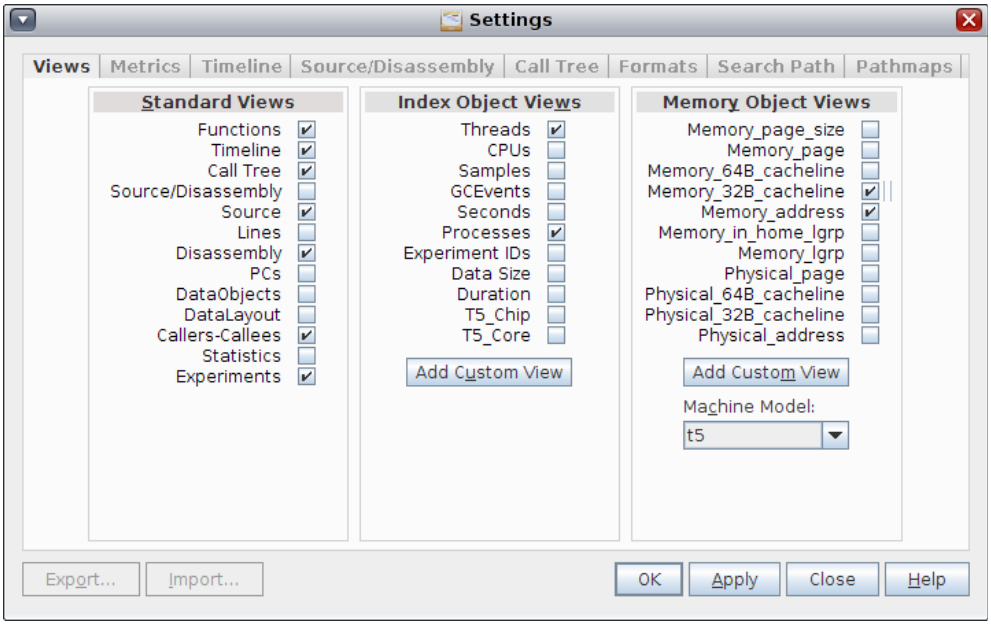
以下过程说明如何使用索引对象视图和内存对象视图以及过滤功能。

在具有精确的内存相关计数器的系统上创建实验时，会在实验中记录计算机模型。计算机模型表示地址到该计算机的内存子系统中各种组件的映射。在性能分析器或 er_print 中装入实验时，会自动装入计算机模型。

用于本教程中屏幕抓图的实验记录在 SPARC T5 系统上，该计算机的 t5 计算机模型随实验自动装入。计算机模型将添加索引对象和内存对象的数据视图。

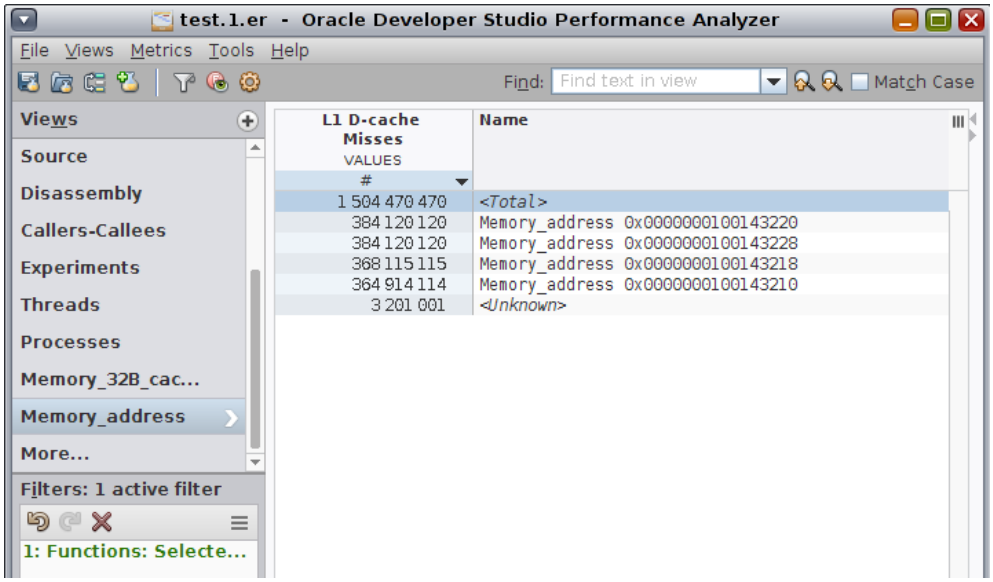
1. 转到 "Functions"（函数）视图并选择 computeB（），然后右键单击并选择 "Add Filter: Include only stacks containing the selected functions"（添加过滤器：仅包括含有所选函数的堆栈）。

   通过过滤，可以将注意力集中于 computeB（） 函数的性能和在该函数中发生的分析事件。
2. 单击工具栏中的 "Settings"（设置）按钮，或者选择 "Tools"（工具）-> "Settings"（设置）打开 "Settings"（设置）对话框，然后在该对话框中选择 "Views"（视图）标签。



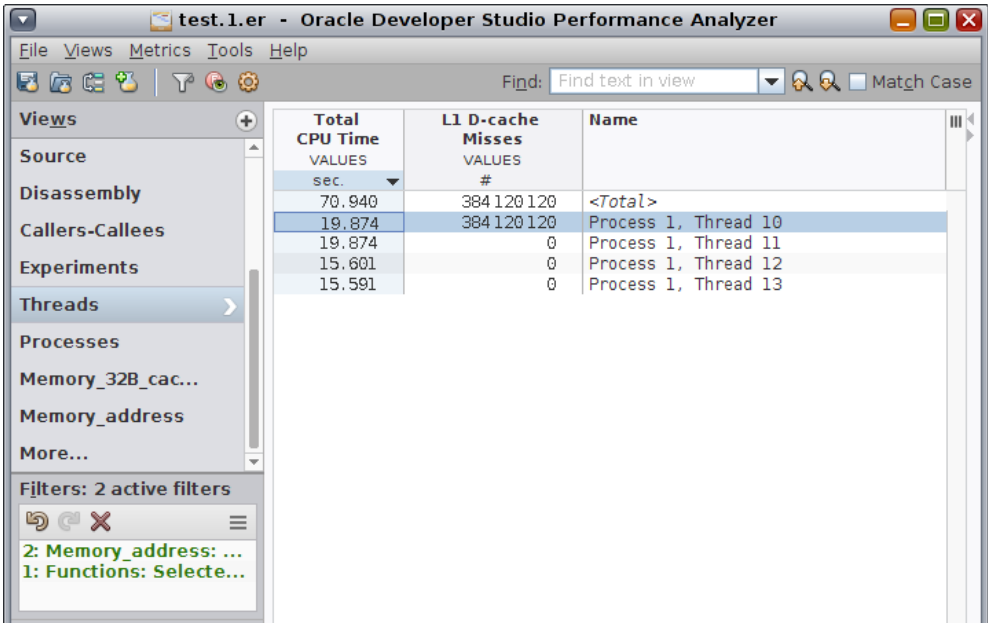
右侧的面板标记有 "Memory Objects Views"（内存对象视图），并显示一个表示 SPARC T5 计算机的内存子系统结构的数据视图列表。

3. 选中 "Memory_address" 和 "Memory_32B_cacheline" 的复选框，然后单击 "OK"（确定）。
4. 在 "Views"（视图）导航面板中选择 "Memory_address" 视图。



在此实验中，可以看到存在四个获取高速缓存未命中次数的不同地址。

5. 选择其中一个地址，然后右键单击并选择 "Add Filter: Include only events with the selected item"（添加过滤器：仅包括含有所选项的事件）。
6. 选择 "Threads"（线程）视图。



正如在之前的屏幕抓图中所见，只有一个线程具有该地址的高速缓存未命中次数。

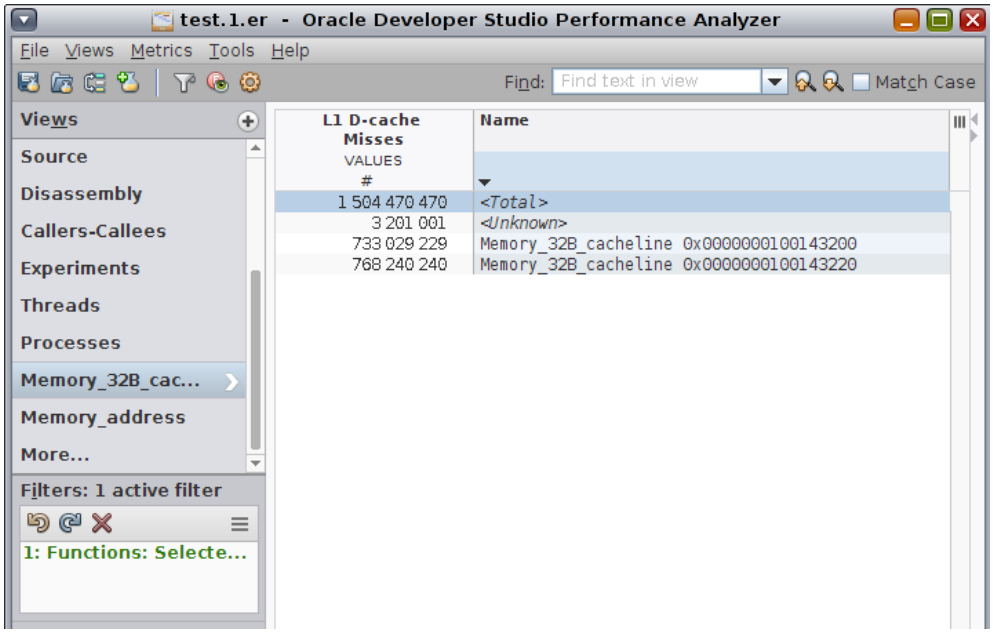
7. 通过在视图中右键单击并从上下文菜单中选择 "Undo Filter Action"（撤消过滤器操作），删除地址过滤器。

   也可以使用 "Active Filters"（活动过滤器）面板中的 "Undo Filter Action"（撤消过滤器操作）删除过滤器。

8. 返回到 "Memory_address" 视图，选择并过滤其他地址，然后检查 "Threads"（线程）视图中的关联线程。

   通过过滤和取消过滤，以及通过此方式在 "Memory_address" 和 "Threads"（线程）视图之间切换，可以确认在四个线程和四个地址之间存在一对一关系。即，四个线程不共享地址。

9. 在 "Views"（视图）导航面板中选择 "Memory_32B_cacheline" 视图。



在 "Active Filters"（活动过滤器）面板中，确认只有该过滤器在函数 `computeB`（）上处于活动状态。该过滤器将显示为 "Functions: Selected Functions"（函数：所选函数）。现在地址上应该没有过滤器处于活动状态。

您应该看到，有两个 32 字节高速缓存行获取四个线程及其四个相应地址的高速缓存未命中次数。这确认了以下情况：虽然之前您看到四个线程不共享地址，但是在此处看到它们确实共享高速缓存行。

伪共享是一个诊断起来非常困难的问题，SPARC T5 芯片以及 Oracle Developer Studio 性能分析器使得您可以进行诊断。

多线程程序上的同步跟踪

本教程包括以下主题。

关于同步跟踪教程

本教程介绍如何在多线程程序上使用性能分析器来检查时钟分析和同步跟踪数据。

使用 "Overview"（概述）页面快速查看突出显示的性能度量并更改数据视图中显示的度量。使用 "Function"（函数）、"Callers-Callees"（调用方-被调用方）和 "Source"（源）视图了解数据。本教程还介绍如何比较两个实验。

本教程可帮助您了解同步跟踪数据并说明如何将其与时钟分析数据相关联。

在记录的实验中看到的数据将与此处显示的数据不同。用于本教程中屏幕抓图的实验是在运行 Oracle Solaris 11.3 的 SPARC T5 系统上记录的。来自运行 Oracle Solaris 或 Linux 的 x86 系统的数据将会有所不同。此外，数据收集本质上是统计性的，随实验的不同而不同，即使运行在同一系统和 OS 上也是如此。

您看到的性能分析器窗口配置可能不会与屏幕抓图完全匹配。通过性能分析器，可以拖动窗口各部分之间的分隔条，折叠各部分以及调整窗口大小。性能分析器记录其配置，并在下次运行时使用相同的配置。在捕获教程所示的屏幕抓图的过程中进行了许多配置更改。

关于 `mttest` 程序

程序 `mttest` 是一个简单程序，对伪数据应用各种同步选项。该程序实现多项不同任务，每个任务都使用相同的基本算法：

- 将多个工作块排队（缺省情况下为 4 个）。
- 衍生多个线程以处理它们（缺省情况下也是 4 个）。

- 在每个任务中，使用特定的同步基元控制对工作块的访问。
- 在同步之后，处理块的工作。

每个任务使用不同的同步方法。mttest 代码按顺序执行每个任务。

关于同步跟踪

通过插入用于同步的各种库函数（例如 mutex_lock（）、pthread_mutex_lock（）、sem_wait（） 等）来实现同步跟踪。将跟踪 pthread 和 Oracle Solaris 同步调用。

目标程序调用这些函数中的一个函数时，数据收集器将拦截该调用。将捕获当前时间、锁的地址以及一些其他数据，然后插入例程调用实际库例程。实际库例程返回时，数据收集器再次读取时间并计算结束时间与开始时间之间的差值。如果该差值超过用户指定的阈值，将记录该事件。如果时间未超过阈值，则不记录该事件。在以上任一情况下，实际库例程的返回值都将返回到调用方。

通过使用 collect 命令的 -s 选项，可以设置用于确定是否记录事件的阈值。如果使用性能分析器收集实验，可以在 "Profile Application"（分析应用程序）对话框中将阈值指定为 "Synchronization Wait Tracing"（同步等待跟踪）的 "Minimum Delay"（最小延迟）。可以将阈值设置为微秒数或设置为关键字 calibrate 或 on。使用 calibrate 或 on 时，数据收集器将确定其获取无争用互斥锁所花费的时间并将阈值设置为该时间值的五倍。阈值指定为 0 或 all 将导致记录所有事件。

在本教程中，将记录两个实验中的同步等待跟踪，一个实验具有校准的阈值，一个实验具有零阈值。这两个实验还包括时钟分析。

设置 mttest 样例代码

开始之前：

查看以下内容以了解有关获取代码和设置您的环境的信息。

- “获取教程的样例代码” [10]
- “为教程设置环境” [10]

您可能需要先浏览 C 分析简介中的入门教程以熟悉性能分析器。

本教程使用与教程多线程程序上的硬件计数器分析相同的 mttest 代码。您应该为本教程生成单独副本。

1. 使用以下命令将 mttest 目录的内容复制到您自己的专用工作区：

```
% cp -r OracleDeveloperStudio12.5-Samples/PerformanceAnalyzer/mttest directory
```

将 *directory* 替换为正在使用的工作目录。

2. 转到该工作目录副本。

```
% cd directory/mttest
```

3. 生成目标可执行文件。

```
% make clobber

% make
```

注 - 仅当之前在目录中运行了 make 时才需要使用 clobber 子命令，但该子命令在任何情况下都可以放心地使用。

在运行 make 之后，目录将包含要在教程中使用的目标应用程序，即一个名为 mttest 的 C 程序。

提示 - 如果您愿意，可以编辑 Makefile 以执行以下操作：使用 GNU 编译器而不是缺省的 Oracle Developer Studio 编译器；以 32 位而不是缺省的 64 位编译；以及添加不同的编译器标志。

收集同步跟踪教程的 mttest 数据

收集数据的最简单方式是在 mttest 目录中运行以下命令：

```
% make syncperf
```

Makefile 的 syncperf 目标启动 collect 命令两次并创建两个实验。

注 - 相比前一个入门教程，此教程可能要花很长时间编译和收集数据。

这两个实验名为 test.1.er 和 test.2.er，每个实验都包含同步跟踪数据和时钟分析数据。对于第一个实验，collect 通过指定 -s on 选项使用校准的阈值记录事件。对于第二个实验，collect 通过指定 -s all 选项将阈值设置为零来记录所有事件。在两个实验中，都通过 -p on 选项启用时钟分析。

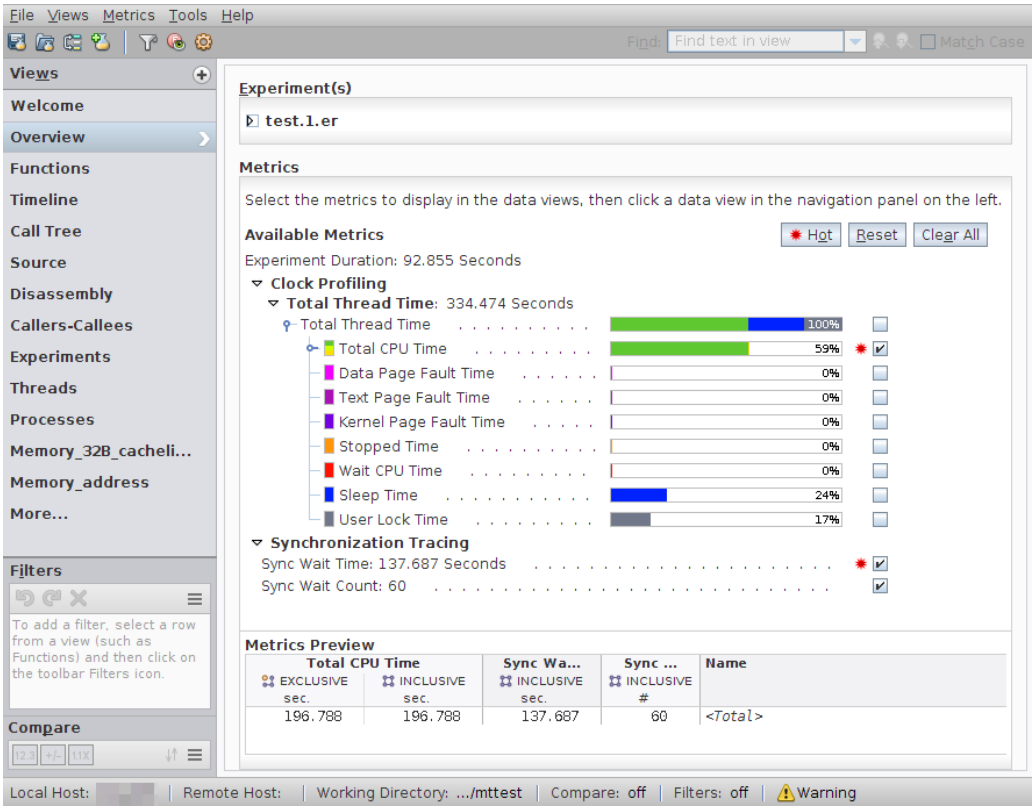
检查 mttest 的同步跟踪实验

本节介绍如何了解通过上一节中 mttest 样例代码创建的实验中的数据。

从 mttest 目录启动性能分析器并按如下所示装入第一个实验：

```
% analyzer test.1.er
```

实验打开时，性能分析器将显示 "Overview"（概述）页面。

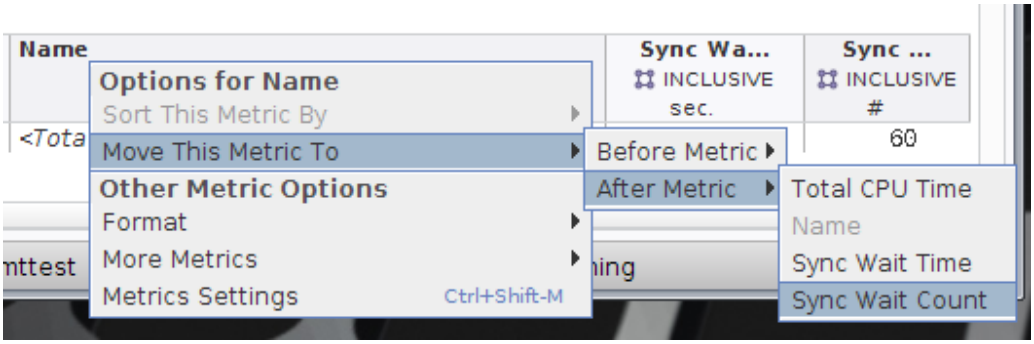


"Clock Profiling"（时钟分析）度量第一个显示并包含带颜色的条。大多数线程时间花费在 "User CPU Time"（用户 CPU 时间）中。一些时间花费在 "Sleep Time'（休眠时间）或 "User Lock Time"（用户锁定时间）中。

"Synchronization Tracing"（同步跟踪）度量显示在另一组中，其中包括两个度量："Sync Wait Time"（同步等待时间）和 "Sync Wait Count"（同步等待计数）。

注 - 如果没有看到 "Sync Wait Time"（同步等待时间）和 "Sync Wait Count"（同步等待计数）度量，您可能需要滚动至右侧查看相关列。您可以将任意列移动至更便于查看的位置，具体方法是右键单击度量列标题，选择 "Move This Metric（将此度量移至），然后相对其他度量选择一个更便于查看的位置。

以下示例将 "Name"（名称）列移动至 "Sync Wait Count"（同步等待计数）度量之后。

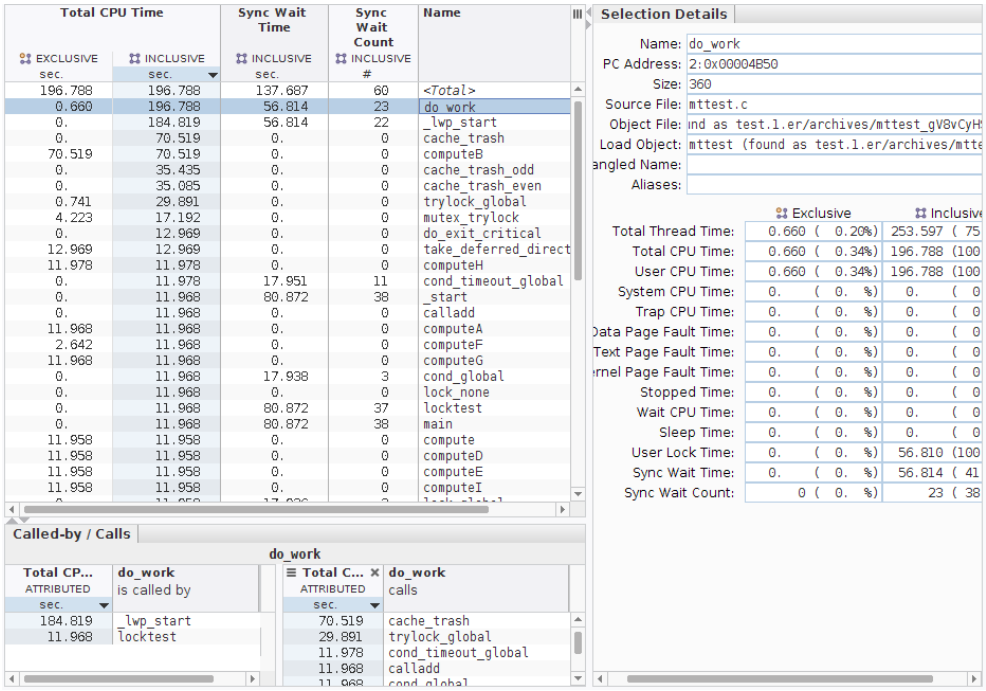


您可以在本教程的以下各节中了解这些度量及其解释。

了解同步跟踪

本节介绍同步跟踪数据并说明如何将其与时钟分析数据相关联。

1. 转至 "Functions"（函数）视图并单击列标题 "Inclusive Total CPU"（包含总 CPU 时间）来根据包含 "Total CPU Time"（总 CPU 时间）进行排序。
2. 选择列表顶部的 `do_work（）` 函数。

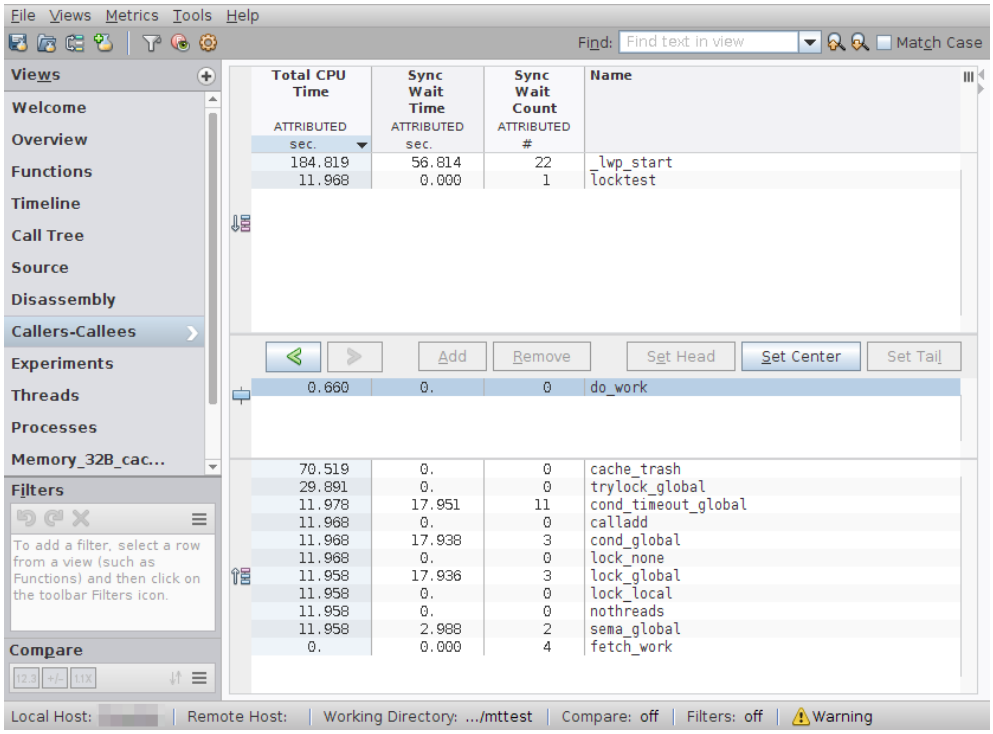


3. 查看 "Functions"（函数）视图底部的 "Called-by/Calls"（调用方/调用）面板，并注意从两个位置调用 `do_work()`，并且其调用十个函数。

   通常情况下，创建处理数据的线程时会调用 `do_work()`，并在从 `_lwp_start()` 调用时显示它。在一种情况下，从 `locktest()` 被调用后，`do_work()` 调用名为 `nothreads()` 的单线程任务。

   `do_work()` 调用的十个函数表示十个不同任务，每个任务使用程序执行的一个不同的同步方法。在从 `mttest` 创建的一些实验中，您可能会看到第十一个函数，该函数使用相对较少的时间获取其他任务的工作块。函数 `fetch_work()` 显示在前面屏幕抓图中的 "Calls"（调用）面板中。

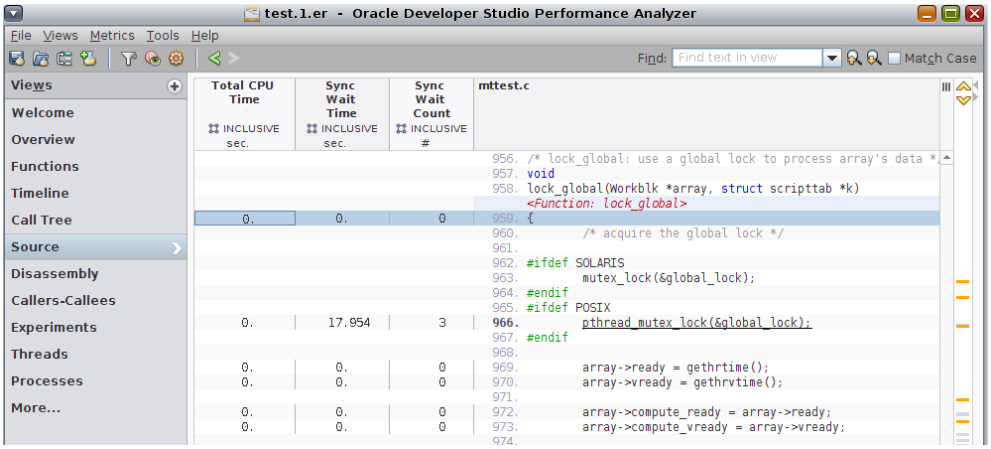
   请注意，除了 "Calls"（调用）面板中的前两个被调用方，所有被调用方都显示大约相同的 "Attributed Total CPU"（归属总 CPU）时间量（约为 10.6 秒）。
4. 切换到 "Callers-Callees"（调用方-被调用方）视图。



"Callers-Callees"（调用方-被调用方）视图显示与 "Called-by/Calls"（调用方/调用）面板相同的调用方和被调用方，但是还显示 "Overview"（概述）页面中选择的其他度量，包括 "Attributed Sync Wait Time"（归属同步等待时间）。

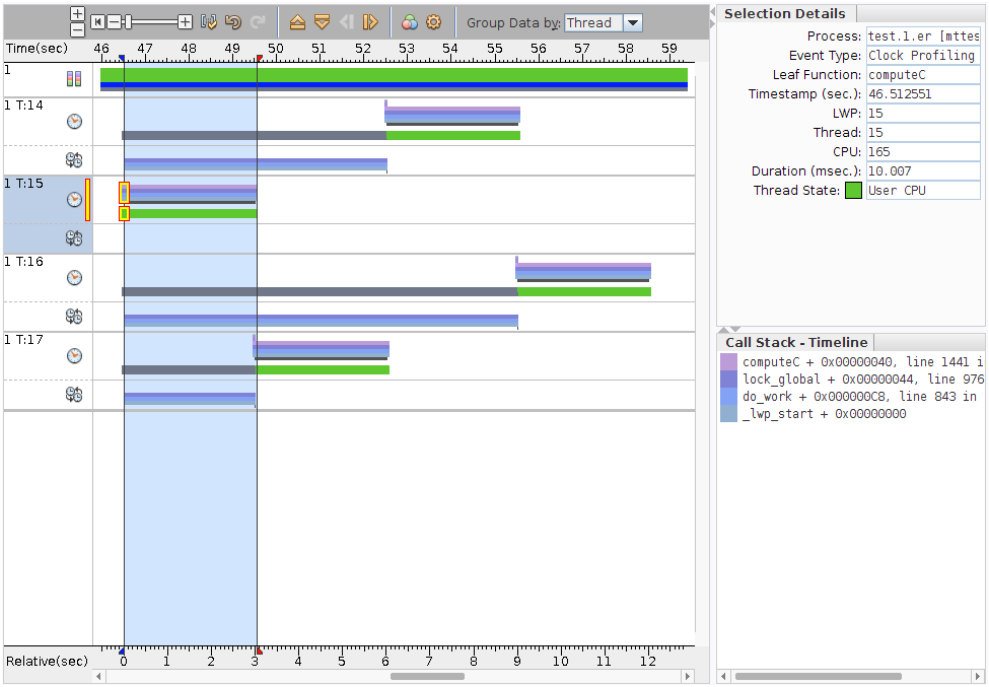
查找 `lock_global（）` 和 `lock_local（）` 两个函数，并注意它们显示大约相同的 "Attributed Total CPU"（归属总 CPU）时间量，但是非常不同的 "Attributed Sync Wait Time"（归属同步等待时间）量。

5. 选择 `lock_global（）` 函数并切换到 "Source"（源）视图。



请注意，所有 "Sync Wait"（同步等待）时间都位于包含对 `pthread_mutex_lock (&global_lock)` 的调用的行上，其 "Total CPU Time"（CPU 总时间）为 0。正如您可以从函数名称猜测到的，执行此任务的四个线程都在获取全局锁时执行其工作，它们逐个获取全局锁。

6. 转回 "Functions"（函数）视图并选择 `lock_global（）`，然后单击 "Filter"（过滤器）图标 并选择 "Add Filter: Include only stacks containing the selected functions"（添加过滤器：仅包括含有所选函数的堆栈）。
7. 选择 "Timeline"（时间线）视图，您应该看到四个线程。
8. 突出显示时间线中发生事件的区域，右键单击后选择 "Zoom"（缩放）-> "Zoom to Selected Time Range"（缩放至所选的时间范围），以此来放大感兴趣的区域。
9. 检查四个线程并比较等待时间与计算时间。



注 - 您在自己的实验中可能会看到不同的线程执行和等待时间。

获取锁的第一个线程（屏幕抓图中的 T:15）工作 ~2.97 秒，然后放弃该锁。您可以看到该线程的状态条为绿色，表示其所有时间都花费在 "User CPU Time"（用户 CPU 时间），没有任何时间花费在 "User Lock Time"（用户锁定时间）。还请注意，使用  标记的 "Synchronization Tracing Call Stacks"（同步跟踪调用堆栈）的第二个条未显示此线程的调用堆栈。

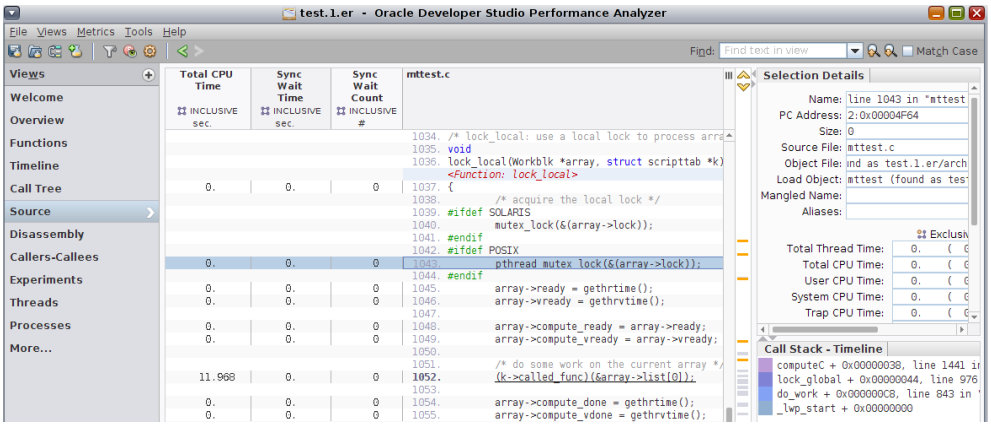
第二个线程（屏幕抓图中的 T:17）已经在 "User Lock Time"（用户锁定时间）等待了 2.99 秒，然后计算了 ~2.90 秒并放弃该锁。"Synchronization Tracing Call Stacks"（同步跟踪调用堆栈）与 "User Lock Time"（用户锁定时间）一致。

第三个线程 (T:14) 已经在 "User Lock Time"（用户锁定时间）等待了 5.98 秒，然后计算了 ~2.95 秒并放弃该锁。

最后一个线程 (T:16) 已经在 "User Lock Time"（用户锁定时间）等待了 8.98 秒，然后计算了 2.84 秒。总计算时间为 2.97+2.90+2.95+2.84 或大约 11.7 秒。

总同步等待时间为 2.99 + 5.98 + 8.98 或大约 17.95 秒，这可以在 "Function"（函数）视图中确认（该视图中报告的值为 17.954 秒）。

10. 通过单击 "Active Filters"（活动过滤器）面板中的 X 来删除过滤器。
11. 转回 "Function"（函数）视图，选择函数 `lock_local`（），然后切换到 "Source"（源）视图。



请注意，包含对 `pthread_mutex_lock(&array->lock)` 的调用的行（屏幕抓图中的行 1043）上的 "Sync Wait Time"（同步等待时间）为 0。这是因为锁对于工作块来说是本地的，所以不存在争用，所有四个线程可以同时进行计算。

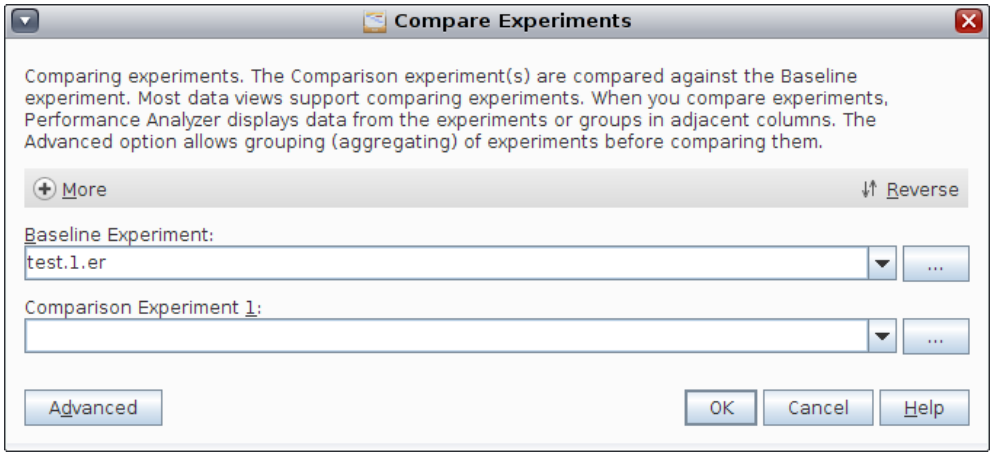
您查看的实验是使用校准阈值记录的。在下一节中，您将与另一个实验进行比较，您运行 `make` 命令时使用零阈值记录该实验。

使用同步跟踪比较两个实验

在本节中将比较两个实验。使用用于记录事件的校准的阈值记录 `test.1.er` 实验，使用零阈值记录 `test.2.er` 实验以包括 `mttest` 程序执行中发生的所有同步事件。

1. 单击工具栏上的 "Compare Experiments"（比较实验）按钮  ，或选择 "File"（文件）-> "Compare Experiments"（比较实验）。

   此时将打开 "Compare Experiments"（比较实验）对话框。

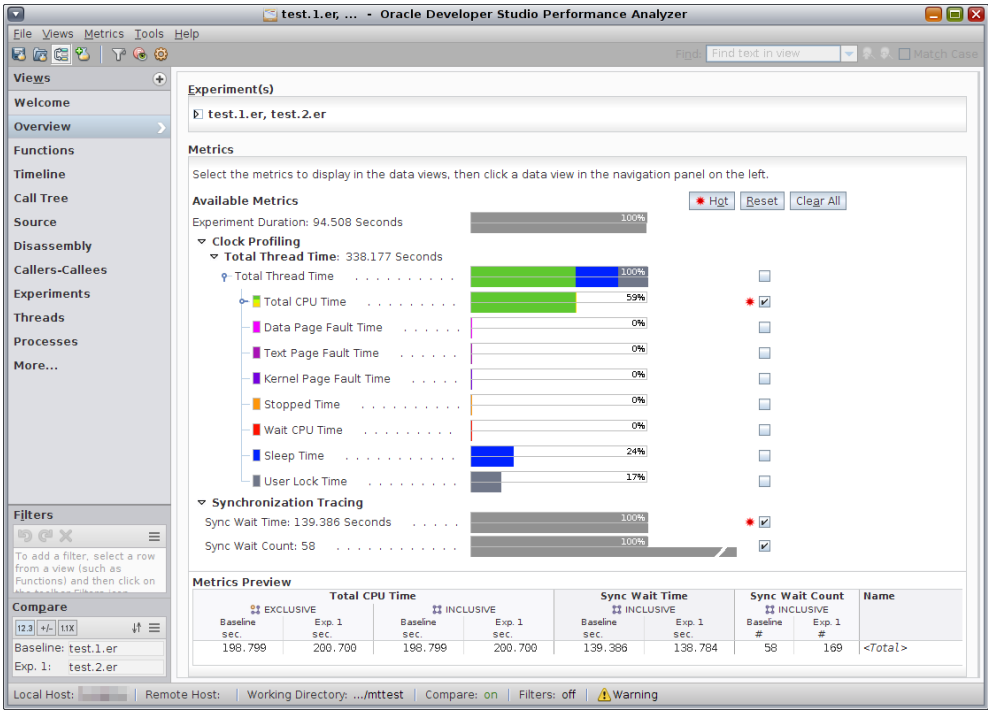


您已经打开的 `test.1.er` 实验列在 "Baseline"（基线）组中。必须添加实验以便与 "Comparison Group"（比较组）面板中的基线实验进行比较。

有关比较实验以及添加多个实验与基线进行比较的更多信息，请单击对话框中的 "Help"（帮助）按钮。

2. 单击 "Comparison Experiment 1"（比较实验 1）旁边的 ... 按钮，然后在 "Select Experiment"（选择实验）对话框中打开 `test.2.er` 实验。
3. 在 "Compare Experiments"（比较实验）对话框中单击 "OK"（确定）以装入第二个实验。

   此时将重新打开 "Overview"（概述）页面，其中包括两个实验的数据。



"Clock Profiling"（时钟分析）度量对每个度量显示两个带颜色的条，每个实验对应一个条。`test.1.er` "Baseline"（基线）实验中的数据位于上面。

如果在数据条上移动鼠标光标，弹出文本显示 "Baseline"（基线）和 "Comparison"（比较）组的数据以及它们之间的差异（以数值和百分比表示）。

请注意，这里记录的 CPU 总时间要比第二个实验中的时间长一点，但几乎是 "Sync Wait Counts"（同步等待计数）的三倍。

4. 切换至 "Functions"（函数）视图，单击 "Inclusive Sync Wait Time Count"（包括同步等待时间计数）列下标有 "Baseline"（基线）的子列标题，按第一个实验中的事件数对函数进行排序。

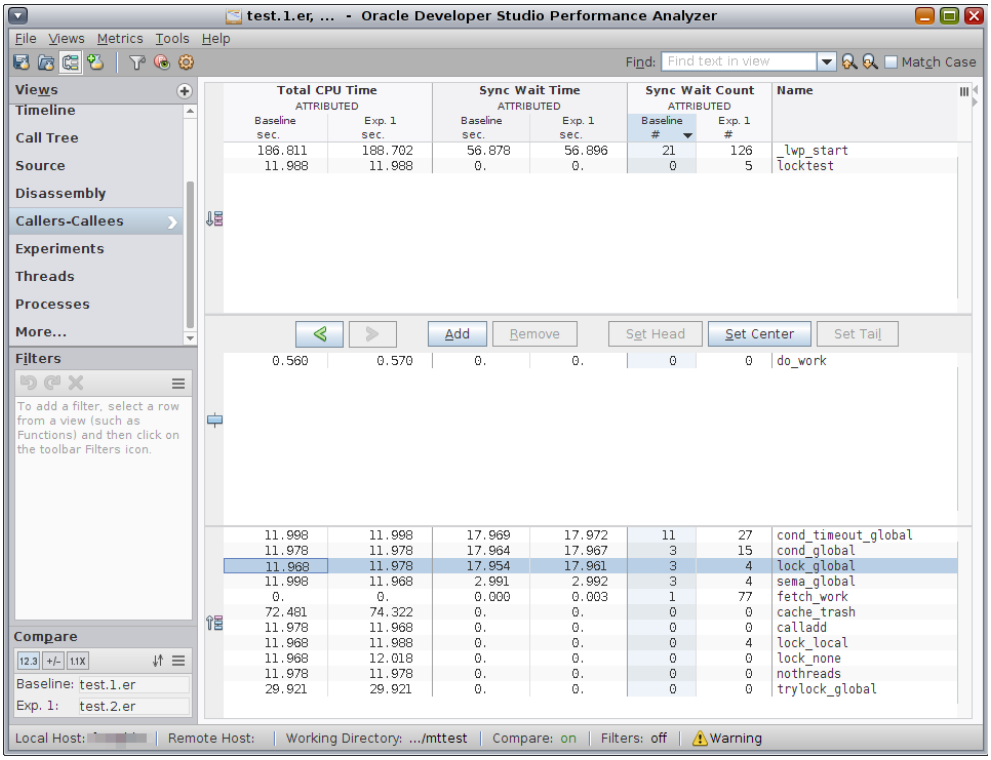
Total CPU Time				Sync Wait Time		Sync Wait Count		Name
EXCLUSIVE		INCLUSIVE		INCLUSIVE		INCLUSIVE		
Baseline sec.	Exp. 1 sec.	Baseline sec.	Exp. 1 sec.	Baseline sec.	Exp. 1 sec.	Baseline #	Exp. 1 #	
198.799	200.700	198.799	200.700	139.386	138.784	58	169	<Total>
0.560	0.570	198.799	200.690	56.878	56.896	21	131	do_work
0.	0.	186.811	188.702	56.878	56.896	21	126	_lwp_start
0.	0.	0.	0.	0.000	0.003	1	77	fetch_work
0.	0.	11.988	11.998	82.507	81.888	37	43	_start
0.	0.	11.988	11.998	82.507	81.888	37	43	main
0.	0.	11.988	11.998	82.507	81.888	36	41	locktest
0.	0.	0.	0.	82.507	81.888	36	36	thread_work
0.	0.	11.998	11.998	17.969	17.972	11	27	cond_timeout_global
0.	0.	11.978	11.978	17.964	17.967	3	15	cond_global
0.	0.	11.968	11.978	17.954	17.961	3	4	lock_global
0.	0.	11.968	11.988	0.	0.	0	4	lock_local
0.	0.	11.998	11.968	2.991	2.992	3	4	sema_global
0.	0.	0.	0.	0.000	0.000	1	2	resolve_symbols
0.	0.	0.010	0.	0.	0.	0	0	__collector_write_packet
0.	0.	0.	0.010	0.	0.	0	0	__cond_timedwait
0.	0.010	0.	0.010	0.	0.	0	0	__lwp_park
0.	0.	0.	0.	0.	0.	0	0	__lwp_wait

test.1.er 和 test.2.er 的最大差异位于 do_work（） 中，这包括来自其直接或间接调用的所有函数的差异，包括 lock_global（） 和 lock_local（） 。

提示 - 如果更改比较格式，甚至可以更容易地比较差异。单击工具栏中的 "Settings"（按钮），选择 "Formats"（格式）标签，然后为 "Comparison Style"（比较样式）选择 "Deltas"（增量）。应用更改后，test.2.er 的度量显示为与 test.1.er 中的度量的 + 或 - 差异。在前面的屏幕抓图中，选择的 pthread_mutex_lock（） 函数在 "test.2.er Incl Sync Wait Count"（test.2.er 包含同步等待计数）列中将显示 +88。

---

5. 选择 "Callers-Callees"（调用方-被调用方）视图。



查看其中两个调用方，`lock_global（）` 和 `lock_local（）`。

`lock_global（）` 函数在 `test.1.er` 中对 "Attributed Sync Wait Count"（归属同步等待计数）显示 3 个事件，但是在 `test.2.er` 中显示 4 个事件。原因是获取 `test.1.er` 中的锁的第一个线程未停止，所以未记录该事件。在 `test.2.er` 实验中，阈值设置为记录所有事件，所以甚至记录了第一个线程的锁获取。

同样，在第一个实验中，`lock_local（）` 没有记录的事件，因为不存在锁争用。第二个实验中有 4 个事件，即使总的来说它们具有极小的延迟。

深入了解性能分析器

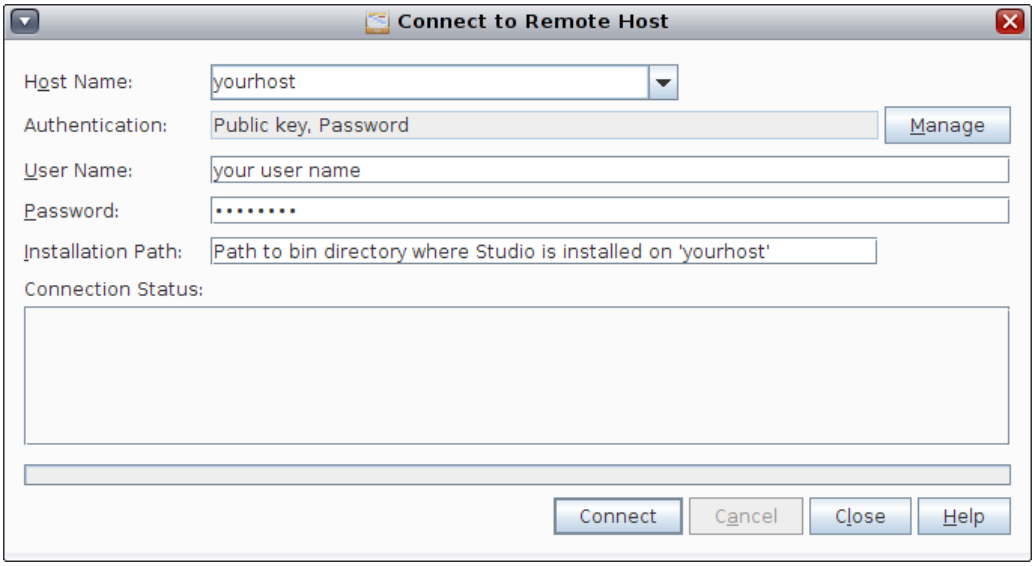
本章将介绍更多教程、使用性能分析器可以开展的其他任务以及在哪里可以找到更多资源。

使用远程性能分析器

可以从受支持的系统或者从无法安装 Oracle Developer Studio 的系统（如 Mac OS 或 Windows）使用远程性能分析器。有关安装和使用此特殊版本性能分析器的信息，请参见《Oracle Developer Studio 12.5：性能分析器》 中的 “远程使用性能分析器”。

远程调用性能分析器时，可看到相同的 "Welcome"（欢迎）页面，但是用于创建和查看实验的选项被禁用并灰显。

单击 "Connect to Remote Host"（连接到远程主机），性能分析器将打开连接对话框：



键入要连接到的系统的名称、您在该系统上的用户名和口令以及该系统上 Oracle Developer Studio 安装的安装路径。单击 "Connect"（连接），性能分析器将使用您的名称和口令登录到远程系统，并验证连接。

此后，"Welcome"（欢迎）页面看起来就像使用本地性能分析器时一样，不同之处在于底部的状态区域显示您连接到的远程主机的名称。从此处继续执行上面的步骤 2。

其他教程

如性能分析器教程简介中所述，在"获取教程的样例代码" [10]中下载的样例 Zip 文件的 `PerformanceAnalyzer` 子目录中还有若干其他教程。以下列表分别详细说明了这些教程：

`cachetest` — `cachetest` 教程说明了编译器优化对代码性能的影响，并介绍了 Oracle Developer Studio 编译器提供的编译器注释，以帮助您了解相关优化。

`ksynprog` — `ksynprog` 教程说明了如何通过运行任务来触发一个经过内核的路径以及如何收集生成的程序的性能数据。

`omptest` — `omptest` 教程说明了如何使用可在性能分析器中查看的 OpenMP 并行化指令和生成的性能特性。

`synprog` — `synprog` 教程介绍了一个简单的程序，通过该程序可执行一系列任务并展示性能分析器的部分性能特性或功能。

更多信息

以下资源提供了有关性能分析器及相关的数据收集工具的更多信息：

- 性能分析器中的集成帮助系统
- 《Oracle Developer Studio 12.5：性能分析器》
- Oracle Developer Studio 开发人员门户 (http://www.oracle.com/technetwork/server-storage/solarisstudio/) 上提供的文章和白皮书。
- 《Oracle Developer Studio 12.5 发行版的新增功能》 中的 第 5 章, “性能分析工具”
- 《Oracle Developer Studio 12.5：发行说明》 中的 “性能分析器和 er_print 实用程序限制”