Oracle® Developer Studio 12.5: パフォーマンスアナライザチュートリアル

Part No: E71971
2016 年 6 月

ORACLE

Part No: E71971

ドキュメントのアクセシビリティについて

オラクルのアクセシビリティについての詳細情報は、Oracle Accessibility ProgramのWeb サイト(http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc)を参照してください。

Oracle Supportへのアクセス

サポートをご契約のお客様には、My Oracle Supportを通して電子支援サービスを提供しています。詳細情報は(http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info)か、聴覚に障害のあるお客様は (http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs)を参照してください。

目次

このドキュメントの使用法

- **概要** – サンプルプログラムで Oracle Developer Studio 12.5 パフォーマンスアナライザを使用するための段階的な手順について説明します。
- **対象読者** - アプリケーション開発者、開発者、アーキテクト、サポートエンジニア
- **必要な知識** - プログラミングの経験、プログラム/ソフトウェア開発テスト、ソフトウェア製品の構築およびコンパイルの能力

製品ドキュメントライブラリ

この製品および関連製品のドキュメントとリソースは http://docs.oracle.com/cd/E60778_01 で入手可能です。

フィードバック

このドキュメントに関するフィードバックを http://www.oracle.com/goto/docfeedback からお聞かせください。

パフォーマンスアナライザのチュートリアルの概要

パフォーマンスアナライザは、Java、C、C++、および Fortran アプリケーションのパフォーマンスを調べるための Oracle Developer Studio ツールです。これを使用して、アプリケーションの実行パフォーマンスを理解して、問題の領域を見つけることができます。これらのチュートリアルは、段階的な手順を使用して、サンプルプログラムでパフォーマンスアナライザを使用する方法を示します。

パフォーマンスアナライザのチュートリアルについて

このドキュメントでは、パフォーマンスアナライザを使用してさまざまなタイプのプログラムをプロファイルする方法を示すいくつかのチュートリアルを提供します。各チュートリアルには、ソースファイルとともにパフォーマンスアナライザを使用するための手順があり、チュートリアルのほとんどの手順にはスクリーンショットが含まれています。

すべてのチュートリアルのソースコードが単一のディストリビューションに含まれています。サンプルソースコードの入手については、10 ページの「チュートリアルのサンプルコードの入手」を参照してください。

チュートリアルは次のとおりです。

- 「C プロファイリングの概要」

  この入門チュートリアルでは、C で記述された `lowfruit` という名前のターゲットコードを使用します。`lowfruit` プログラムは非常に単純であり、それぞれが効率的な方法と非効率的な方法で実装された 2 つのプログラミングタスクのコードを含んでいます。このチュートリアルでは、C ターゲットプログラムでパフォーマンスの実験を収集する方法と、パフォーマンスアナライザでさまざまなデータビューを使用する方法を示します。各タスクの 2 つの実装を調べて、効率的なタスクと効率的ではないタスクがパフォーマンスアナライザでどのように示されるかを確認します。

- 「Java プロファイリングの概要」

  この入門チュートリアルでは、Java で記述された `jlowfruit` という名前のターゲットコードを使用します。C プロファイリングチュートリアルで使用されるコードと類似した `jlowfruit` プログラムは非常に単純であり、それぞれが効率的な方法と非効率的な方法で実装された 2 つのプログラミングタスクのコードを含んでいます。このチュートリアルでは、Java ターゲットでパフォーマンスの実験を収集する方法と、パフォーマンスアナライザでさまざまなデータビューを使用する方法を示します。各タスクの 2 つの実装を調べて、効率的なタスクと効率的ではないタスクがパフォーマンスアナライザでどのように示されるかを確認します。

- 「Java および Java-C++ 混合プロファイリング」

  このチュートリアルは、多数のプログラミング操作を 1 つずつ実行する `jsynprog` という名前の Java コードに基づいています。算術を行うもの、ガベージコレクションを呼び出すもの、動的にロードされた C++ 共有オブジェクトを使用して、Java からネイティブコードへの呼び出しとその逆の呼び出しを行うものがあります。このチュートリアルでは、さまざまな操作の実装方法と、プログラムに関するパフォーマンスデータがパフォーマンスアナライザで表示される方法を確認します。
- 「マルチスレッドプログラムでのハードウェアカウンタプロファイリング」

  このチュートリアルは、多数のタスクを実行して各タスクのスレッドを生成し、タスクごとに異なる同期手法を使用する、`mttest` という名前のマルチスレッドプログラムに基づいています。このチュートリアルでは、タスクにおける計算の間のパフォーマンスの違いを確認して、ハードウェアカウンタプロファイリングを使用して、2 つの関数の間の予期しないパフォーマンスの違いを調べて理解します。
- 「マルチスレッドプログラムでの同期トレース」

  このチュートリアルも、多数のタスクを実行して各タスクのスレッドを生成し、タスクごとに異なる同期手法を使用する、`mttest` という名前のマルチスレッドプログラムに基づいています。このチュートリアルでは、同期手法の間のパフォーマンスの違いを調べます。

チュートリアルのサンプルコードの入手

パフォーマンスアナライザのチュートリアルで使用されるプログラムは、すべての Oracle Developer Studio ツールに使用されるコードが含まれているディストリビューション内にあります。以前にダウンロードしていない場合にサンプルコードを入手するには、次の手順を使用します。

1. Oracle Developer Studio の Web ページ `http://www.oracle.com/technetwork/server-storage/solarisstudio` にある Oracle Developer Studio 12.5 のサンプルアプリケーションのページにアクセスします。
2. Oracle Developer Studio の Web ページのダウンロードセクションに移動します。
3. ページのリンクからライセンスを読んで、「Accept」を選択して同意します。
4. リンクをクリックして zip ファイルをダウンロードし、ダウンロードページの手順を使用して unzip します。

サンプルファイルをダウンロードして展開したあとで、`OracleDeveloperStudio12.5-Samples/PerformanceAnalyzer` ディレクトリ内にサンプルを見つけることができます。

このディレクトリには、このドキュメントでは説明されていない追加のサンプルである `cachetest`、`ksynprog`、`omptest`、および `synprog` が含まれています。各サンプルのサブディレクトリには、`Makefile` と、パフォーマンスアナライザの追加のデモンストレーションに使用できる手順が記載された `README` ファイルが含まれています。

環境をチュートリアル用に設定

チュートリアルを試す前に、『Oracle Developer Studio 12.5: インストールガイド』の 第 5 章, 「Oracle Developer Studio 12.5 のインストール後」で説明されているように、パス上に Oracle Developer Studio `bin` ディレクトリがあること、パスに適切な Java バージョンが含まれていることを確認します。

プログラムを構築できるように、`make` または `gmake` コマンドもパス上に必要です。

C プロファイリングの概要

この章では、次の内容について説明します。

C プロファイリングチュートリアルについて

このチュートリアルでは、Oracle Developer Studio パフォーマンスアナライザを使用したプロファイリングのもっとも簡単な例を示して、パフォーマンスアナライザを使用してパフォーマンスの実験を収集して検査する方法について説明します。このチュートリアルでは、「概要」、「関数」ビュー、「ソース」ビュー、「タイムライン」を使用します。

プログラム `lowfruit` は、2 つの異なるタスク (ループで初期化するためのタスクと、順序付きリストに数値を挿入するためのタスク) を実行する単純なプログラムです。それぞれのタスクは、非効率的な方法とより効率的な方法の 2 回実行されます。

ヒント - 「Java プロファイリングの概要」のチュートリアルでは、同等の Java プログラムを使用して、パフォーマンスアナライザを使用した類似のアクティビティーを示します。

記録する実験で目にするデータは、ここで示されているものとは異なります。チュートリアルのスクリーンショットに使用される実験は、Oracle Solaris 11.3 が実行されている SPARC T5 システムで記録されました。Oracle Solaris または Linux を実行する x86 システムからのデータは異なります。さらに、データ収集には統計的な性質があり、同じシステムおよび OS で実行する場合でも実験ごとに異なります。

パフォーマンスアナライザの実際のウィンドウ構成は、スクリーンショットと厳密に一致しない場合があります。パフォーマンスアナライザでは、ウィンドウのコンポーネント間の区切りバーをドラッグしたり、コンポーネントを縮小したり、ウィンドウのサイズを変更したりできます。パフォーマンスアナライザはその構成を記録し、次回の実行時に同じ構成を使用します。チュートリアルで示しているスクリーンショットの取得過程で、多くの構成を変更しました。

このチュートリアルは、Oracle Developer Studio がインストールされているシステムでローカルに実行されます。107 ページの「リモートパフォーマンスアナライザの使用」の説明に従って、リモートで実行することもできます。

lowfruit サンプルコードの設定

始める前に:

コードの取得および環境の設定については、次の情報を参照してください。

- 10 ページの「チュートリアルのサンプルコードの入手」
- 11 ページの「環境をチュートリアル用に設定」

1. 次のコマンドを使用して、lowfruit ディレクトリの内容を独自のプライベート作業領域にコピーします。

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

   directory は使用する作業ディレクトリです。
2. 作業ディレクトリに移動します。

   ```
   % cd directory/lowfruit
   ```
3. ターゲットの実行可能ファイルをビルドします。

   ```
   % make clobber

   % make
   ```

   注記 - clobber サブコマンドコマンドは、このディレクトリで以前に make を実行した場合にのみ必要ですが、どの場合にも安全に使用できます。

   make の実行後に、このディレクトリには、チュートリアルで使用するターゲットプログラムと、lowfruit という名前の実行可能ファイルが含められます。

   次のセクションでは、パフォーマンスアナライザを使用して lowfruit プログラムからデータを収集して、実験を作成する方法を示します。

ヒント - 必要に応じて、次のために Makefile を編集できます。デフォルトの Oracle Developer Studio コンパイラではなく GNU コンパイラを使用します。デフォルトの 64 ビットではなく 32 ビットでビルドします。各種のコンパイラフラグを追加します。

パフォーマンスアナライザを使用したデータの収集

このセクションでは、パフォーマンスアナライザの「アプリケーションのプロファイル」機能を使用して、実験のデータを収集する方法について説明します。

ヒント - アプリケーションをプロファイルする方法を確認するためにこれらの手順に従わない方が望ましい場合は、`lowfruit` の `Makefile` に含まれている `make` ターゲットを使用して実験を記録できます。

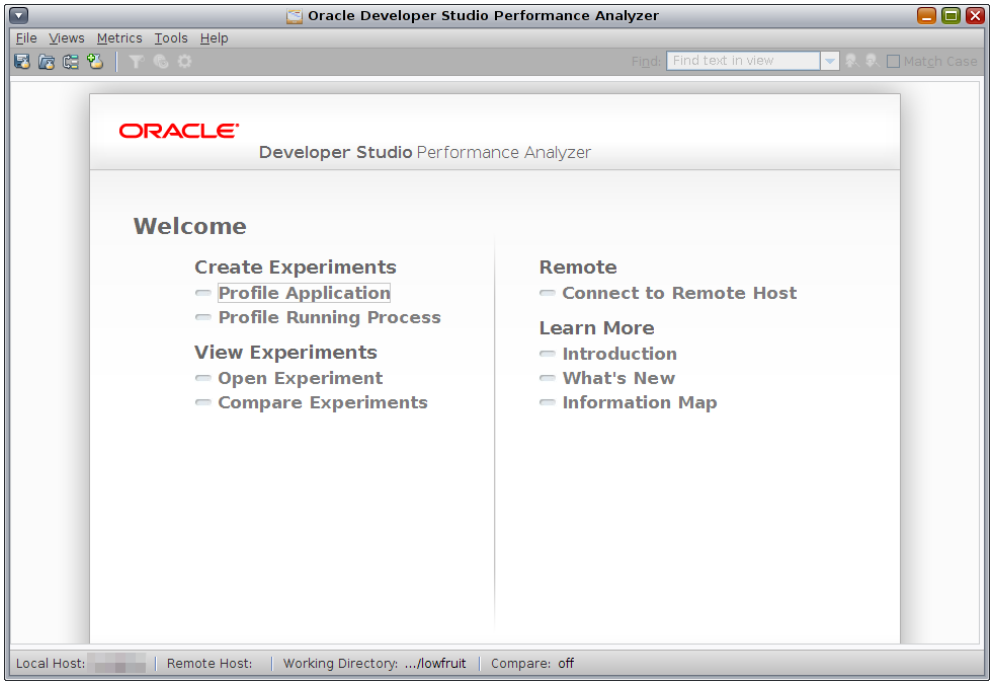
```
make collect
```

`collect` ターゲットは `collect` コマンドを起動して、このセクションでパフォーマンスアナライザを使用して作成するものと同じような実験を記録します。その後、19 ページの「パフォーマンスアナライザを使用した `lowfruit` データの検査」にスキップできます。

1. `lowfruit` ディレクトリを開いているときに、パフォーマンスアナライザを起動します。

   ```
   % analyzer
   ```

   パフォーマンスアナライザが起動し、「ようこそ」ページが表示されます。

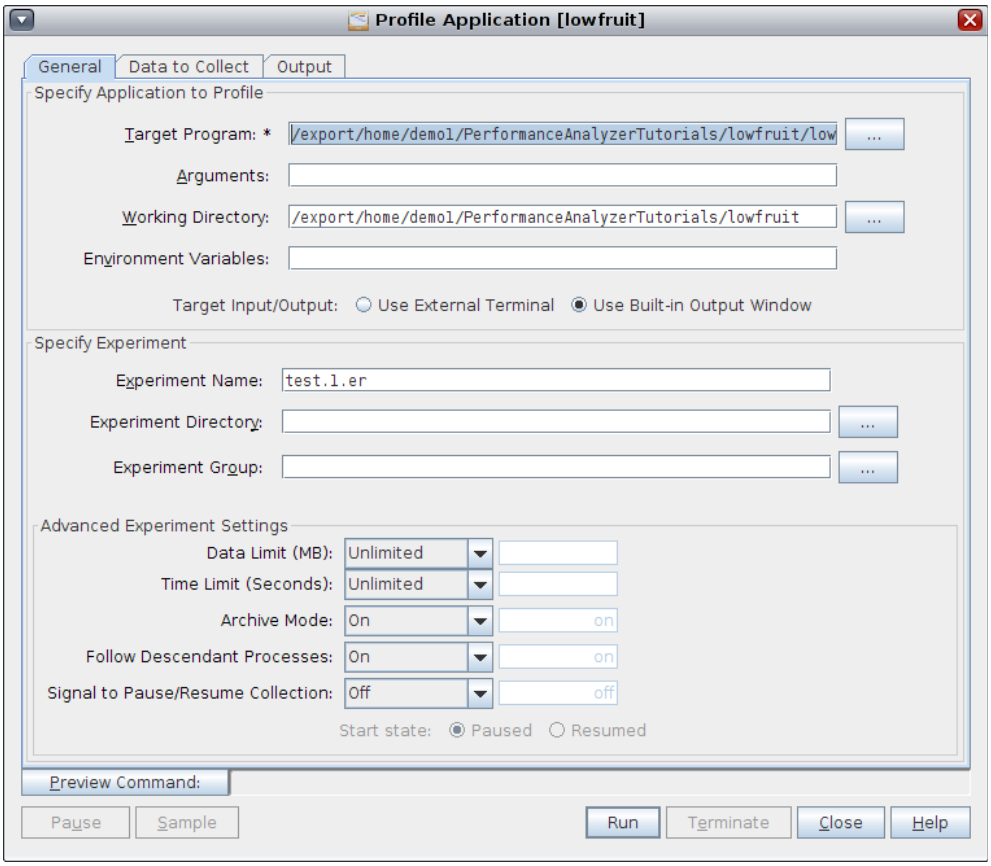


   パフォーマンスアナライザをはじめて使用する場合、最近の実験は「実験を開く」項目の下には表示されません。前に使用したことがある場合、現在パフォーマンスアナライザを実行しているシステムから最近開いた実験のリストが表示されます。

2. 「ようこそ」ページの「実験の作成」の下にある「アプリケーションのプロファイル」リンクをクリックします。

「一般」タブが選択された「アプリケーションのプロファイル」ダイアログボックスが開きます。このページのオプションは、「プロファイルするアプリケーションの指定」、「実験の指定」、および「拡張実験設定」に分類されています。

3. 「ターゲットプログラム」フィールドに、プログラム名 `lowfruit` を入力します。



ヒント - パフォーマンスアナライザを起動して、コマンド `analyzer lowfruit` を使用してパフォーマンスアナライザを起動するときにターゲット名を指定することで、すでに入力されたプログラム名を使用してこのダイアログボックスを直接開くこともできます。この方法は、パフォーマンスアナライザをローカルで実行する場合にのみ機能します。

---

4. 「プロファイルするアプリケーションの指定」パネルの下部にある「ターゲット入力/出力」オプションでは、「組み込み出力ウィンドウの使用」を選択します。

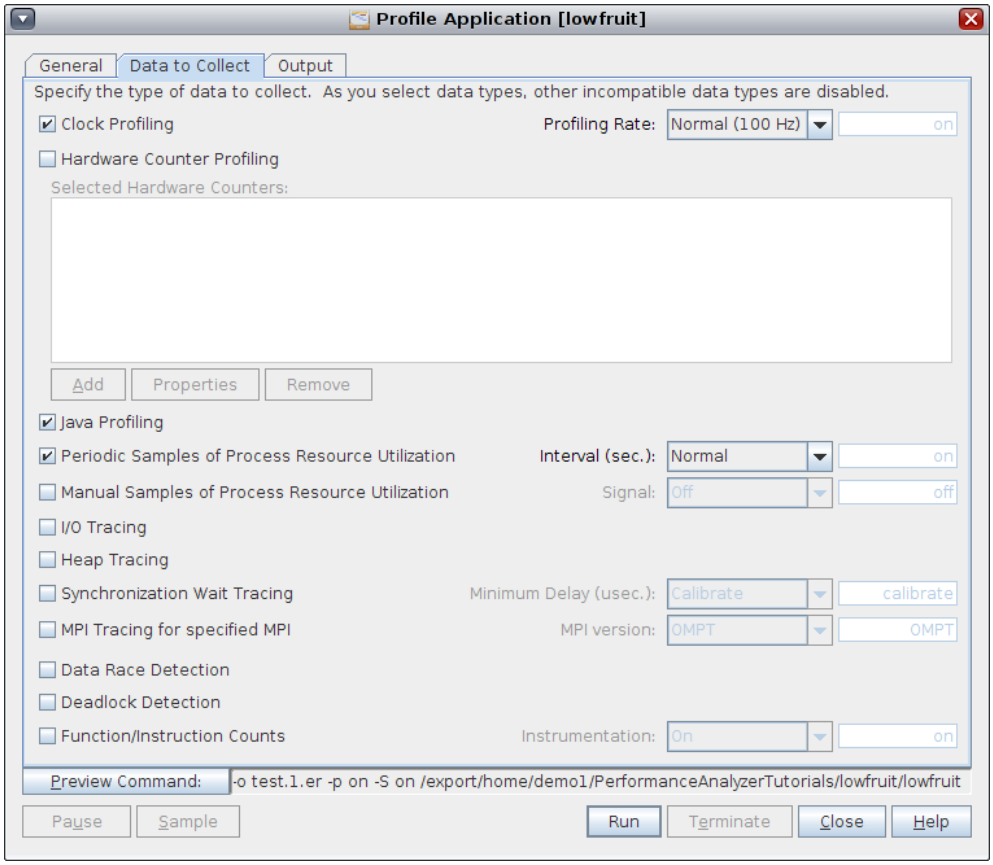
   「ターゲット入力/出力」オプションは、ターゲットプログラム `stdout` および `stderr` がリダイレクトされるウィンドウを指定します。デフォルト値は「外部端末の使用」ですが、このチュー

トリアルでは、「パフォーマンスアナライザ」ウィンドウのすべてのアクティビティーを維持するために、「ターゲット入力/出力」オプションが「組み込み出力ウィンドウの使用」に変更されました。このオプションを使用すると、「アプリケーションのプロファイル」ダイアログボックスの「出力」タブには「`stdout`」と「`stderr`」が表示されます。

リモートで実行している場合、組み込み出力ウィンドウのみがサポートされるため、「ターゲット入力/出力」オプションはありません。

5. 「実験名」オプションでは、デフォルトの実験名は `test.1.er` ですが、名前が `.er` で終わりまだ使用中ではないかぎり、別の名前に変更できます。
6. 「収集するデータ」タブをクリックします。

   「収集するデータ」では、収集するデータのタイプを選択して、すでに選択されているデフォルトを表示できます。



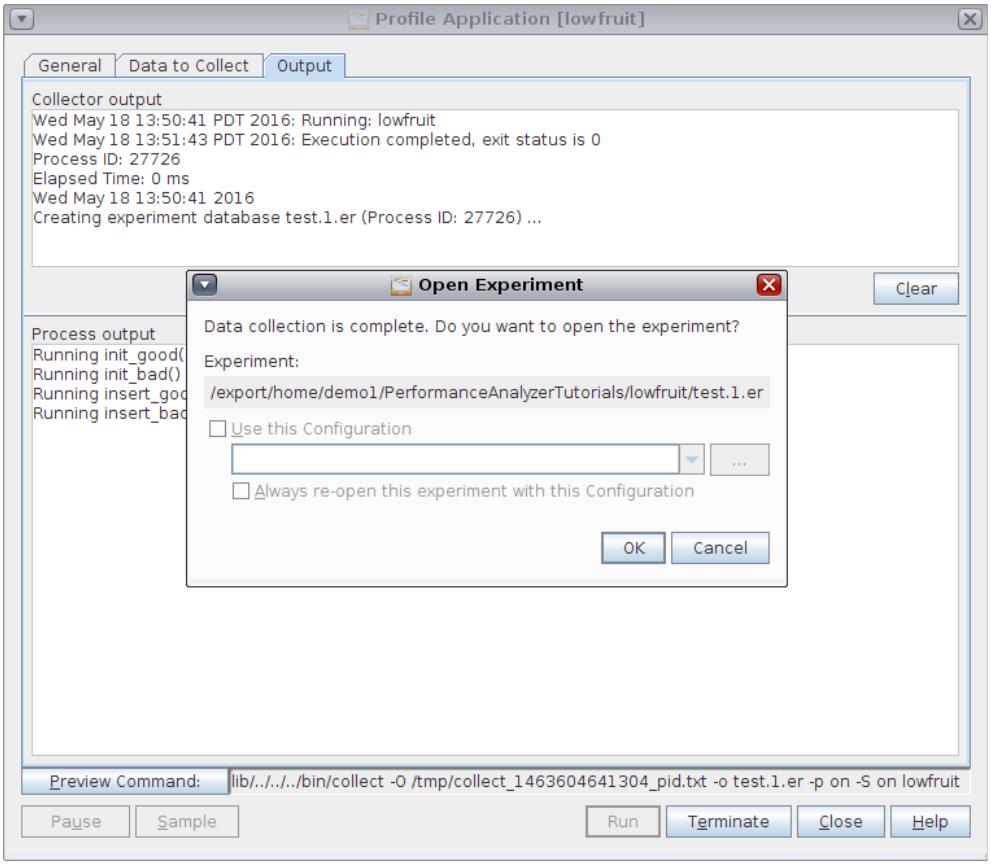
スクリーンショットで確認できるように、「Java プロファイリング」はデフォルトで有効になっていますが、`lowfruit` などの Java 以外のターゲットでは無視されます。

オプションで、「プレビューコマンド」ボタンをクリックして、プロファイリングの開始時に実行する `collect` コマンドを表示できます。

7. 「実行」ボタンをクリックします。

   「アプリケーションのプロファイル」ダイアログボックスには「出力」タブが表示され、「プロセス出力」パネルでプログラムの実行時にプログラム出力が示されます。

   プログラムの完了後に、記録した実験を開くかどうかがダイアログボックスで尋ねられます。



8. ダイアログボックスで「了解」をクリックします。

   実験が開きます。次のセクションでは、データの検査方法を示します。

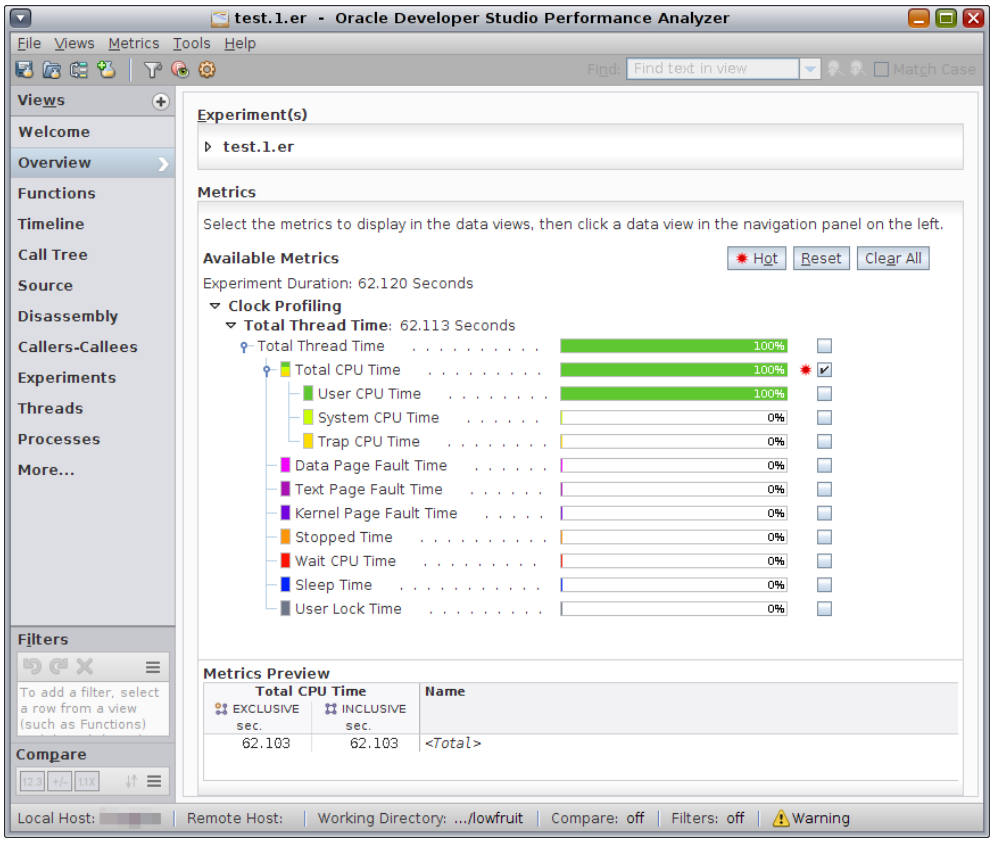
パフォーマンスアナライザを使用した lowfruit データの検査

このセクションでは、lowfruit サンプルコードから作成された実験のデータを調べる方法を示します。

1. 前のセクションで作成した実験がまだ開いていない場合は、lowfruit ディレクトリからパフォーマンスアナライザを起動して、次のようにして実験をロードできます。

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

実験が開いたら、パフォーマンスアナライザに「概要」画面が表示されます。



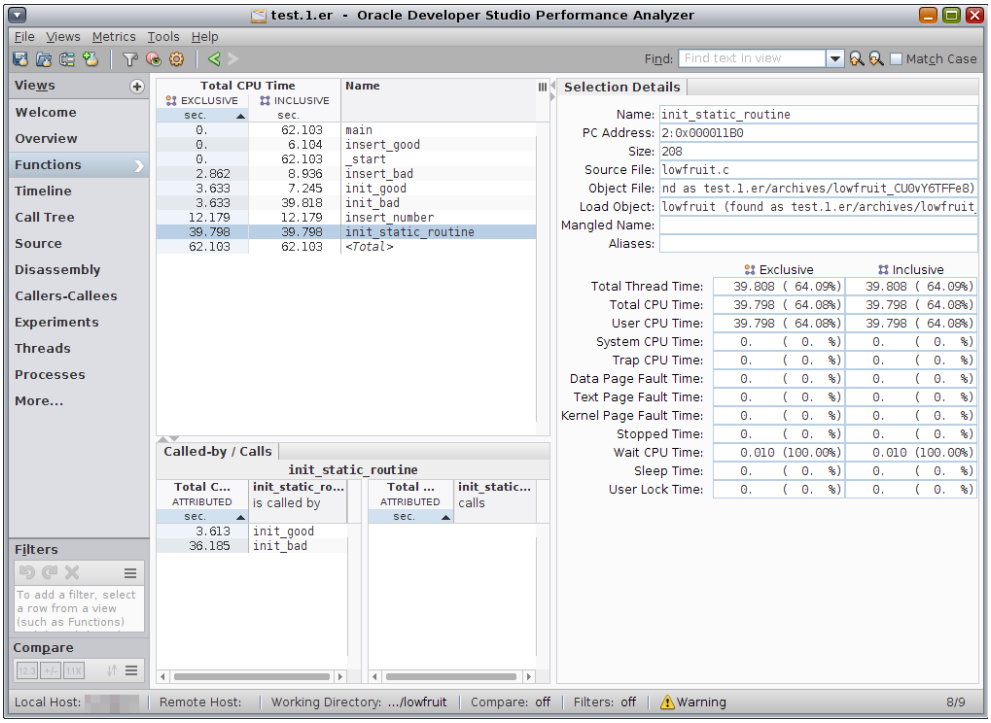
この実験では、「概要」には基本的に 100% の「ユーザー CPU 時間」が示されます。プログラムはシングルスレッドであり、その 1 つのスレッドは CPU の制約を受けます。実験は Oracle Solaris システムに記録され、「概要」には記録されたメトリックが 12 個表示されますが、「CPU 時間合計」のみがデフォルトで有効になっています。

色付きのインジケータがあるメトリックは、Oracle Solaris によって定義された 10 個のマイクロステートで費やされた時間です。これらのメトリックには、さまざまな待ち時間のほかに、一緒にすると「CPU 時間合計」と等しくなる「ユーザー CPU 時間」、「システム CPU 時間」、および「トラップ CPU 時間」が含まれます。「スレッド合計時間」は、すべてのマイクロステートの合計です。

Linux ではマイクロステートアカウンティングがサポートされないため、Linux マシンでは「CPU 時間合計」のみが記録されます。

デフォルトでは、「包括的 CPU 合計時間」と「排他的 CPU 合計時間」の両方のプレビューが表示されます。メトリックの*包括的*とは、そのメトリックが呼び出すすべての関数またはメソッドで集計されたメソッドを含む、その関数またはメソッドのメトリック値を指します。*排他的*とは、その関数またはメソッド内で集計されたメトリックのみを指します。

2. 左側の「ビュー」ナビゲーションバージョンで「関数」ビューをクリックするか、メニューバーから「ビュー」->「関数」を使用して選択します。



「関数」ビューには、アプリケーション内の関数のリストがそれぞれの関数のパフォーマンスメトリックとともに表示されます。リストは、最初は各関数で使用された「排他的 CPU 合計時間」でソートされます。リストには、ターゲットアプリケーションのすべての関数とプログラムで使用するすべての共有オブジェクトが含まれています。最上位の関数 (もっともコストが高い関数) がデフォルトで選択されています。

右側の「選択の詳細」ウィンドウには、選択した関数について記録されたすべてのメトリックが表示されます。

関数リストの下にある「呼び出し元/呼び出し回数」パネルは、選択した関数に関する詳細情報を提供し、2 つのリストに分割されています。「呼び出し元」リストには選択した関数の呼び出し元が表示され、メトリック値は、その呼び出し元への関数の合計メトリックの属性を示しています。「呼び出し回数」リストは、選択した関数の呼び出し先を示し、呼び出し先の包括的メトリックが選択した関数の合計メトリックにどのように関係したかを示しています。「呼び出し元/呼び出し回数」パネルのいずれかのリストで関数をダブルクリックすると、その関数が、メインの「関数」ビューで選択した関数になります。

3. さまざまな関数を選択してみて、選択の変更によって「関数」ビューのウィンドウが更新される方法を確認します。

   「選択の詳細」ウィンドウには、「ロードオブジェクト」フィールドに示されているように、ほとんどの関数が `lowfruit` 実行可能ファイルから生成されることが示されます。

   また、列ヘッダーをクリックしてみて、ソートを「排他的 CPU 合計時間」から「包括的 CPU 合計時間」に変更したり、「名前」でソートに変更したりすることもできます。

4. 「関数」ビューで、2 つのバージョンの初期化タスク `init_bad()` と `init_good()` を比較します。

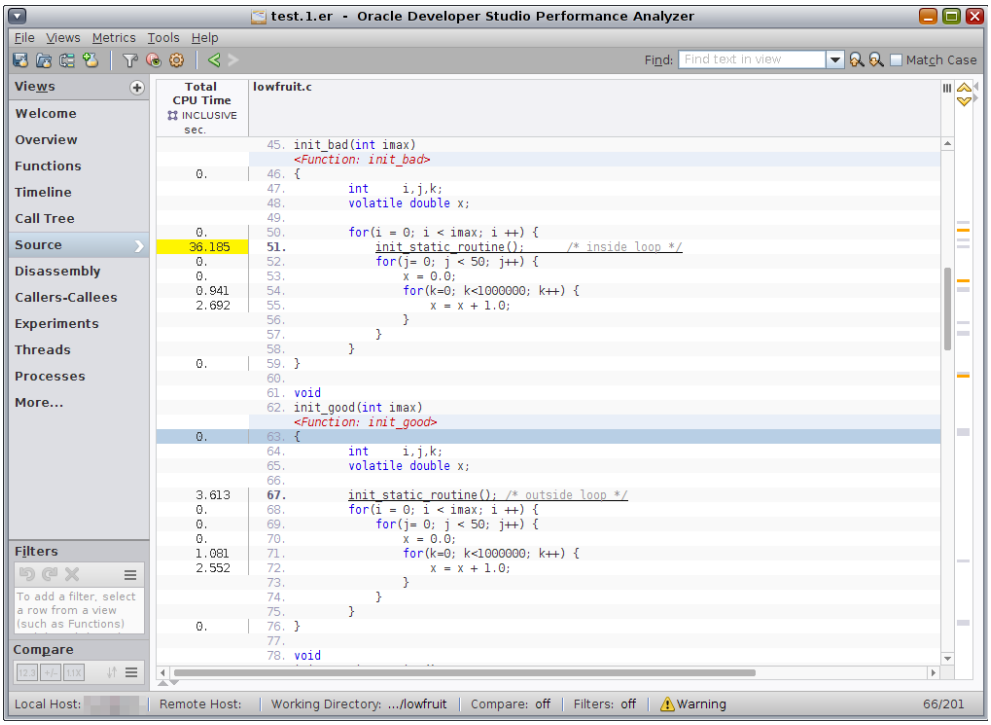
   2 つの関数では、「排他的 CPU 合計時間」はほとんど同じですが、包括的時間はたいへん異なることがわかります。`init_bad()` 関数は、呼び出し先で費やす時間が原因で遅くなります。両方の関数が同じ呼び出し先を呼び出しますが、そのルーチンで費やす時間には大差があります。2 つのルーチンのソースを検査することで理由を確認できます。

5. 関数 `init_good()` を選択してから、「ソース」ビューをクリックするか、メニューバーから「ビュー」->「ソース」を選択します。

6. ウィンドウを調整して、コードの領域をさらに確保します。上マージンの下矢印をクリックして「呼び出し元/呼び出し回数」パネルを縮小して、横マージンの右矢印をクリックして「選択の詳細」パネルを縮小します。

注記 - チュートリアルの残りの部分では、必要に応じてこれらのパネルの展開や縮小を行なってください。

`init_bad()` と `init_good()` の両方のソースを表示するには、少し上へスクロールしてください。「ソース」ビューは次のスクリーンショットのようになるはずです。



`init_static_routine()` への呼び出しは `init_good()` のループの外部で行われるのに対して、`init_good()` にはループ内の `init_static_routine()` への呼び出しがあります。正しくないバージョンでは、正しいバージョンより約 10 倍長く (ループカウントに対応) かかります。

この例は、表示されているほどくだらないものではありません。これは、表の行ごとにアイコンが示された表を生成する実際のコードに基づいています。この例では初期化をループ内で行うべきではないことが簡単にわかりますが、実際のコードでは、初期化はライブラリルーチンに組み込まれており、明らかではありませんでした。

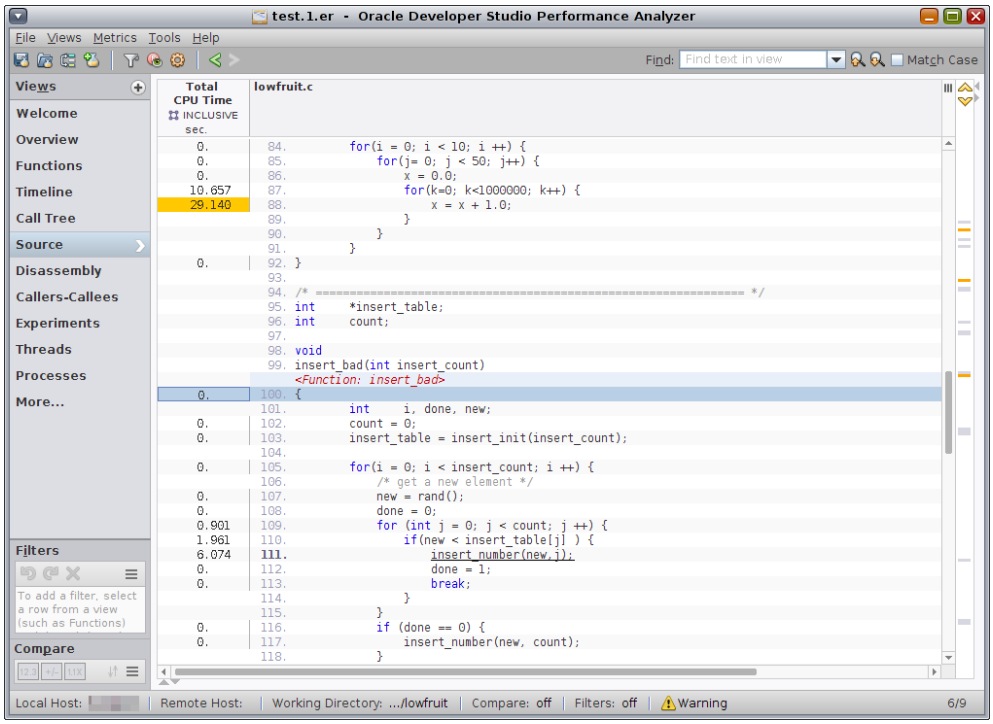
そのコードを実装するために使用されたツールキットでは、2 つのライブラリ呼び出し (API) を使用できました。最初の API では、アイコンが表の行に追加され、2 番目の API では、アイコンのベクトルが表全体に追加されました。最初の API を使用したコーディングの方が簡単ですが、アイコンが追加されるたびに、ツールキットでは、表全体の正しい値を設定するためにすべての行の高さが再計算されました。コードで代替の API を使用して一度にすべてのアイコンを追加したときに、高さの再計算は一度のみ行われました。

7. 次に、「関数」ビューに戻って、2 つのバージョンの挿入タスク `insert_bad()` と `insert_good()` を調べます。

   `insert_bad()` の「排他的 CPU 合計時間」は莫大ですが、`insert_good()` ではごくわずかです。各エントリをリストに挿入するために呼び出される関数 `insert_number()` での時

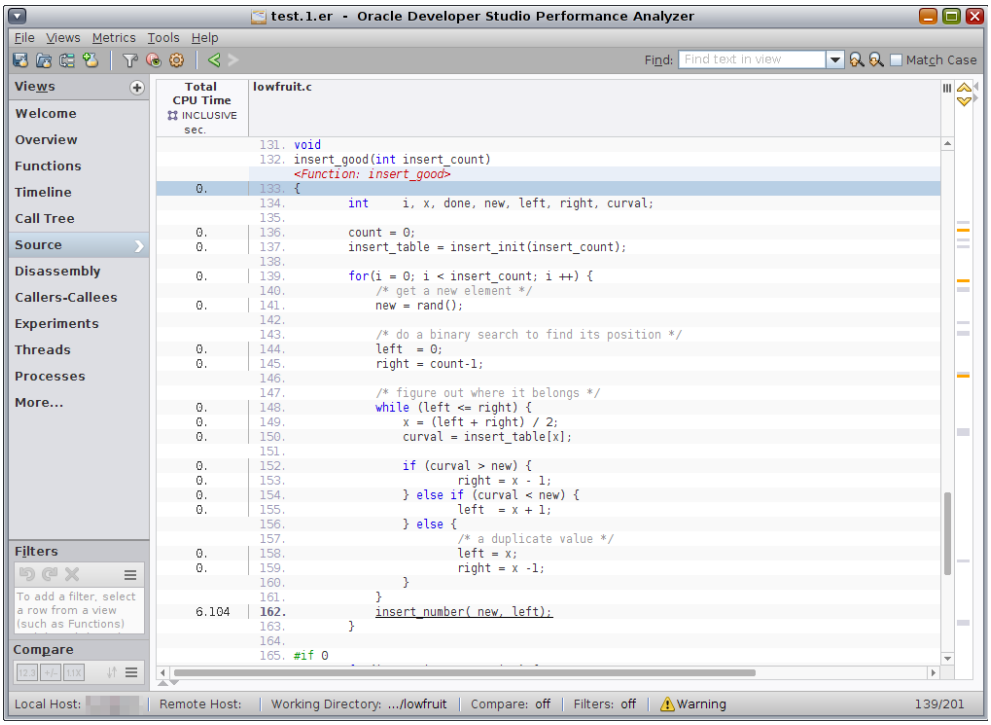
間を表す、各バージョンの包括的時間と排他的時間の差は同じです。ソースを検査することで理由を確認できます。

8. insert_bad() を選択して、「ソース」ビューに切り替えます。



insert_number() への呼び出しを除き、時間は、新しい数値を挿入するために適した場所のリニア検索で表示されるループで使用されます。

9. 次に、下へスクロールして insert_good() を確認します。



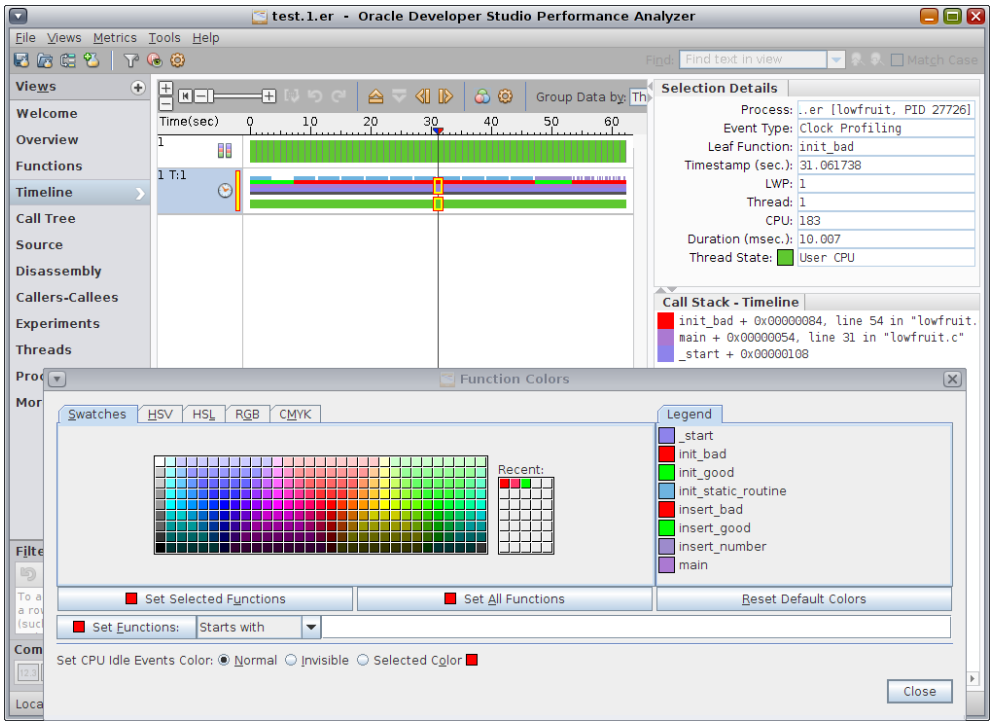
コードは、挿入に適した場所を見つけるためにバイナリ検索を行なっているためより複雑ですが、`insert_number()` への呼び出しを除き、費やされる合計時間は `insert_bad()` での時間より大幅に少なくなります。この例は、バイナリ検索はリニア検索より効率的である可能性があることを示しています。

「タイムライン」ビューでは、ルーチンでの違いをグラフィカルに確認することもできます。

10. 「タイムライン」ビューをクリックするか、メニューバーから「ビュー」->「タイムライン」を選択します。

    プロファイリングデータは、すべてのスレッドのプロファイリングクロックのティックごとに 1 つ、一連のイベントとして記録されます。「タイムライン」ビューには、それぞれの個別のイベントが、そのイベントで記録された呼び出しスタックとともに表示されます。呼び出しスタックは、呼び出しスタック内のフレームのリストとして表示され、上部にはリーフ PC (イベントの時点で実行する次の指示)、次にこれを呼び出す呼び出し側などが表示されます。プログラムのメインスレッドでは、呼び出しスタックの上部は常に `_start` です。

11. 「タイムライン」ツールバーで、関数に色を付けるための「呼び出しスタック関数の色」アイコン  をクリックするか、メニューバーから「ツール」->「関数の色」を選択すると、次に示すダイアログボックスが表示されます。



関数の色が変更され、スクリーンショットで正しいバージョンの関数と正しくないバージョンの関数がより明確に区別されます。`init_bad()` 関数と `insert_bad()` 関数は両方とも赤色で、`init_good()` と `insert_good()` は両方とも明るい緑です。

12. 「タイムライン」ビューをより似た外観にするには、「関数の色」ダイアログボックスで次を行います。
    - 「凡例」でメソッドのリストを下へスクロールして、`init_bad()` メソッドを見つけます。
    - `init_bad()` メソッドを選択して、「見本」で赤色の正方形をクリックし、「選択した関数の設定」ボタンをクリックします。
    - `insert_bad()` メソッドを選択して、「見本」で赤色の正方形をクリックし、「選択した関数の設定」ボタンをクリックします。
    - `init_good()` メソッドを選択して、「見本」で緑色の正方形をクリックし、「選択した関数の設定」ボタンをクリックします。
    - `insert_good()` メソッドを選択して、「見本」で緑色の正方形をクリックし、「選択した関数の設定」ボタンをクリックします。
13. 「タイムライン」のいちばん上のバーを確認します。

    マウスカーソルを最初の列の上に移動するとツールチップで確認できるように、「タイムライン」のいちばん上のバーは「CPU 使用率標本」バーです。「CPU 使用率標本」バーの各セ

グメントは、その秒の実行の間のターゲットのリソース使用率を示す 1 秒の間隔を表します。

この例では、すべての間隔が「ユーザー CPU 時間」の集計に費やされたため、すべてのセグメントは緑です。マイクロステートへの色のマッピングはスクリーンショットに示されていませんが、「選択の詳細」ウィンドウにはこれが示されます。

14. 「タイムライン」の 2 番目のバーを確認します。

    2 番目のバーはクロックプロファイリング呼び出しスタックのバーで、「1 T:1」というラベルが付いており、プロセス 1 と、この例での唯一のスレッドであるスレッド 1 を意味します。「クロックプロファイリング呼び出しスタック」バーには、プログラムの実行中に発生するイベントのデータの 2 つのバーが示されます。上部のバーは、呼び出しスタックの色分けされた表現を示し、下部のバーは、各イベントでのスレッドの状態を示します。この例の状態は常に「ユーザー CPU 時間」であるため、緑で塗りつぶされた線で表示されます。

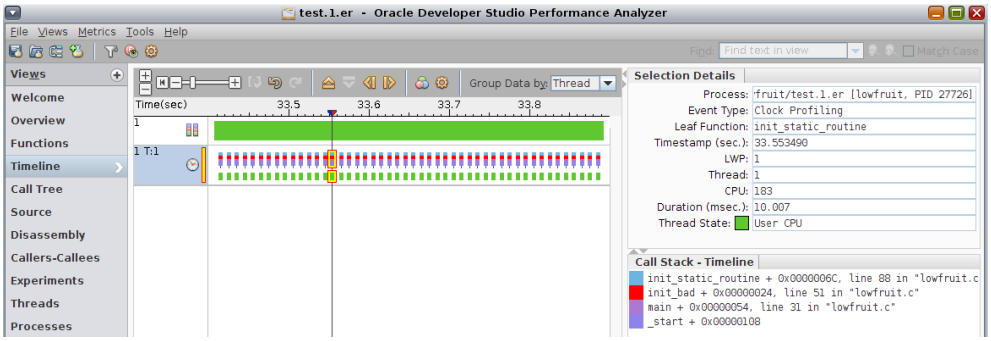
    「クロックプロファイリング呼び出しスタック」バー内の任意の場所でクリックして、もっとも近いイベントを選択すると、そのイベントの詳細が「選択の詳細」ウィンドウに表示されます。呼び出しスタックのパターンから、スクリーンショットでは明るい緑で示されている `init_good()` ルーチンと `insert_good()` ルーチンでの時間は、赤色で示されている `init_bad()` ルーチンと `insert_bad()` ルーチンでの対応する時間よりも大幅に短いことがわかります。

15. 「タイムライン」で正しいルーチンと正しくないルーチンに対応する領域でイベントを選択して、「選択の詳細」ウィンドウの下にある「呼び出しスタック - タイムライン」ウィンドウで呼び出しスタックを確認します。

    「呼び出しスタック」ウィンドウで任意のフレームを選択してから、「ビュー」ナビゲーションバーの「ソース」ビューを選択して、そのソース行のソースに移動します。また、呼び出しスタックでフレームをダブルクリックして「ソース」ビューに移動することも、呼び出しスタックでフレームを右クリックしてポップアップメニューから選択することもできます。

16. 「タイムライン」の上部にあるスライダを使用するか、+ キーを使用するか、またはマウスでダブルクリックすることで、イベントでズームインします。

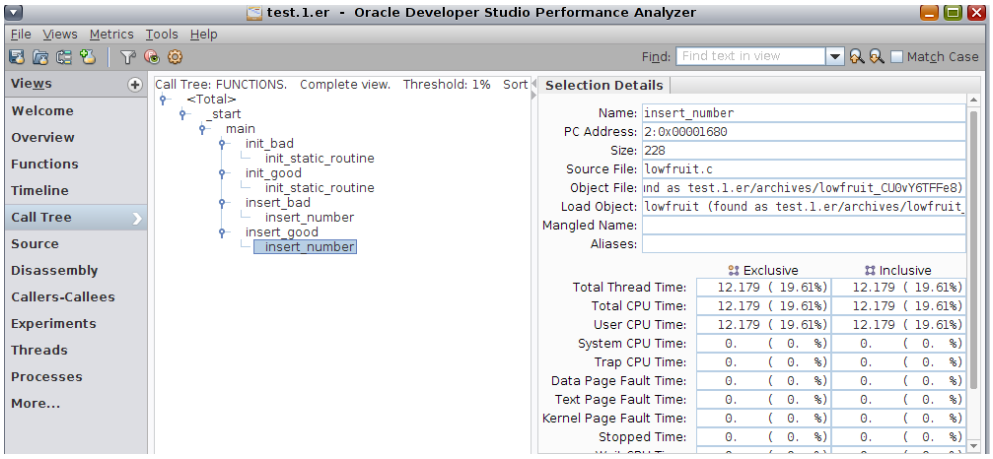
    十分にズームインすると、表示されるデータは連続していなくても、プロファイルティック (この例では約 10 ミリ秒) ごとに 1 つの個別のイベントで構成されていることがわかります。



「タイムライン」ビューに関する詳細情報のヘルプを表示するには、F1 キーを押します。

17. 「呼び出しツリー」ビューをクリックするか、「ビュー」->「呼び出しツリー」を選択してプログラムの構造を表示します。

    「呼び出しツリー」ビューにはプログラムの動的呼び出しグラフが表示され、「選択の詳細」パネルにパフォーマンス情報が表示されます。



パフォーマンスアナライザには、プログラム構造内をナビゲートできる「呼び出し元-呼び出し先」ビューや、記録した実験の詳細が表示される「実験」ビューなどのデータの追加のビューが多数あります。この単純な例では、「スレッド」ビューと「プロセッサ」ビューはあまり興味深いものではありません。

「ビュー」リストで + ボタンをクリックすると、ナビゲーションバーにほかのビューを追加できます。アセンブリ言語のプログラマは、「逆アセンブリ」を調べることもできます。ほかのビューを調べてみてください。

パフォーマンスアナライザは、非常に強力なフィルタリング機能も備えています。時間、スレッド、関数、ソース行、命令、呼び出しスタック断片、およびこれらの任意の組み合わせでフィルタできます。サンプルコードは非常に単純でフィルタリングは必要ないため、フィルタリングの使用はこのチュートリアルの範囲外です。

Java プロファイリングの概要

この章では、次の内容について説明します。

Java プロファイリングチュートリアルについて

このチュートリアルでは、Oracle Developer Studio パフォーマンスアナライザを使用したプロファイリングのもっとも簡単な例を示して、パフォーマンスアナライザを使用してパフォーマンスの実験を収集して検査する方法について説明します。このチュートリアルでは、「概要」、「関数」ビュー、「ソース」ビュー、「タイムライン」ビュー、および「呼び出しツリー」ビューを使用します。

プログラム `jlowfruit` は、2 つの異なるタスク (ループで初期化するためのタスクと、順序付きリストに数値を挿入するためのタスク) を実行する単純なプログラムです。それぞれのタスクは、非効率的な方法とより効率的な方法の 2 回実行されます。

ヒント - 「C プロファイリングの概要」のチュートリアルでは、同等の C プログラムを使用して、パフォーマンスアナライザを使用した類似のアクティビティーを示します。

記録する実験で目にするデータは、ここで示されているものとは異なります。チュートリアルのスクリーンショットに使用される実験は、Oracle Solaris 11.3 が実行されている SPARC T5 システムで記録されました。Oracle Solaris または Linux を実行する x86 システムからのデータは異なります。さらに、データ収集には統計的な性質があり、同じシステムおよび OS で実行する場合でも実験ごとに異なります。

パフォーマンスアナライザの実際のウィンドウ構成は、スクリーンショットと厳密に一致しない場合があります。パフォーマンスアナライザでは、ウィンドウのコンポーネント間の区切りバーをドラッグしたり、コンポーネントを縮小したり、ウィンドウのサイズを変更したりできます。パフォーマンスアナライザはその構成を記録し、次回の実行時に同じ構成を使用します。チュートリアルで示しているスクリーンショットの取得過程で、多くの構成を変更しました。

このチュートリアルは、Oracle Developer Studio がインストールされているシステムでローカルに実行されます。107 ページの「リモートパフォーマンスアナライザの使用」の説明に従って、リモートで実行することもできます。

jlowfruit サンプルコードの設定

始める前に:

コードの取得および環境の設定については、次を参照してください。

- 10 ページの「チュートリアルのサンプルコードの入手」
- 11 ページの「環境をチュートリアル用に設定」

1. 次のコマンドを使用して、jlowfruit ディレクトリの内容を独自のプライベート作業領域にコピーします。

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

   mydirectory は使用する作業ディレクトリです。
2. その作業ディレクトリのコピーを変更します。

   ```
   % cd directory/jlowfruit
   ```
3. ターゲットの実行可能ファイルをビルドします。

   ```
   % make clobber

   % make
   ```

   注記 - clobber サブコマンドコマンドは、このディレクトリで以前に make を実行した場合にのみ必要ですが、どの場合にも安全に使用できます。

   ---

   make の実行後に、このディレクトリには、チュートリアルで使用するターゲットアプリケーションと、jlowfruit.class という名前の Java クラスファイルが含められます。

ヒント - サンプルのコンパイルで問題が生じた場合は、次のコマンドを使用して javac のバージョンを確認します。

```
% javac -version
```

出力に少なくとも javac 1.7 が報告されない場合は、PATH を JDK 7 以上に更新する必要があります。

次のセクションでは、パフォーマンスアナライザを使用して jlowfruit プログラムからデータを収集して、実験を作成する方法を示します。

パフォーマンスアナライザを使用した jlowfruit からのデータの収集

このセクションでは、パフォーマンスアナライザの「アプリケーションのプロファイル」機能を使用して、Java アプリケーションで実験のデータを収集する方法について説明します。

ヒント - パフォーマンスアナライザからアプリケーションをプロファイルする方法を確認するためにこれらの手順に従わない方が望ましい場合は、jlowfruit の Makefile に含まれている make ターゲットを使用して実験を記録できます。

```
% make collect
```

collect ターゲットは collect コマンドを起動して、このセクションでパフォーマンスアナライザを使用して作成するものと同じような実験を記録します。その後、34 ページの「パフォーマンスアナライザを使用した jlowfruit データの検査」にスキップできます。

---

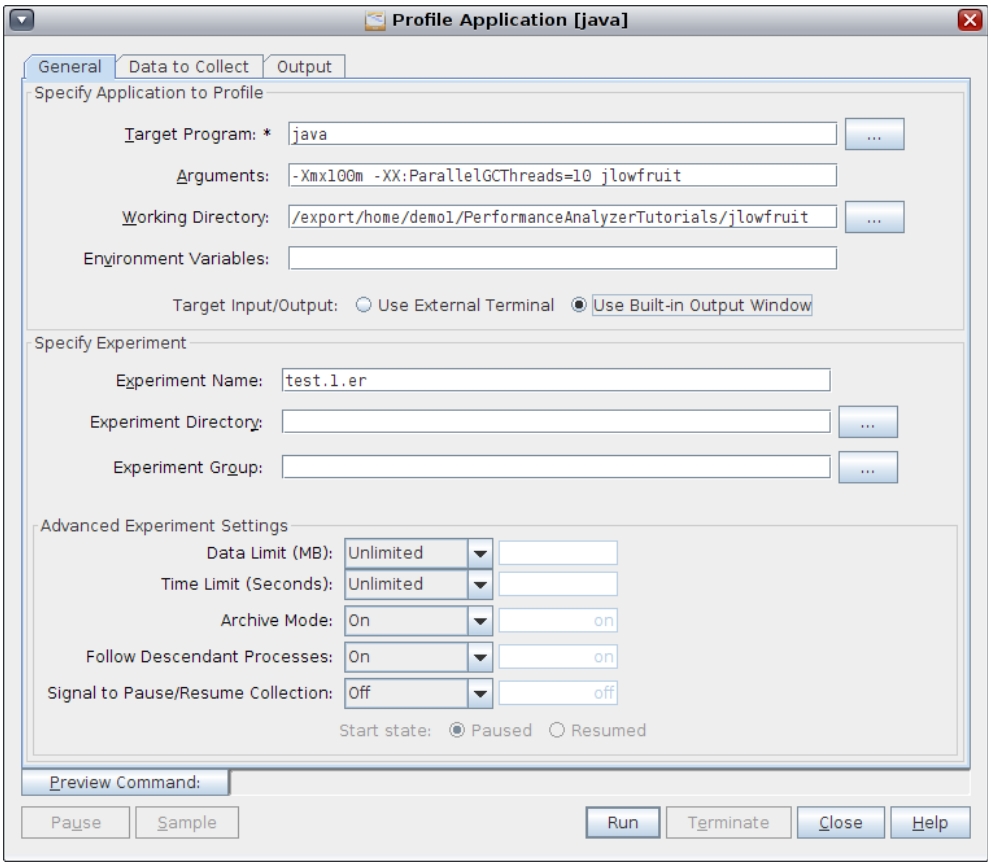
1. jlowfruit ディレクトリを開いているときに、ターゲット java とその引数を使用してパフォーマンスアナライザを起動します。

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

   「一般」タブが選択され、analyzer コマンドを使用して指定した情報を使用していくつかのオプションがすでに入力された「アプリケーションのプロファイル」ダイアログボックスが開きます。

   「ターゲットプログラム」は「java」に設定され、「引数」は次のように設定されます

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



2. 「ターゲット入力/出力」オプションでは、「組み込み出力ウィンドウの使用」を選択します。

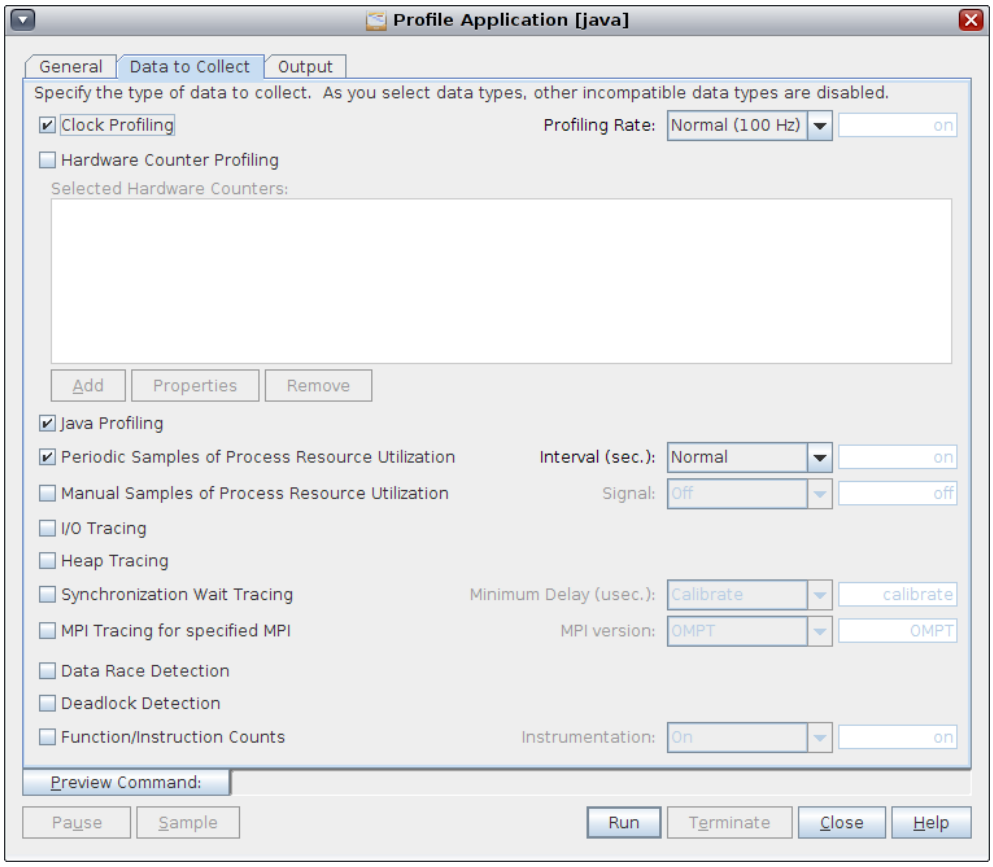
   「ターゲット入力/出力」オプションは、ターゲットプログラム `stdout` および `stderr` がリダイレクトされるウィンドウを指定します。デフォルト値は「外部端末の使用」ですが、このチュートリアルでは、「パフォーマンスアナライザ」ウィンドウのすべてのアクティビティーを維持するために、「ターゲット入力/出力」オプションを「組み込み出力ウィンドウの使用」に変更してください。このオプションを使用すると、「アプリケーションのプロファイル」ダイアログボックスの「出力」タブには「`stdout`」と「`stderr`」が表示されます。

   リモートで実行している場合、組み込み出力ウィンドウのみがサポートされるため、「ターゲット入力/出力」オプションはありません。

3. 「実験名」オプションでは、デフォルトの実験名は `test.1.er` ですが、名前が `.er` で終わりまだ使用中ではないかぎり、別の名前に変更できます。

4. 「収集するデータ」タブをクリックします。

   「収集するデータ」タブでは、収集するデータのタイプを選択して、すでに選択されているデフォルトを表示できます。



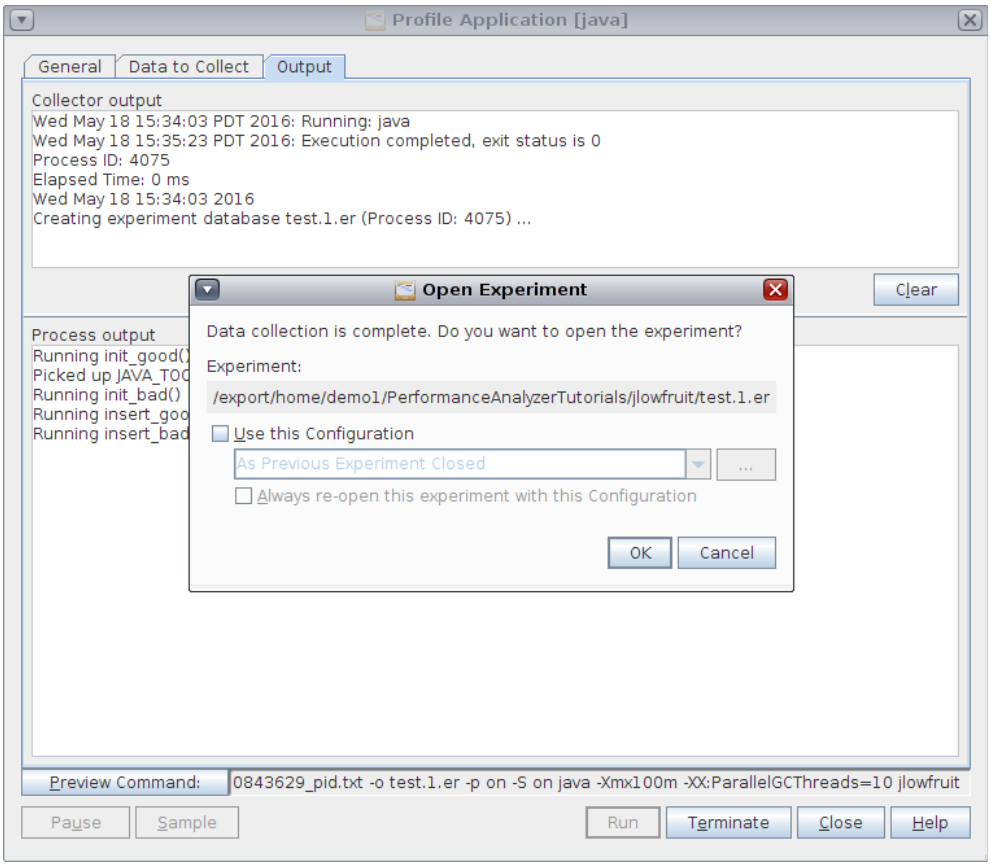
スクリーンショットで確認できるように、「Java プロファイリング」はデフォルトで有効になっています。

オプションで、「プレビューコマンド」ボタンをクリックして、プロファイリングの開始時に実行する `collect` コマンドを表示できます。

5. 「実行」ボタンをクリックします。

「アプリケーションのプロファイル」ダイアログボックスには「出力」タブが表示され、プログラムの実行時に「プロセス出力」パネルでプログラム出力が示されます。

プログラムの完了後に、記録した実験を開くかどうかがダイアログボックスで尋ねられます。



6. ダイアログボックスで「了解」をクリックします。
   実験が開きます。次のセクションでは、データの検査方法を示します。

パフォーマンスアナライザを使用した jlowfruit データの検査

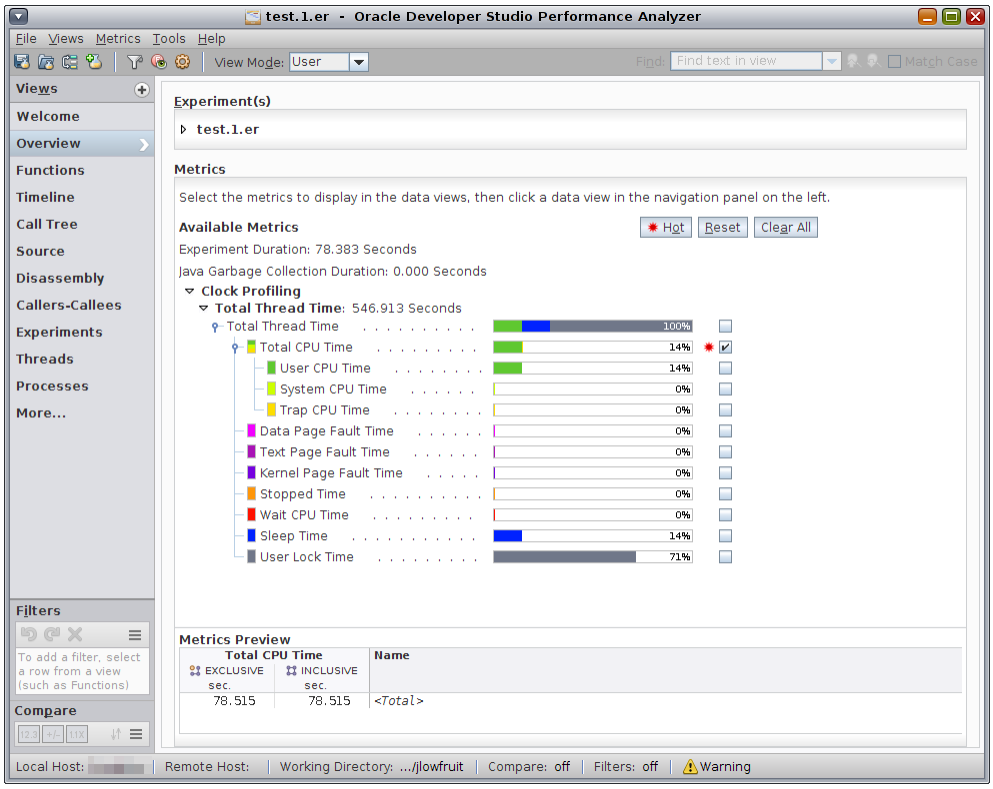
このセクションでは、jlowfruit サンプルコードから作成された実験のデータを調べる方法を示します。

1. 前のセクションで作成した実験がまだ開いていない場合は、jlowfruit ディレクトリからパフォーマンスアナライザを起動して、次のようにして実験をロードできます。

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

   実験が開くと、パフォーマンスアナライザの「概要」ページが表示されます。

2. 「概要」ページにはメトリック値のサマリーが表示され、メトリックを選択できます。



この実験では、「概要」には、すべてが「ユーザー CPU 時間」だった約 14% の「CPU 時間合計」と、約 14% の「休眠時間」と 71% の「ユーザーロック時間」が示されます。ユーザー Java コード `jlowfruit` はシングルスレッドであり、その 1 つのスレッドは CPU の制約を受けますが、すべての Java プログラムで多数のシステムスレッドを含む複数のスレッドが使用されます。これらのスレッドの数は、ガベージコレクタのパラメータや、プログラムが実行されたマシンのサイズなどの JVM オプションの選択によって異なります。

実験は Oracle Solaris システムに記録され、「概要」には記録されたメトリックが 12 個表示されますが、「CPU 時間合計」のみがデフォルトで有効になっています。

色付きのインジケータがあるメトリックは、Oracle Solaris によって定義された 10 個のマイクロステートで費やされた時間です。これらのメトリックには、さまざまな待ち時間のほかに、一緒にすると「CPU 時間合計」と等しくなる「ユーザー CPU 時間」、「システム CPU 時間」、および「トラップ CPU 時間」が含まれます。「スレッド合計時間」は、すべてのマイクロステートの合計です。

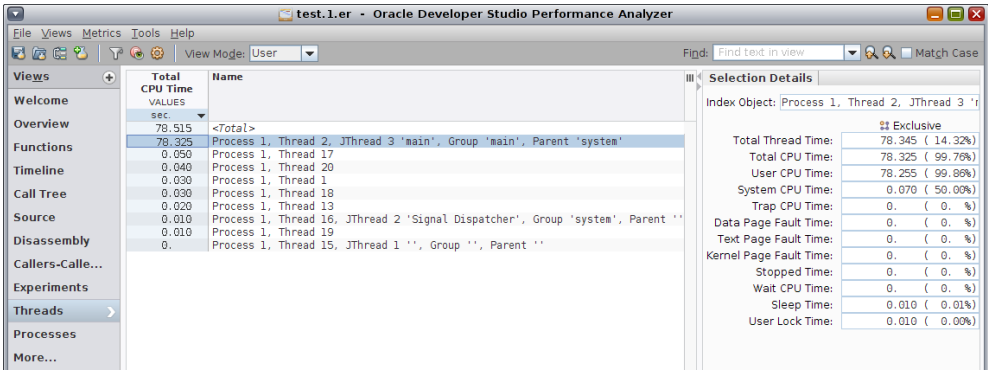
Linux ではマイクロステートアカウンティングがサポートされないため、Linux マシンでは「CPU 時間合計」のみが記録されます。

デフォルトでは、「包括的 CPU 合計時間」と「排他的 CPU 合計時間」の両方のプレビューが表示されます。メトリックの*包括的*とは、そのメトリックが呼び出すすべての関数またはメソッドで集計されたメソッドを含む、その関数またはメソッドのメトリック値を指します。*排他的*とは、その関数またはメソッド内で集計されたメトリックのみを指します。

3. 「ホット」ボタンをクリックして、高い値を持つメトリックを選択してデータビューでそれらのメトリックを表示します。

   下部にある「メトリックのプレビュー」パネルが更新され、テーブルフォーマットのデータが事前に設定されたデータビューでのメトリックの表示方法が示されます。次に、どのスレッドがどのメトリックを処理するかを確認します。

4. 「表示」ナビゲーションパネルで「スレッド」ビューの名前をクリックするか、メニューバーから「表示」->「スレッド」を選択することで、「スレッド」ビューに切り替えます。



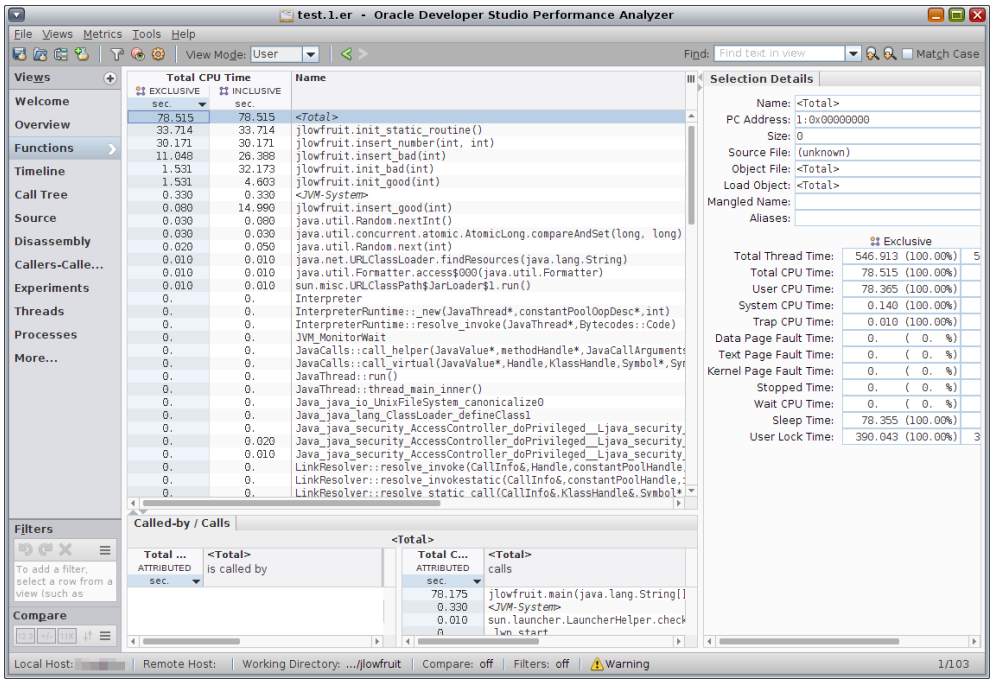
ほとんどすべての「CPU 時間合計」を使用したスレッドは、この単純なアプリケーションでの唯一のユーザー Java スレッドであるスレッド 2 です。

スレッド 15 は、実際には JVM によって内部で作成されるにもかかわらず、ほとんどの場合ユーザースレッドです。これは、起動中にのみアクティブで、ほとんどの時間が集計されません。実験で、スレッド 15 と似た 2 番目のスレッドが作成されることがあります。

スレッド 1 はその間中スリープ状態で費やします。

残りのスレッドは、JVM がそれ自体を内部で同期する方法であるロックの待機にその時間を使用します。これらのスレッドには、HotSpot のコンパイルおよびガベージコレクションに使用されるスレッドが含まれています。このチュートリアルでは、JVM システムの動作を検査しませんが、これは別のチュートリアルである「Java および Java-C++ 混合プロファイリング」で検査されます。

5. 「表示」ナビゲーションパネルで「関数」ビューをクリックするか、メニューバーから「表示」->「関数」を選択します。



「関数」ビューには、アプリケーション内の関数のリストがそれぞれの関数のパフォーマンスメトリックとともに表示されます。リストは、最初は各関数で使用された「排他的 CPU 合計時間」でソートされます。「関数」ビューには JVM の関数も多数ありますが、これらは比較的低いメトリックです。リストには、ターゲットアプリケーションのすべての関数とプログラムで使用するすべての共有オブジェクトが含まれています。最上位の関数 (もっともコストが高い関数) がデフォルトで選択されています。

右側の「選択の詳細」ウィンドウには、選択した関数について記録されたすべてのメトリックが表示されます。

関数リストの下にある「呼び出し元/呼び出し回数」パネルは、選択した関数に関する詳細情報を提供し、2 つのリストに分割されています。「呼び出し元」リストには選択した関数の呼び出し元が表示され、メトリック値は、その呼び出し元への関数の合計メトリックの属性を示しています。「呼び出し回数」リストは、選択した関数の呼び出し先を示し、呼び出し先の包括的メトリックが選択した関数の合計メトリックにどのように関係したかを示しています。「呼び出し元/呼び出し回数」パネルのいずれかのリストで関数をダブルクリックすると、その関数が、メインの「関数」ビューで選択した関数になります。

6. さまざまな関数を選択してみて、選択の変更によって「関数」ビューの「呼び出し元/呼び出し回数」パネルおよび「選択の詳細」ウィンドウがどのように更新されるかを確認します。

   「選択の詳細」ウィンドウには、「ロードオブジェクト」フィールドに示されているように、ほとんどの関数が `jlowfruit.class` から生成されることが示されます。

また、列ヘッダーをクリックしてみて、ソートを「排他的 CPU 合計時間」から「包括的 CPU 合計時間」に変更したり、「名前」でソートに変更したりすることもできます。

7. 「関数」ビューで、2 つのバージョンの初期化タスク `jlowfruit.init_bad()` と `jlowfruit.init_good()` を比較します。

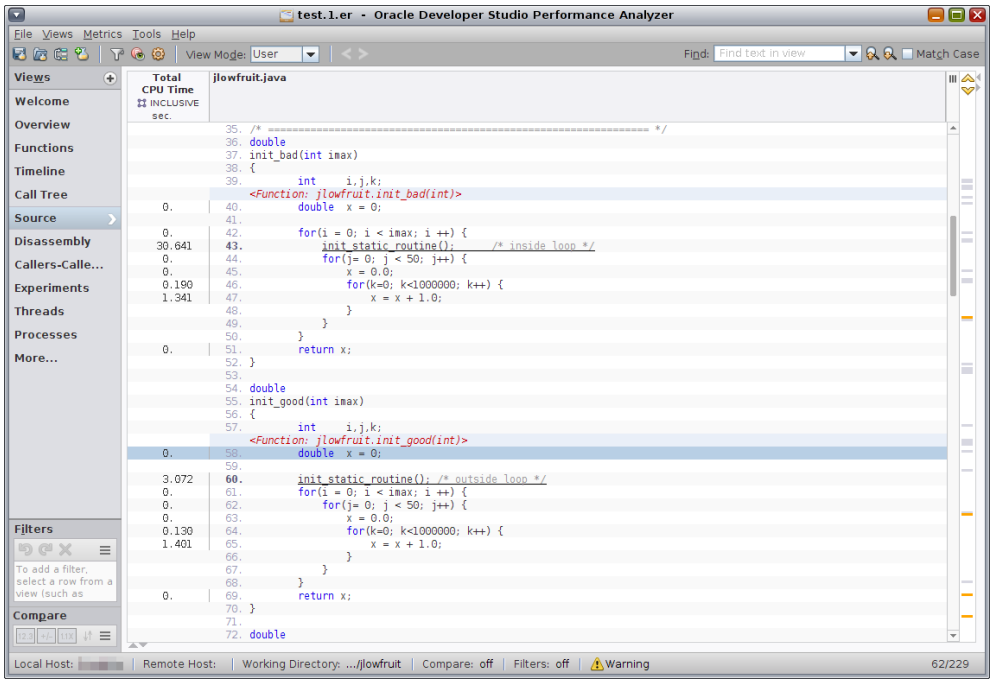
   2 つの関数では、「排他的 CPU 合計時間」はほとんど同じですが、包括的時間はたいへん異なることがわかります。`jlowfruit.init_bad()` 関数は、呼び出し先で費やす時間が原因で遅くなります。両方の関数が同じ呼び出し先を呼び出しますが、そのルーチンで費やす時間には大差があります。2 つのルーチンのソースを検査することで理由を確認できます。

8. 関数 `jlowfruit.init_good()` を選択してから、「ソース」ビューをクリックするか、メニューバーから「ビュー」->「ソース」を選択します。

9. ウィンドウを調整して、コードの領域をさらに確保します。上マージンの下矢印をクリックして「呼び出し元/呼び出し回数」パネルを縮小して、横マージンの右矢印をクリックして「選択の詳細」パネルを縮小します。

注記 - チュートリアルの残りの部分では、必要に応じてこれらのパネルの展開や縮小を行なってください。

`jlowfruit.init_bad()` と `jlowfruit.init_good()` の両方のソースを表示するには、少し上へスクロールしてください。「ソース」ビューは次のスクリーンショットのようになるはずです。



jlowfruit.init_static_routine() への呼び出しは jlowfruit.init_good() のループの外部で行われるのに対して、jlowfruit.init_bad() にはループ内の jlowfruit.init_static_routine() への呼び出しがあります。正しくないバージョンでは、正しいバージョンより約 10 倍長く (ループカウントに対応) かかります。

この例は、表示されているほどくだらないものではありません。これは、表の行ごとにアイコンが示された表を生成する実際のコードに基づいています。この例では初期化をループ内で行うべきではないことが簡単にわかりますが、実際のコードでは、初期化はライブラリルーチンに組み込まれており、明らかではありませんでした。

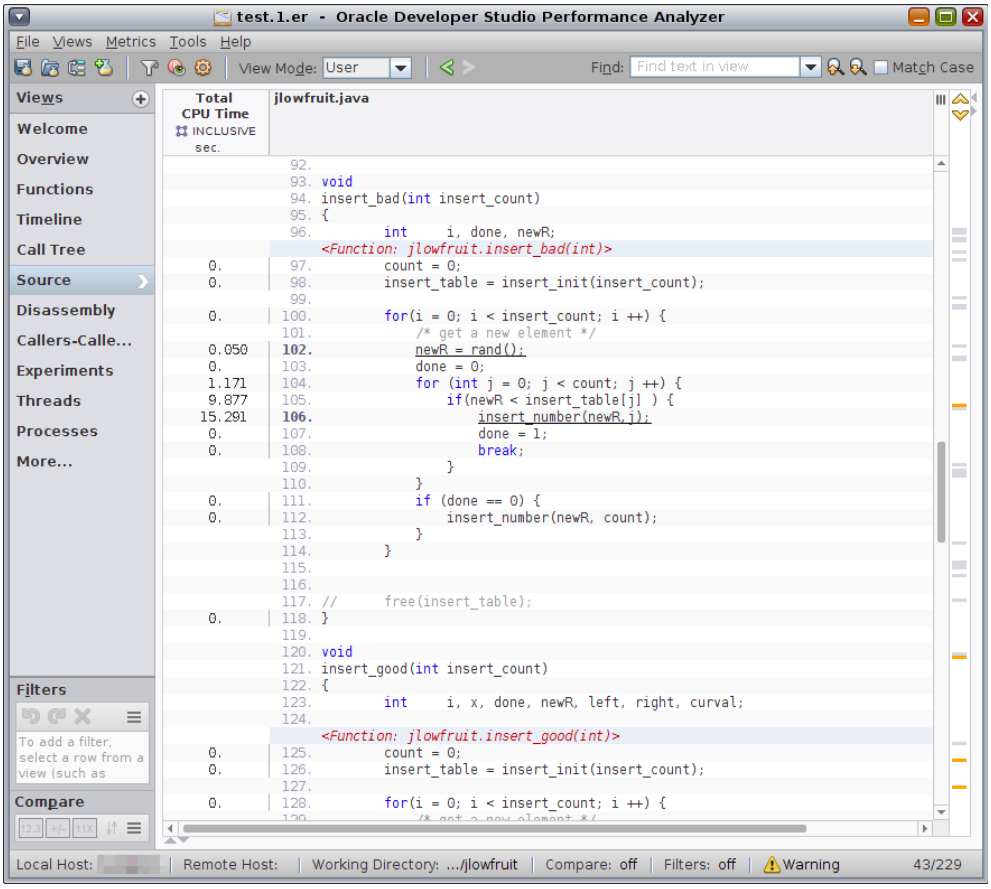
そのコードを実装するために使用されたツールキットでは、2 つのライブラリ呼び出し (API) を使用できました。最初の API では、アイコンが表の行に追加され、2 番目の API では、アイコンのベクトルが表全体に追加されました。最初の API を使用したコーディングの方が簡単ですが、アイコンが追加されるたびに、ツールキットでは、表全体の正しい値を設定するためにすべての行の高さが再計算されました。コードで代替の API を使用して一度にすべてのアイコンを追加したときに、高さの再計算は一度のみ行われました。

10. 次に、「関数」ビューに戻って、2 つのバージョンの挿入タスク jlowfruit.insert_bad() と jlowfruit.insert_good() を調べます。

    jlowfruit.insert_bad() の「排他的 CPU 合計時間」は莫大ですが、jlowfruit.insert_good() ではごくわずかです。各エントリをリストに挿入するために呼び出される関

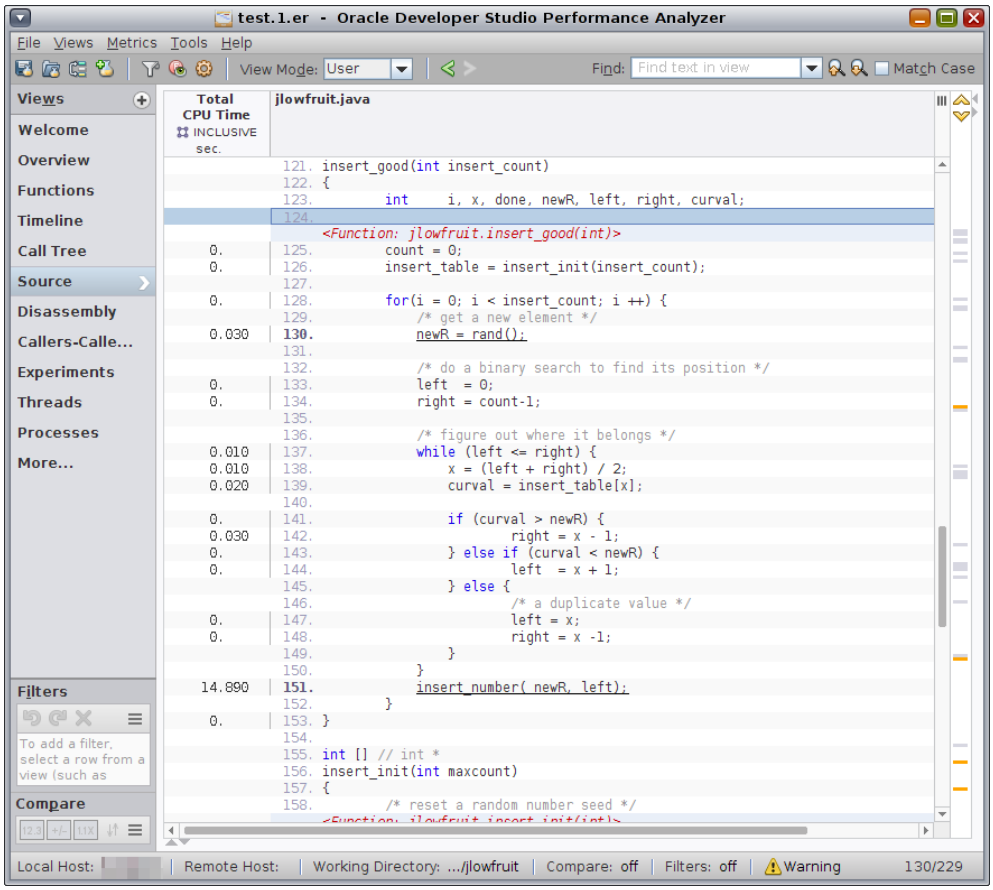
数 jlowfruit.insert_number() での時間を表す、各バージョンの包括的時間と排他的時間の差は同じです。ソースを検査することで理由を確認できます。

11. jlowfruit.insert_bad() を選択して、「ソース」ビューに切り替えます。



jlowfruit.insert_number() への呼び出しを除き、時間は、新しい数値を挿入するために適した場所のリニア検索で表示されるループで使用されます。

12. 次に、下へスクロールして jlowfruit.insert_good() を確認します。



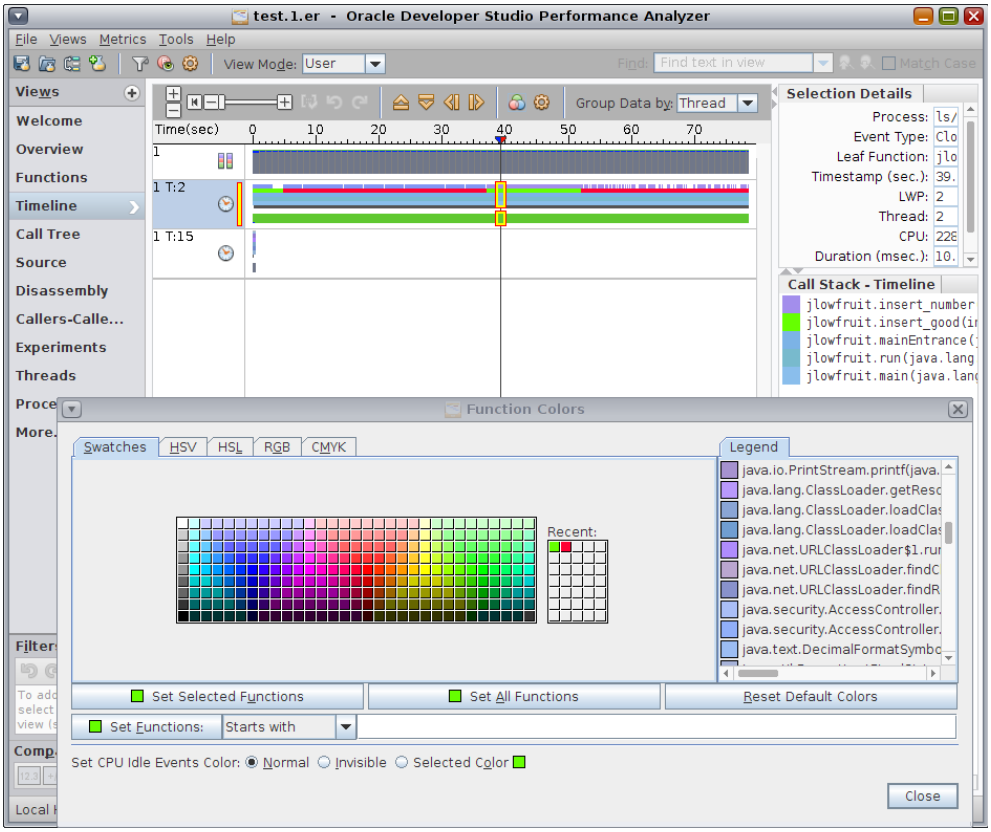
コードは、挿入に適した場所を見つけるためにバイナリ検索を行なっているためより複雑ですが、`jlowfruit.insert_number()` への呼び出しを除き、費やされる合計時間は `jlowfruit.insert_bad()` での時間より大幅に少なくなります。この例は、バイナリ検索はリニア検索より効率的である可能性があることを示しています。

「タイムライン」ビューでは、ルーチンでの違いをグラフィカルに確認することもできます。

13. 「タイムライン」ビューをクリックするか、メニューバーから「ビュー」->「タイムライン」を選択します。

プロファイリングデータは、すべてのスレッドのプロファイリングクロックのティックごとに 1 つ、一連のイベントとして記録されます。「タイムライン」ビューには、それぞれの個別のイベントが、そのイベントで記録された呼び出しスタックとともに表示されます。呼び出しスタックは、呼び出しスタック内のフレームのリストとして表示され、上部にはリーフ PC (イベントの時点で実行する次の指示)、次にこれを呼び出す呼び出し側などが表示されます。プログラムのメインスレッドでは、呼び出しスタックの最上段は常に `main` です。

14. 「タイムライン」ツールバーで、関数に色を付けるための「呼び出しスタック関数の色」アイコン をクリックするか、メニューバーから「ツール」->「関数の色」を選択すると、次に示すダイアログボックスが表示されます。



関数の色が変更され、スクリーンショットで正しいバージョンの関数と正しくないバージョンの関数がより明確に区別されます。`jlowfruit.init_bad()` 関数と `jlowfruit.insert_bad()` 関数は両方とも赤色で、`jlowfruit.init_good()` と `jlowfruit.insert_good()` は両方とも明るい緑です。

15. 「タイムライン」ビューをより似た外観にするには、「関数の色」ダイアログボックスで次を行います。
    - 「凡例」で java メソッドのリストを下へスクロールして、`jlowfruit.init_bad()` メソッドを見つけます。
    - `jlowfruit.init_bad()` メソッドを選択して、「見本」で赤色の正方形をクリックし、「選択した関数の設定」ボタンをクリックします。

- jlowfruit.insert_bad() メソッドを選択して、「見本」で赤色の正方形をクリックし、「選択した関数の設定」ボタンをクリックします。
- jlowfruit.init_good() メソッドを選択して、「見本」で緑色の正方形をクリックし、「選択した関数の設定」ボタンをクリックします。
- jlowfruit.insert_good() メソッドを選択して、「見本」で緑色の正方形をクリックし、「選択した関数の設定」ボタンをクリックします。

16. 「タイムライン」のいちばん上のバーを確認します。

    マウスカーソルを最初の列の上に移動するとツールチップで確認できるように、「タイムライン」のいちばん上のバーは「CPU 使用率標本」バーです。「CPU 使用率標本」バーの各セグメントは、その秒の実行の間のターゲットのリソース使用率を示す 1 秒の間隔を表します。

    この例では、セグメントはほとんどが灰色で一部が緑であり、「合計時間」のほんのわずかのみが「ユーザー CPU 時間」の集計に費やされたことを反映しています。マイクロステートへの色のマッピングはスクリーンショットに示されていませんが、「選択の詳細」ウィンドウにはこれが示されます。

17. 「タイムライン」の 2 番目のバーを確認します。

    2 番目のバーはクロックプロファイリング呼び出しスタックのバーで、「1 T:2」というラベルが付いており、プロセス 1 と、この例での主なユーザースレッドであるスレッド 2 を意味します。「クロックプロファイリング呼び出しスタック」バーには、プログラムの実行中に発生するイベントのデータの 2 つのバーが示されます。上部のバーは、呼び出しスタックの色分けされた表現を示し、下部のバーは、各イベントでのスレッドの状態を示します。この例の状態は常に「ユーザー CPU 時間」であるため、緑で塗りつぶされた線で表示されます。

    異なるスレッド番号のラベルが付いた 1 つまたは 2 つの追加のバーが表示されるはずですが、これらのバーには実行の開始時には少数のイベントしかありません。

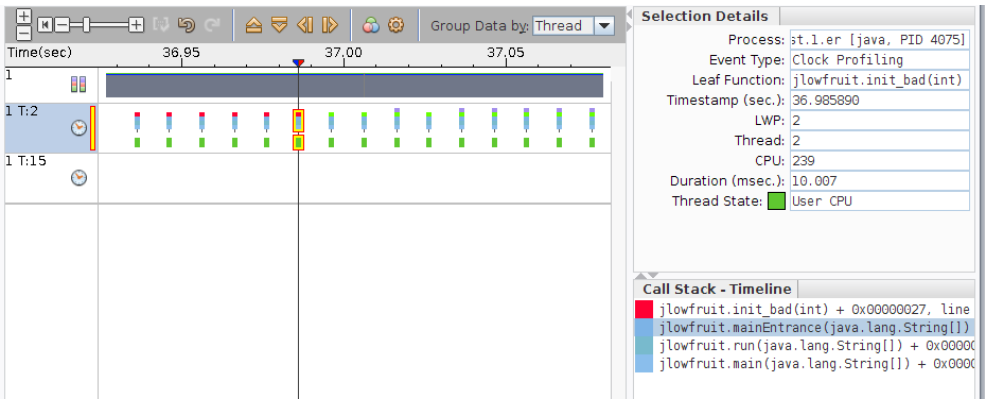
    「クロックプロファイリング呼び出しスタック」バー内の任意の場所でクリックして、もっとも近いイベントを選択すると、そのイベントの詳細が「選択の詳細」ウィンドウに表示されます。呼び出しスタックのパターンから、スクリーンショットでは明るい緑で示されている jlowfruit.init_good() ルーチンと jlowfruit.insert_good() ルーチンでの時間は、赤色で示されている jlowfruit.init_bad() ルーチンと jlowfruit.insert_bad() ルーチンでの対応する時間よりも大幅に短いことがわかります。

18. 「タイムライン」で正しいルーチンと正しくないルーチンに対応する領域でイベントを選択して、「選択の詳細」ウィンドウの下にある「呼び出しスタック - タイムライン」ウィンドウで呼び出しスタックを確認します。

    「呼び出しスタック」ウィンドウで任意のフレームを選択してから、「ビュー」ナビゲーションバーの「ソース」ビューを選択して、そのソース行のソースに移動します。また、呼び出しスタックでフレームをダブルクリックして「ソース」ビューに移動することも、呼び出しスタックでフレームを右クリックしてポップアップメニューから選択することもできます。

19. 「タイムライン」の上部にあるスライダを使用するか、+ キーを使用するか、またはマウスでダブルクリックすることで、イベントでズームインします。

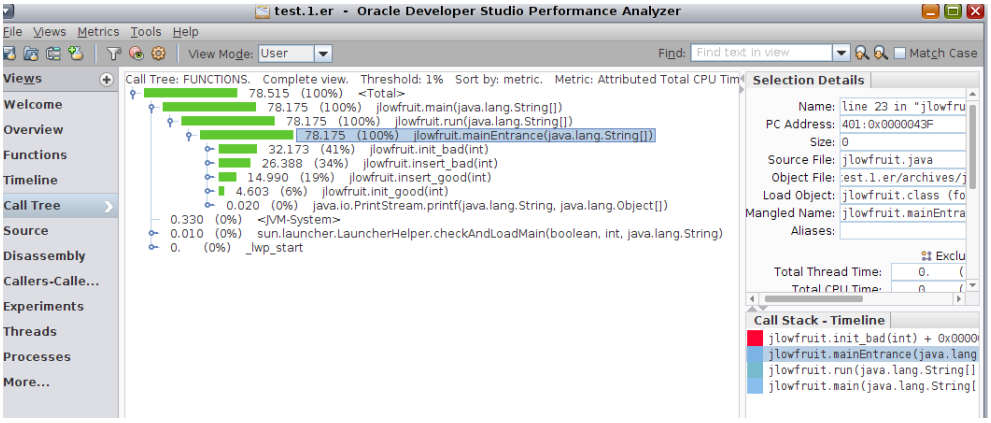
    十分にズームインすると、表示されるデータは連続していなくても、プロファイルティック (この例では約 10 ミリ秒) ごとに 1 つの個別のイベントで構成されていることがわかります。



「タイムライン」ビューに関する詳細情報のヘルプを表示するには、F1 キーを押します。

20. 「呼び出しツリー」ビューをクリックするか、「ビュー」->「呼び出しツリー」を選択してプログラムの構造を表示します。

    「呼び出しツリー」ビューには、パフォーマンス情報の注釈が付いたプログラムの動的呼び出しグラフが表示されます。



Java および Java-C++ 混合プロファイリング

この章では、次の内容について説明します。

Java-C++ プロファイリングチュートリアルについて

このチュートリアルでは、Java プロファイリングのための Oracle Developer Studio パフォーマンスアナライザの機能について説明します。これは、サンプルコードを使用してパフォーマンスアナライザで次を行う方法を示します。

- 「概要」ページ、「スレッド」ビュー、「関数」ビュー、「タイムライン」ビューを含むさまざまなデータビューでパフォーマンスデータを調べます。
- Java コードと C++ コードの両方について「ソース」と「逆アセンブリ」を調べます。
- ユーザーモード、エキスパートモード、およびマシンモードの違いを学習します。
- プログラムを実行する JVM の動作にドリルダウンして、HotSpot コンパイル済みメソッドのために生成されるネイティブコードを確認します。
- ユーザーコードによってガベージコレクタを呼び出す方法と、HotSpot コンパイラを呼び出す方法を確認します。

`jsynprog` は、Java プログラムに特有のサブタスクを多数含んでいる Java プログラムです。また、このプログラムは、C++ 共有オブジェクトをロードして、このオブジェクトからさまざまなルーチンを呼び出して、動的にロードされた C++ ライブラリから、Java コードからネイティブコードにシームレスに移行して再度ネイティブコードから Java コードに戻す方法を示します。

`jsynprog.main` は、さまざまなクラスから関数を呼び出すメインメソッドです。これは、Java Native Interface (JNI) 呼び出しによって `gethrtime` と `gethrvtime` を使用して独自の動作

の時間を設定し、独自のタイミングでアカウンティングファイルを書き込んで、さらにはメッセージを stdout に書き込みます。

jsynprog.main には多数のメソッドがあります。

- Routine.memalloc はメモリー割り当てを行い、ガベージコレクションを呼び出します
- Routine.add_int は整数の加算を行います
- Routine.add_double は倍精度 (浮動小数点) の加算を行います
- Sub_Routine.add_int は、Routine.add_int をオーバーライドする派生呼び出しです
- Routine.has_inner_class は内部クラスを定義してこれを使用します
- Routine.recurse は直接的な再帰を示します
- Routine.recursedeep は深い再帰を行なって、切り詰められたスタックをツールで処理する方法を示します
- Routine.bounce は間接的な再帰を示します。この場合、bounce は、次に bounce への呼び出しを行う bounce_b を呼び出します
- Routine.array_op は配列の演算を行います
- Routine.vector_op はベクトル演算を行います
- Routine.sys_op は、システムクラスのメソッドを使用します
- jsynprog.jni_JavaJavaC: Java メソッドは、C 関数を呼び出す別の Java メソッドを呼び出します
- jsynprog.JavaCJava: Java メソッドは、次に Java メソッドを呼び出す C 関数を呼び出します
- jsynprog.JavaCC: Java は、別の C 関数を呼び出す C 関数を呼び出します

これらのメソッドの一部はほかのメソッドから呼び出されるため、すべてが最上位タスクを表しているわけではありません。

記録する実験で目にするデータは、ここで示されているものとは異なります。チュートリアルのスクリーンショットに使用される実験は、Oracle Solaris 11.3 が実行されている SPARC T5 システムで記録されました。Oracle Solaris または Linux を実行する x86 システムからのデータは異なります。さらに、データ収集には統計的な性質があり、同じシステムおよび OS で実行する場合でも実験ごとに異なります。

パフォーマンスアナライザの実際のウィンドウ構成は、スクリーンショットと厳密に一致しない場合があります。パフォーマンスアナライザでは、ウィンドウのコンポーネント間の区切りバーをドラッグしたり、コンポーネントを縮小したり、ウィンドウのサイズを変更したりできます。パフォーマンスアナライザはその構成を記録し、次回の実行時に同じ構成を使用します。チュートリアルで示しているスクリーンショットの取得過程で、多くの構成を変更しました。

jsynprog サンプルコードの設定

始める前に:

コードの取得および環境の設定については、次を参照してください。

- 10 ページの「チュートリアルのサンプルコードの入手」
- 11 ページの「環境をチュートリアル用に設定」

パフォーマンスアナライザについてよく理解するために、最初に「Java プロファイリングの概要」の入門チュートリアルを行うことができます。

1. 次のコマンドを使用して、jsynprog ディレクトリの内容を独自のプライベート作業領域にコピーします。

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

   directory は使用する作業ディレクトリです。
2. その作業ディレクトリのコピーを変更します。

   ```
   % cd directory/jsynprog
   ```
3. ターゲットの実行可能ファイルをビルドします。

   ```
   % make clobber

   % make
   ```

   注記 - clobber サブコマンドコマンドは、このディレクトリで以前に make を実行した場合にのみ必要ですが、どの場合にも安全に使用できます。

   ---

   make, の実行後に、このディレクトリには、チュートリアルで使用するターゲットアプリケーション、jsynprog.class という名前の Java クラスファイル、および動的にロードされて Java プログラムから呼び出される C++ コードが入った libcloop.so という名前の共有オブジェクトが含められます。

ヒント - 必要に応じて、次のために Makefile を編集できます。デフォルトの Oracle Developer Studio コンパイラではなく GNU コンパイラを使用します。デフォルトの 64 ビットではなく 32 ビットでビルドします。各種のコンパイラフラグを追加します。

---

jsynprog からのデータの収集

データを収集するためのもっとも簡単な方法は、jsynprog ディレクトリで次のコマンドを実行することです。

```
% make collect
```

Makefile の collect ターゲットは、collect コマンドを起動して実験を記録します。デフォルトでは、実験の名前は test.1.er です。

collect ターゲットは、JVM にオプションの -J "-Xmx100m -XX:ParallelGCThreads=10" を指定し、デフォルトでクロックプロファイリングデータを収集します。

または、パフォーマンスアナライザの「アプリケーションのプロファイル」ダイアログを使用してデータを記録することもできます。Java 入門チュートリアルの 31 ページの「パフォーマンスアナライザを使用した jlowfruit からのデータの収集」の手順に従って、「引数」フィールドに jlowfruit ではなく jsynprog を指定します。

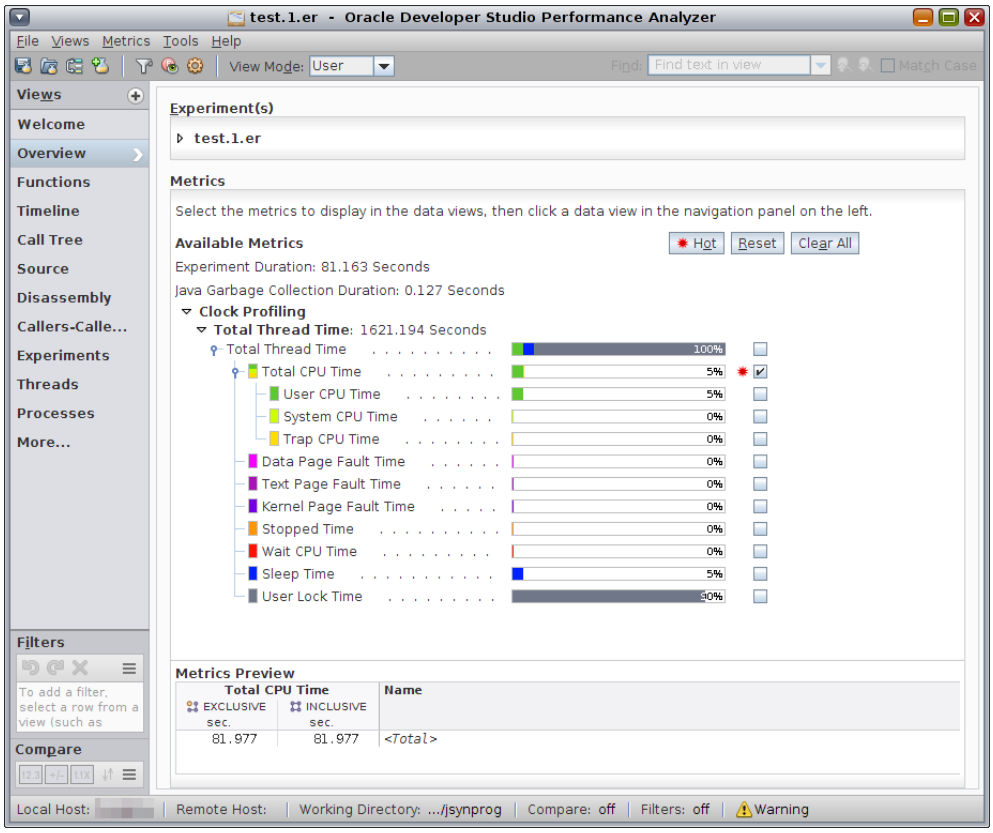
jsynprog データの検査

この手順では、前のセクションの説明に従って実験をすでに作成してあることを前提としています。

1. jsynprog ディレクトリからパフォーマンスアナライザを起動して、実験の名前が test.1.er ではない場合は名前を指定して次のように実験をロードします。

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

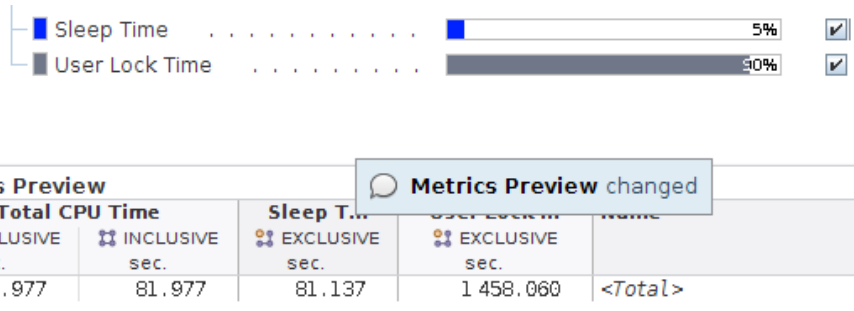
   実験が開くと、パフォーマンスアナライザの「概要」ページが表示されます。



現在、パフォーマンスアナライザのツールバーには、最初は「ユーザーモード」に設定されているビューモードセレクタがあり、これはプログラムのユーザーモデルを示しています。

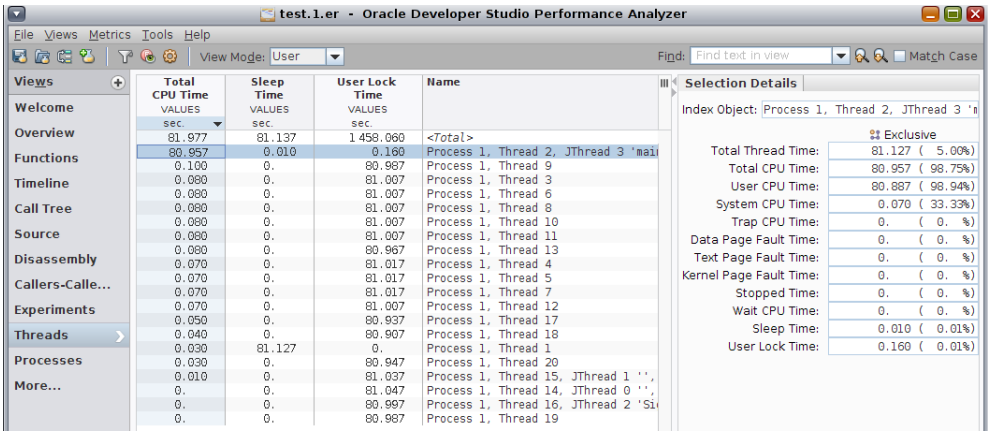
「概要」には、実験が約 81 秒実行されたが、使用合計時間は 1600 秒を超えたことを示しており、プロセス内には平均で 20 個のスレッドがあったことを意味しています。

2. 「休眠時間」メトリックと「ユーザーロック時間」メトリックのチェックボックスにチェックマークを付けて、データビューに追加します。



これらのメトリックが追加されたデータの外観を示すために「メトリックのプレビュー」が更新されます。

3. ナビゲーションパネルで「スレッド」ビューを選択すると、スレッドのデータが表示されます。



スレッド 2 のみでかなりの「CPU 合計時間」が集計されました。ほかの各スレッドには、「CPU 合計時間」の少数のプロファイルイベントのみがありました。

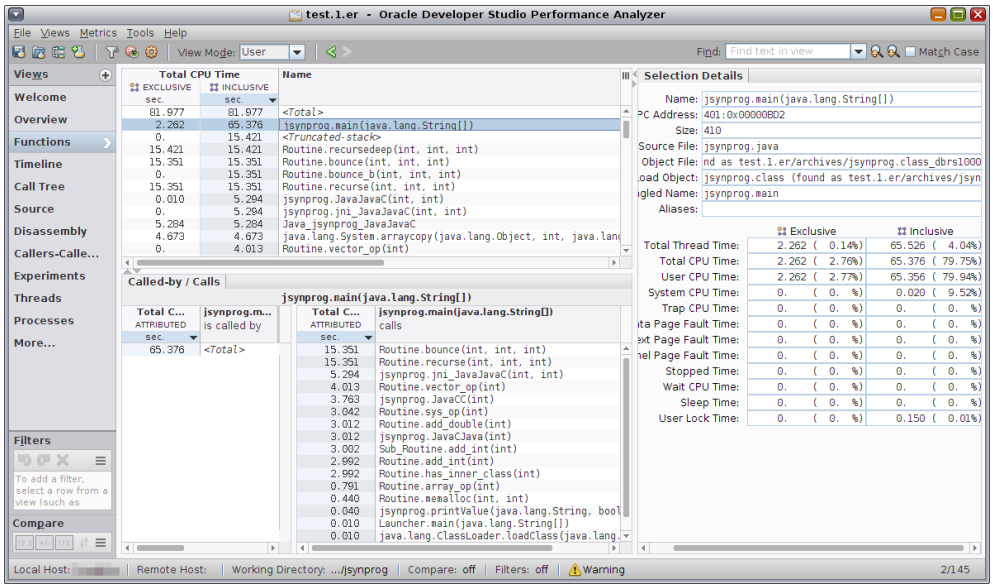
4. 「スレッド」ビューで任意のスレッドを選択して、そのスレッドに関するすべての情報を右側の「選択の詳細」ウィンドウに表示します。

   スレッド 1 とスレッド 2 を除くほとんどすべてのスレッドがすべての時間を「ユーザーロック」状態で費やしたことがわかるはずです。これは、JVM がそれ自体を内部で同期する方法を示しています。スレッド 1 は、ユーザー Java コードを起動してから、終了するまでスリープ状態になります。

5. 「概要」に戻って、「休眠時間」と「ユーザーロック時間」の選択を解除します。
6. ナビゲーションパネルで「関数」ビューを選択して、列ヘッダーをクリックして「排他的 CPU 合計時間」、「包括的 CPU 合計時間」、または「名前」でソートします。

降順または昇順でソートできます。

リストを「包括的 CPU 合計時間」で降順にソートしたままにして、最上位の関数 `jsynprog.main()` を選択します。そのルーチンは、JVM が実行を開始するために呼び出す初期ルーチンです。



「関数」ビューの下部にある「呼び出し元/呼び出し回数」パネルは、`jsynprog.main()` 関数が `<Total>` によって呼び出されることを示しており、これはスタックの上部にあったことを意味しています。

パネルの「呼び出し回数」側には、`jsynprog.main()` が、45 ページの「Java-C++ プロファイリングチュートリアルについて」に示されている、メインルーチンから直接呼び出されるサブタスクごとにさまざまな異なるルーチンを 1 つ呼び出すことが示されています。リストには少数のほかのルーチンも含まれています。

Java および C++ 混合コードの検査

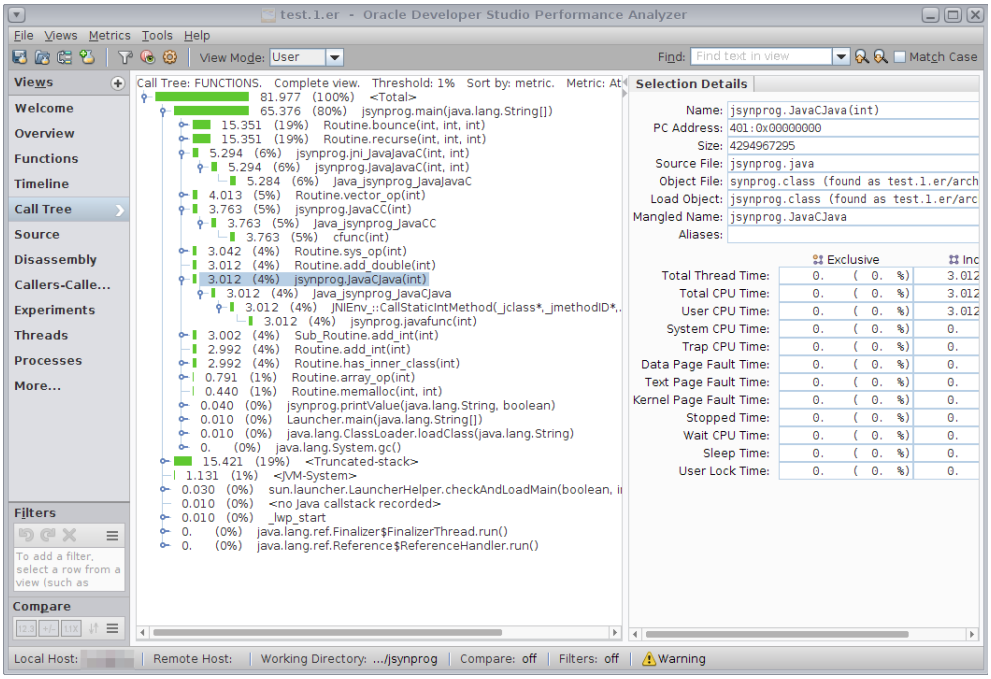
このセクションでは、「呼び出しツリー」ビューと「ソース」ビューについて説明し、Java から C++ への呼び出しと C++ から Java への呼び出しの間の関係を確認する方法を示します。また、ナビゲーションパネルに「逆アセンブリ」ビューを追加する方法を示します。

1. 「関数」ビューのリストの上部にある各関数を順に選択して、「選択の詳細」ウィンドウで詳細情報を調べます。

関数によっては、ソースファイルが `jsynprog.java` と報告されるものと、`cloop.cc` と報告されるものがあります。これは、`jsynprog` プログラムによって、`cloop.cc` C++ ソースファイルから構築された `libcloop.so` という名前の C++ 共有オブジェクトがロードされたためです。パフォーマンスアナライザでは、Java から C++ への呼び出しと C++ から Java への呼び出しがシームレスに報告されます。

2. ナビゲーションツリーで「呼び出しツリー」を選択します。

   「呼び出しツリー」ビューには、Java と C++ の間のこれらの呼び出しが行われる方法がグラフィカルに示されます。

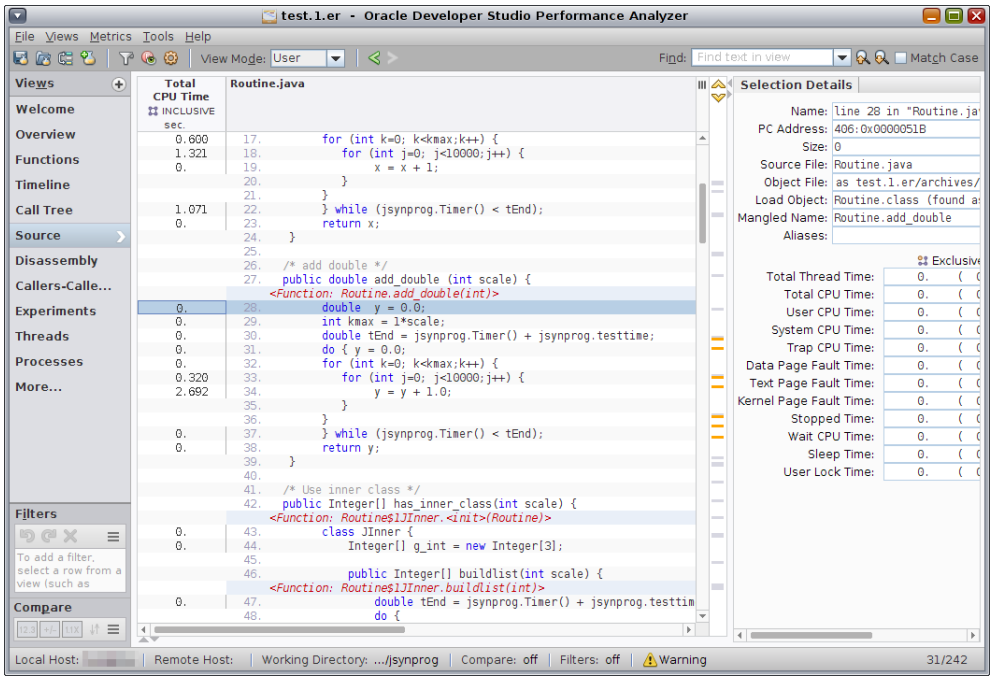


3. 「呼び出しツリー」ビューで次を行なって、Java から C++ への呼び出しと C++ から Java への呼び出しを確認します。

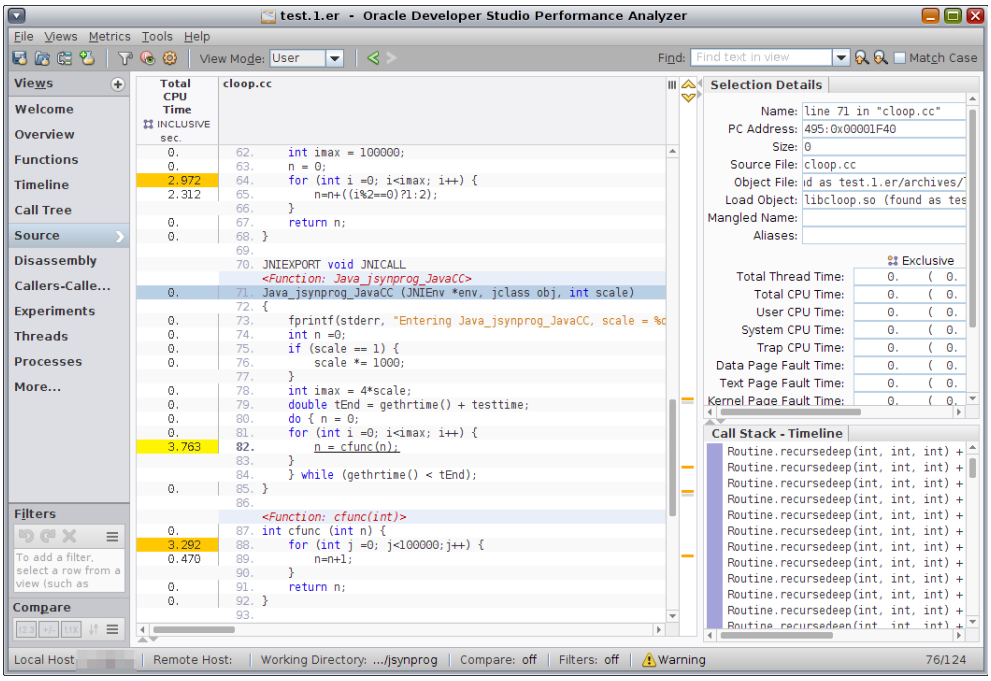
   - 名前に「C」が含まれるさまざまな関数を指す行を展開します。
   - `jsynprog.JavaCC()` の行を選択します。この関数は Java コードから生成されますが、C++ コードから生成された `Java_jsynprog_JavaCC()` への呼び出しを行います。
   - `jsynprog.JavaCJava()` の行を選択します。この関数も Java コードから生成されますが、C++ コードである `Java_jsynprog_JavaCJava()` を呼び出します。その関数は、Java からメソッド `jsynprog.javafunc()` への呼び出しを行う `JNIEnv_::CallStaticIntMethod()` の C++ メソッドへの呼び出しを行います。

4. Java または C++ のいずれかからメソッドを選択して、「ソース」ビューに切り替えて適切な言語で示されたソースをパフォーマンスメトリックとともに表示します。

Java メソッドを選択したあとの「ソース」ビューの例を次に示します。

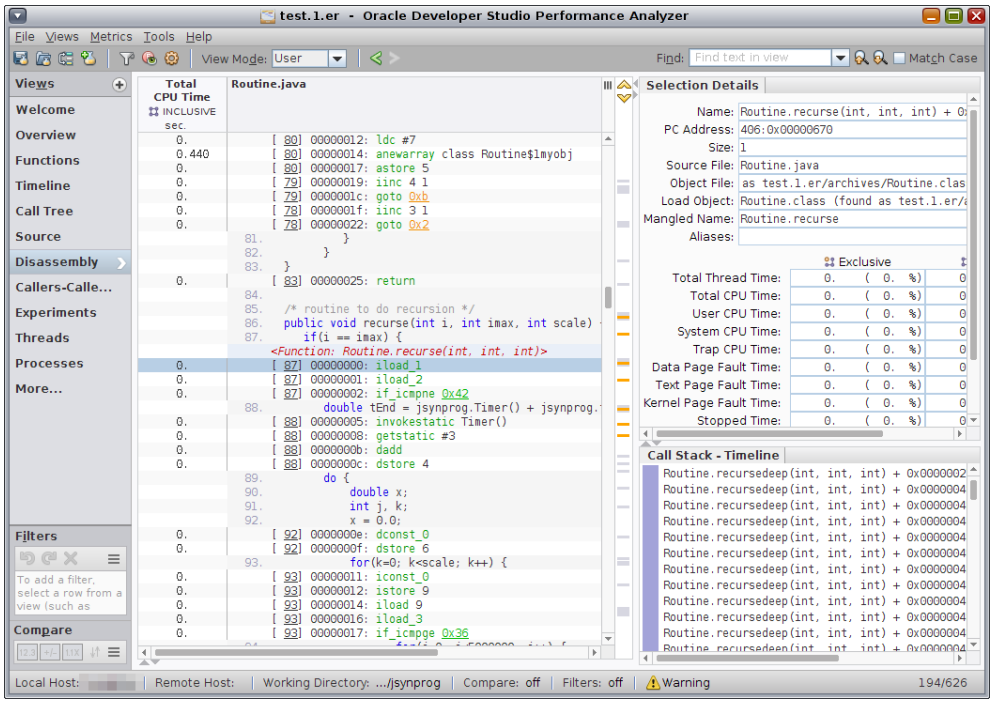


C++ メソッドを選択したあとの「ソース」ビューの例を次に示します。

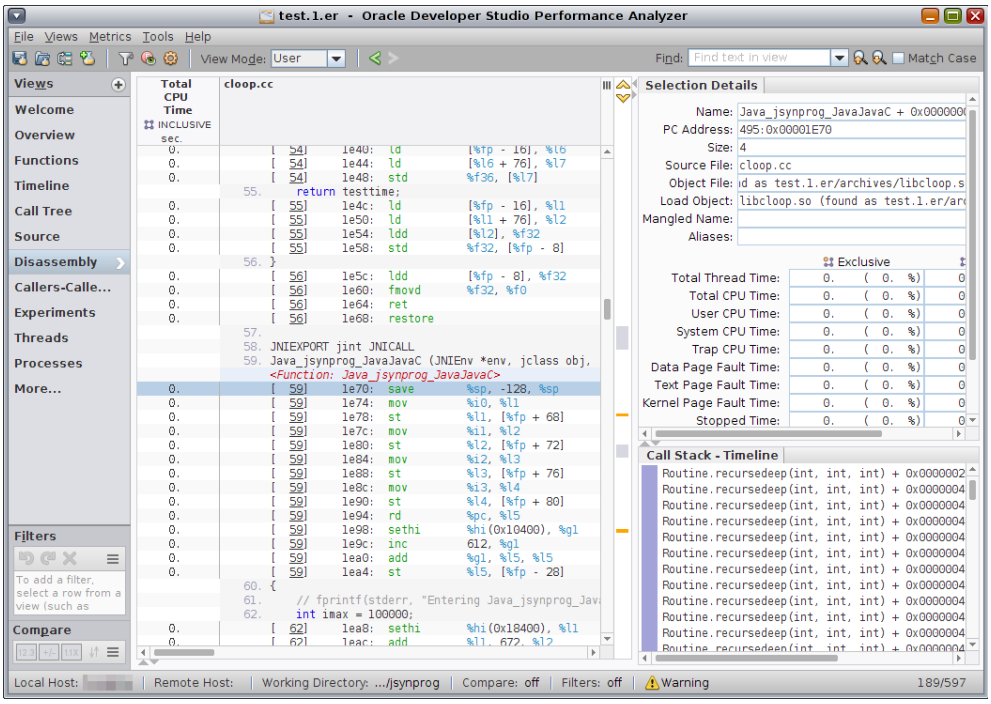


5. ナビゲーションパネルに「逆アセンブリ」タブがまだ表示されていない場合は、ナビゲーションパネルの上部で、「ビュー」ラベルの横にある「+」ボタンをクリックし、「逆アセンブリ」のチェックボックスにチェックマークを付けて、このビューを追加します。

   最後に選択した関数の「逆アセンブリ」ビューが表示されます。Java 関数の場合、次のスクリーンショットに示すように、「逆アセンブリ」ビューには Java バイトコードが示されます。



C++ 関数の場合、次のスクリーンショットに示すように、「逆アセンブリ」ビューにはネイティブマシンコードが示されます。



次のセクションでは、「逆アセンブリ」ビューをさらに使用します。

JVM の動作の理解

このセクションでは、フィルタ、エクスパートモード、およびマシンモードを使用することで JVM で何が発生しているかを調べる方法を示します。

1. 「関数」ビューを選択して、<JVM-System> という名前のルーチンを見つけます。

   <JVM と入力して Enter を押した場合、ツールバーの「検索」ツールを使用すると非常にすばやく見つけることができます。

   この実験では、<JVM-System> は約 1 秒間の CPU 合計時間を使用しました。<JVM-System> 関数の「時間」は、ユーザーコードではなく JVM の作業を表しています。

2. <JVM-System> を右クリックして、「フィルタを追加: 選択した関数を含むスタックのみを含める」を選択します。

   ナビゲーションパネルの下にある「フィルタ」パネルには、以前は「アクティブなフィルタはありません」と表示されており、現在は追加したフィルタの名前とともに 1 つのアクティブな

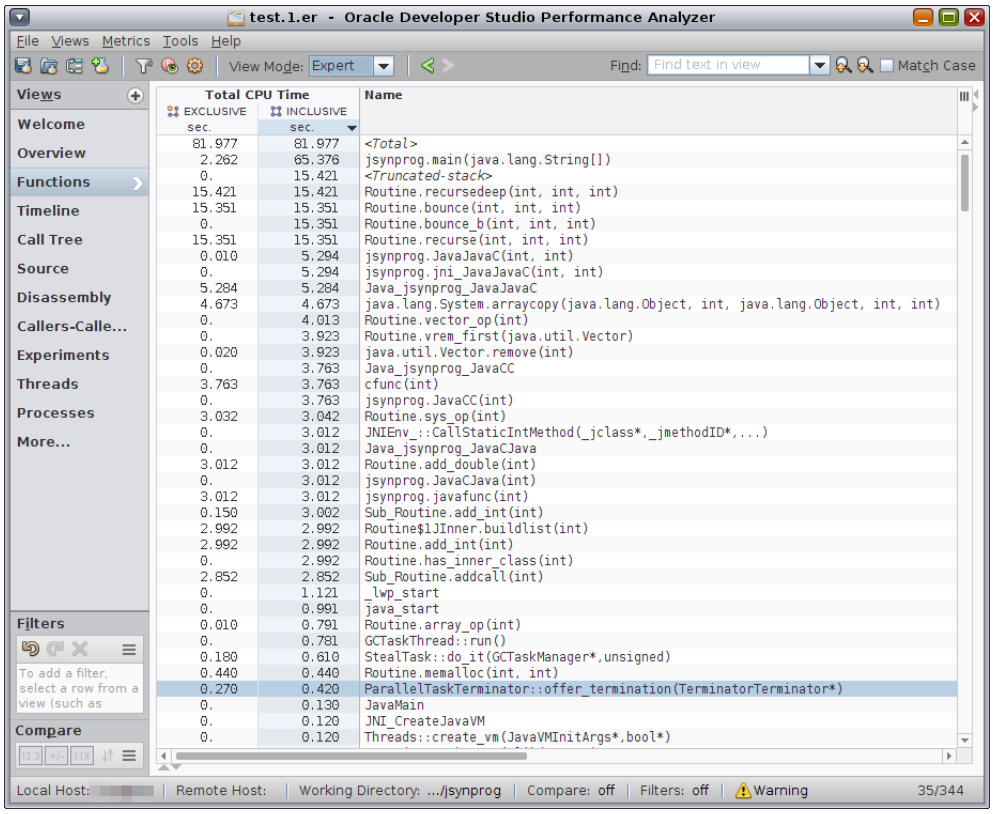
フィルタと表示されています。<JVM-System> のみが残るように「関数」ビューがリフレッシュされます。

3. パフォーマンスアナライザのツールバーで、ビューモードセレクタを「ユーザーモード」から「エキスパートモード」に変更します。

   「関数」ビューがリフレッシュされて、<JVM-System> 時間で表された多数の関数が表示されます。関数 <JVM-System> 自体は表示されなくなります。

4. 「アクティブなフィルタ」パネルの「X」をクリックしてフィルタを削除します。

   「関数」ビューがリフレッシュされ、ユーザー関数が再度表示されますが、<JVM-System> によって表された関数はまだ表示されているのに対して、<JVM-System> 関数は表示されなくなります。

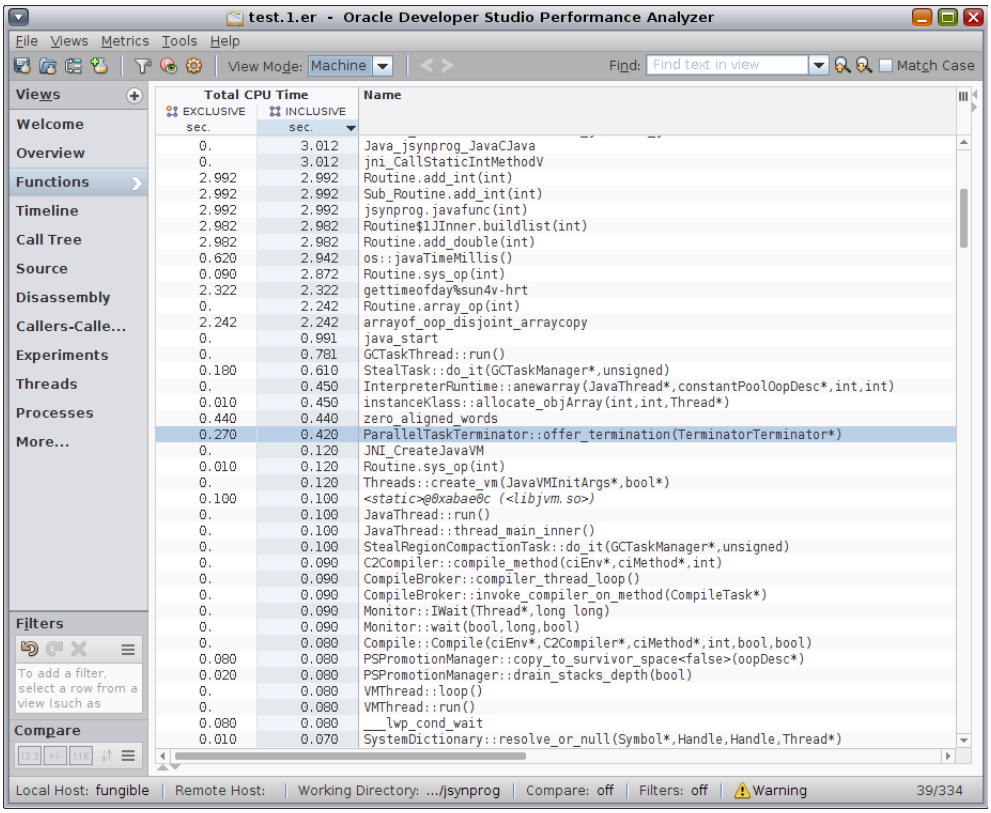


<JVM-System> を展開するためにフィルタリングを実行する必要はありません。この手順には、ユーザーモードとエキスパートモードの違いをより簡単に示すためのフィルタリングが含まれています。

要約すると、ユーザーモードではすべてのユーザー関数が表示されますが、JVM で使用したすべての時間が <JVM-System> に集計されるのに対して、エキスパートモードではその <JVM-System> アグリゲーションが展開されます。

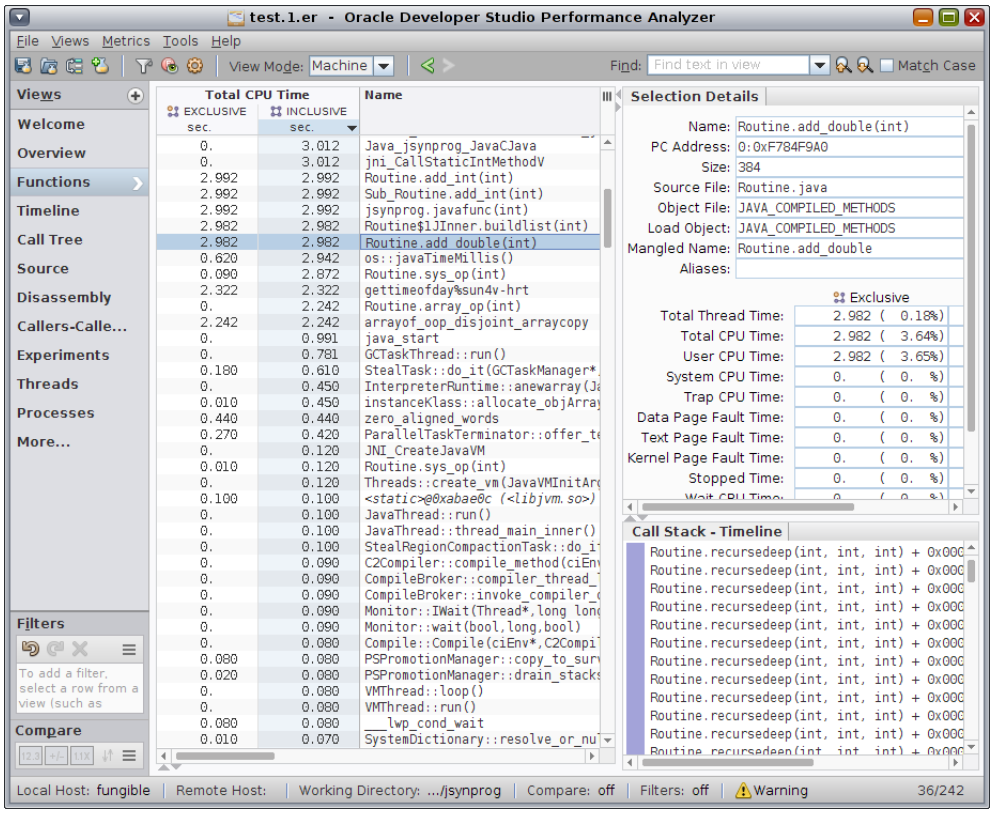
次に、マシンモードを調べることができます。

5. 表示モードリストで「マシンモード」を選択します。



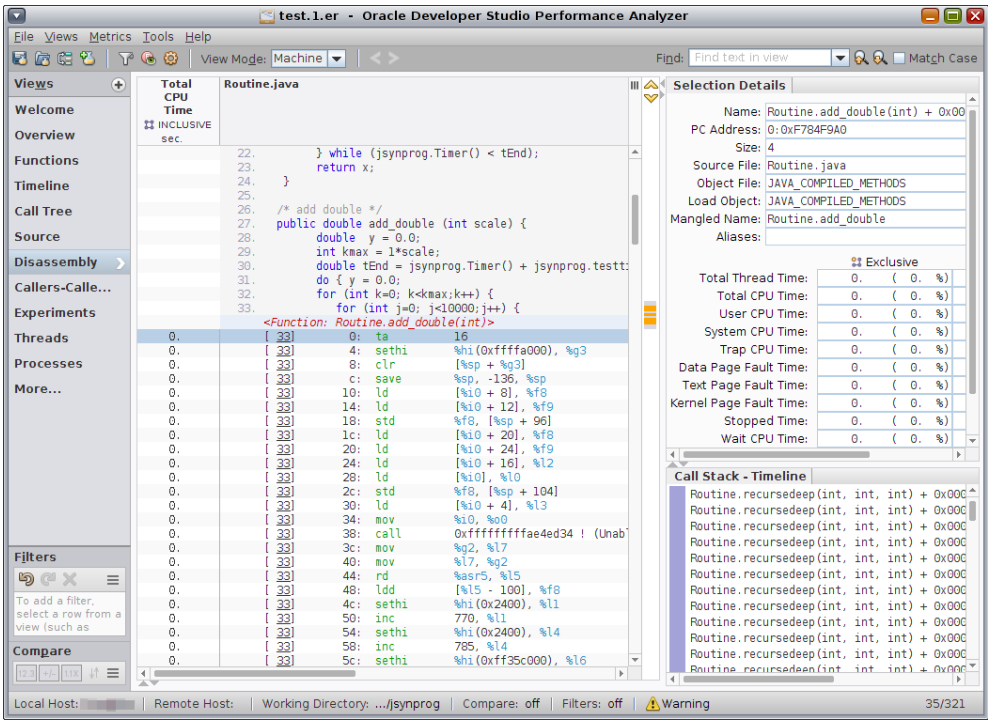
マシンモードでは、解釈されるどのユーザーメソッドも「関数」ビューで名前別には表示されません。解釈されるメソッドに使用された時間は、Java バイトコードを説明的に実行する JVM の一部を表す `Interpreter` エントリに集計されます。

ただし、マシンモードでは、「関数」ビューには HotSpot コンパイル済みのすべてのユーザーメソッドが表示されます。`Routine.add_int()` などのコンパイル済みメソッドを選択すると、「選択の詳細」ウィンドウには、メソッドの Java ソースファイルが「ソースファイル」として表示されますが、「オブジェクトファイル」と「ロードオブジェクト」には `JAVA_COMPILED_METHODS` が表示されます。



6. マシンモードのままで、「関数」ビューでコンパイル済みメソッドが選択された状態で「逆アセンブリ」ビューに切り替えます。

   「逆アセンブリ」ビューには、HotSpot コンパイラによって生成されたマシンコードが表示されます。コードの上にある列ヘッダーで「ソースファイル」、「オブジェクトファイル」、および「ロードオブジェクト」の名前を確認できます。



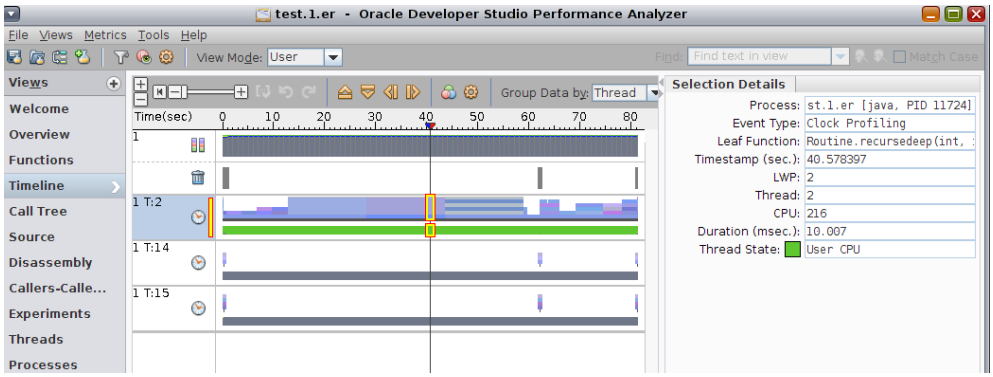
その関数のほとんどの作業がコードでさらに実行されるため、ほとんどの表示可能な行で表示される「CPU 合計時間」はゼロです。

次のセクションに進みます。

Java ガベージコレクタの動作の理解

この手順では、Java ガベージコレクションを呼び出すアクティビティーを調べて、「タイムライン」ビューの使用方法と「タイムライン」でのビューモード設定の影響を示します。

1. ビューモードを「ユーザーモード」に設定して、ナビゲーションパネルで「タイムライン」を選択し、このハイブリッド Java/ネイティブアプリケーション `jsynprog` の実行の詳細を表示します。



上部に「CPU 使用率標本」バーと、3 つのスレッドのプロファイルデータが表示されるはずです。スクリーンショットでは、プロセス 1、スレッド 2、14、15 のデータが表示されています。表示されるスレッドの番号付けと数は、OS、システム、および使用している Java のバージョンによって異なる可能性があります。

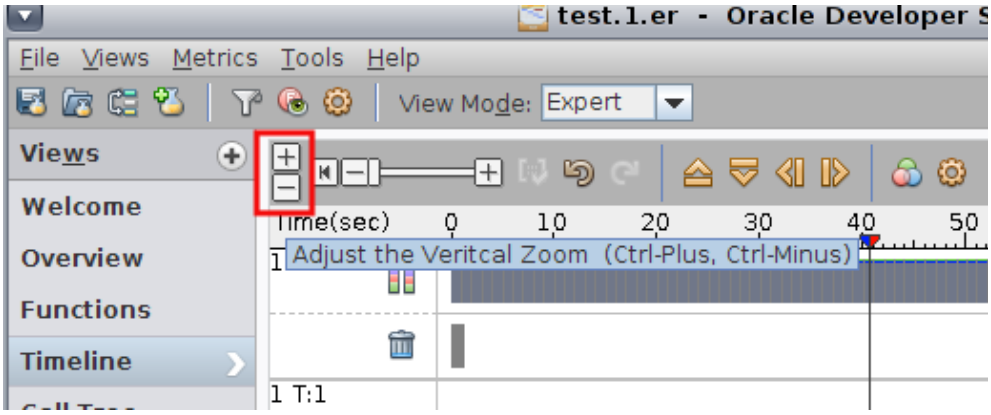
この例では、最初のスレッドである T:2 というラベルが付いたスレッド 2 のみで、「ユーザー CPU」としてマイクロステートが表示されます。ほかの 2 つのスレッドはそのすべての時間を JVM 同期の一部である「ユーザーロック」の待機に使用します。

2. ビューモードを「エキスパートモード」に設定します。

   ユーザースレッド T:2 はほとんど変更されていないように見えますが、「タイムライン」ビューにはより多くのスレッドが表示されるようになったはずです。

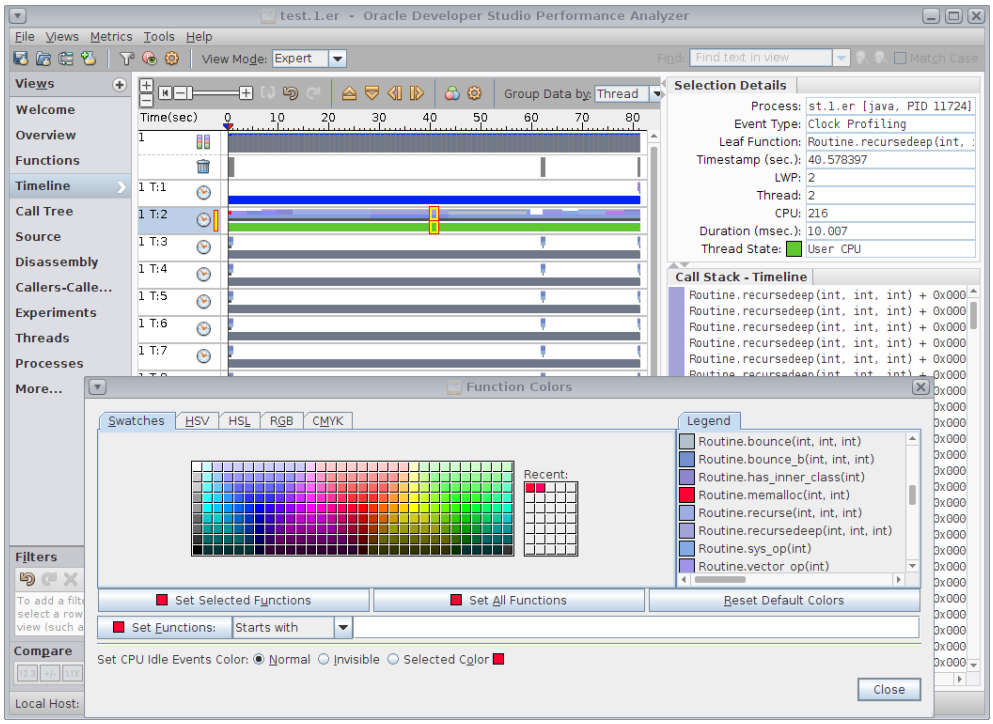
3. タイムラインの上部にある縦方向ズームコントロールを使用して、すべてのスレッドを表示できるようにズームを調整します。

   次のスクリーンショットでは、縦方向ズームコントロールは赤色で強調表示されています。マイナスボタンをクリックして、20 個のスレッドすべてが表示されるまでスレッド行の高さを縮小します。



4. 「タイムライン」ツールバーの「呼び出しスタック関数の色」アイコン をクリックして、関数 `Routine.memalloc()` の色を赤色に設定します。

   「関数の色」ダイアログの「凡例」で `Routine.memalloc()` 関数を選択して、「スイッチ」内の赤色のボックスをクリックして「選択した関数の設定」をクリックします。

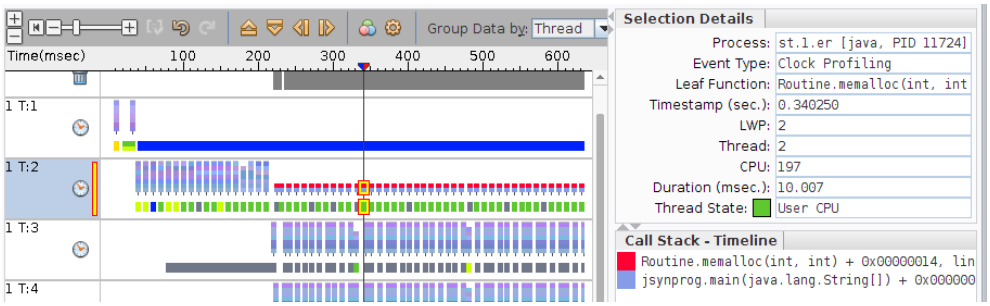


スレッド 2 のスタックの上部に赤色のバーが表示されるようになりました。その領域は、`Routine.memalloc()` ルーチンが実行されていた時間の一部を表しています。

縦方向にズームアウトして呼び出しスタックのフレームをさらに表示して、対象の時間の領域に横方向にズームインする必要がある場合があります。

5. 「タイムライン」ツールバーの横方向スライダを使用して、スレッド T:2 内の個々のイベントが表示されるほど近くまでズームインします。

   キーボードの + キーをダブルクリックするか押すことでズームすることもできます。



タイムラインの各行には、実際には 3 つのデータバーが含まれています。上部のバーは、そのイベントの呼び出しスタックの表現です。中央のバーには、すべてのイベントを表示するために、イベントが近すぎる間隔で一緒に発生した場合は常に黒色のティックマークが表示されます。つまり、ティックマークが表示された場合、そのスペースに複数のイベントがあることがわかります。

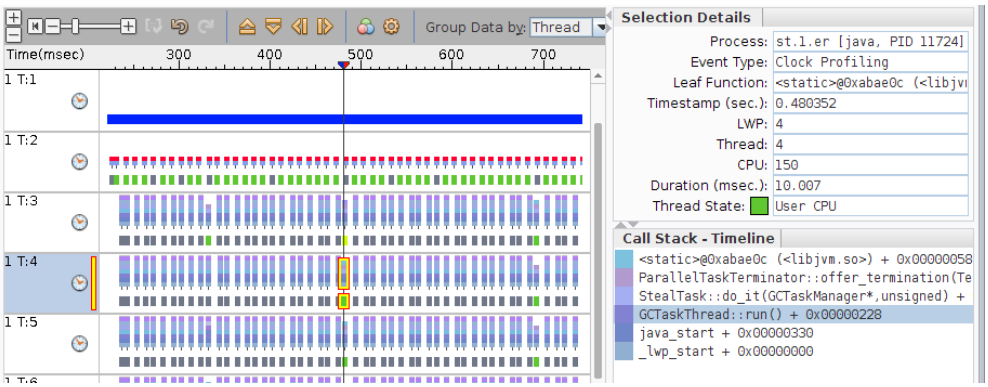
下部のバーはイベント状態のインジケータです。T:2 の場合、下部のバーは緑で、「ユーザー CPU 時間」が使用されていたことを示しています。スレッド 3 - 12 の場合、下部のバーは灰色で、「ユーザーロック時間」を示しています。

ただし、ユーザースレッド T:2 は赤色で示されているルーチン `Routine.memalloc` 内にあるため、これらのスレッド 3 - 12 すべてに、同時に到着するクラスタ化されたイベントが多数あります。

6. `Routine.memalloc` 領域までズームインして、次を行なってその領域のみを含めるためにフィルタします。
   - 上部に赤色の関数呼び出しがある `Routine.memalloc` 領域の先頭近くにある T:2 バーをクリックします。
   - クリックして、呼び出しスタックの先頭にある赤色が終わるその領域の終わりの近くまでマウスをドラッグします。
   - 右クリックして、「ズーム」->「選択した時間範囲まで」を選択します。
   - まだ範囲を選択した状態で、右クリックして「フィルタの追加: 選択した時間範囲と交差するイベントのみを含める」を選択します。

   ズーム後に、スレッド 3 - 12 には、「ユーザー CPU 時間」を示すために緑になっているいくつかのイベント状態と、「CPU 待ち時間」を示すために赤色になっている少数のイベント状態があることがわかります。
7. スレッド 3 - 12 でいずれかのイベントをクリックすると、それぞれのスレッドのイベントにスタックの `GCTaskThread::run()` が含まれていることが「呼び出しスタック」パネルでわかります。

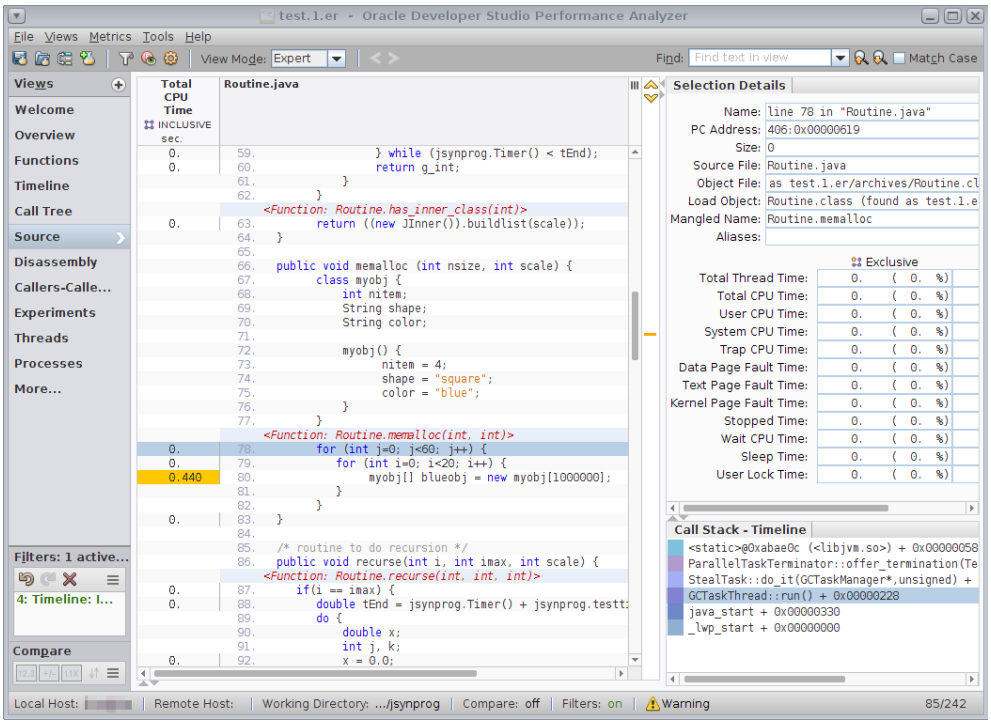
   これらのスレッドは、JVM がガベージコレクションを実行するために使用するスレッドを表します。GC スレッドは、ユーザー CPU 時間を多く使用せず、ユーザースレッドが `Routine.memalloc` 内にある間だけ実行されます。



8. 「関数」ビューに戻って、「包括的 CPU 合計」列ヘッダーをクリックして、包括的 CPU 時間合計でソートします。

   最上位の関数の 1 つが `GCTaskThread::run()` 関数であることがわかるはずです。これによって、ユーザータスク `Routine.memalloc` が何らかの方法でガベージコレクションを呼び出しているという結論に達します。

9. `Routine.memalloc` 関数を選択して、「ソース」ビューに切り替えます。



ソースコードのこのフラグメントから、ガベージコレクションが呼び出されている理由が簡単にわかります。このコードは、100 万個のオブジェクトの配列を割り当てて、同じ場所にありそれぞれがループを通過するこれらのオブジェクトへのポインタを格納します。これによって古いオブジェクトは使用されなくなるため、不要になります。

次のセクションに進みます。

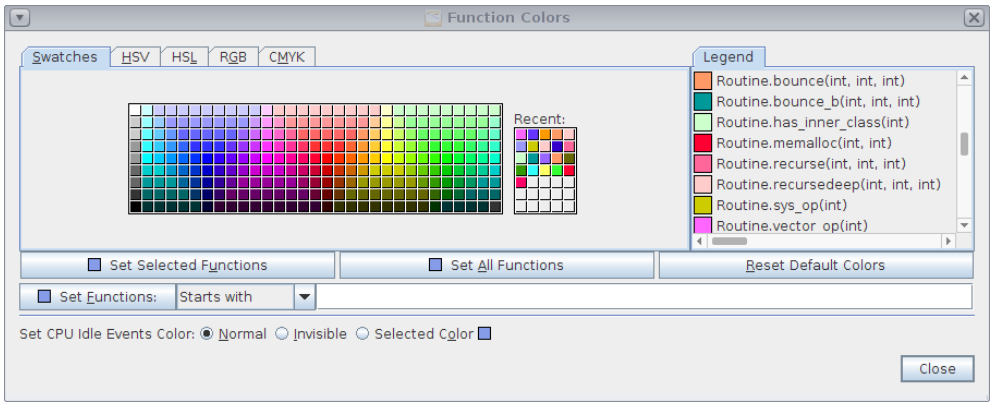
Java HotSpot コンパイラの動作の理解

この手順は前のセクションからの続きであり、「タイムライン」ビューと「スレッド」ビューを使用して HotSpot コンパイルを行うスレッドをフィルタして見つける方法を示しています。

1. 「タイムライン」ビューを選択して、「アクティブなフィルタ」パネルの「X」をクリックしてフィルタを削除してから、キーボードの 0 を押して横方向ズームをデフォルトにリセットします。

   「タイムライン」ツールバーの横方向スライダの前面にある |< ボタンをクリックすることも、「タイムライン」で右クリックして「リセット」を選択することもできます。
2. 「関数の色」ダイアログを再度開き、`Routine.*` 関数ごとに異なる色を選択します。

「タイムライン」ビューで、色の変更がスレッド 2 の呼び出しスタック内に表示されます。



3. スレッド 2 の色の変更が表示される期間、「タイムライン」のすべてのスレッドを確認します。

   イベントのパターンがスレッド 2 の色の変更とほとんど同じ時間に発生しているスレッドがいくつかあることがわかるはずです。この例では、スレッド 17、18、および 19 です。

4. 「スレッド」ビューに移動して、スレッド 2 と、スレッド 2 が `Routine.*` 関数への呼び出しを示す期間のアクティビティーを示す実験内のスレッドを選択します。

   「名前」列ヘッダーをクリックして、最初に名前でソートすると簡単なことがあります。次に、Ctrl を押しながらスレッドをクリックして、複数のスレッドを選択します。

   この例では、スレッド 2、17、18、19 が選択されています。

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

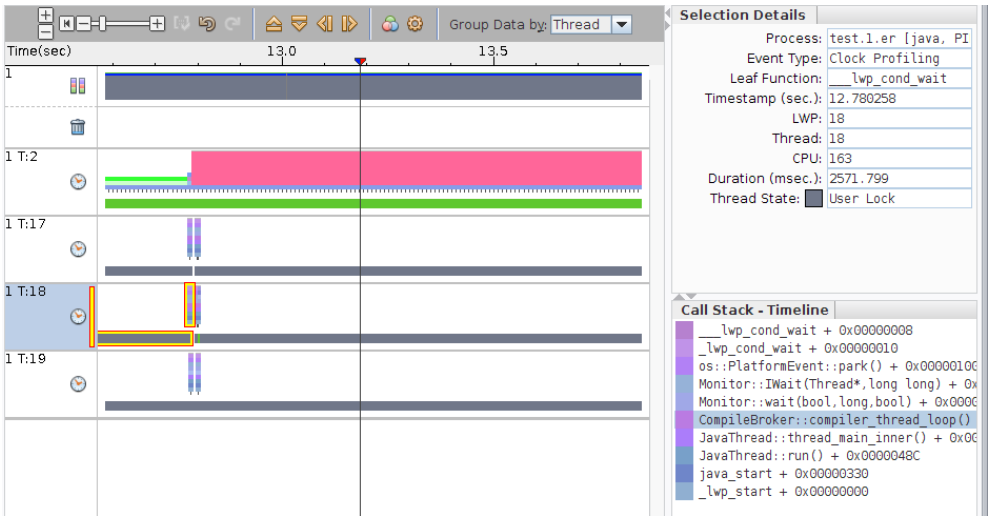
5. ツールバーのフィルタボタン  をクリックして、「フィルタを追加: 選択した項目に関するイベントのみを含める」を選択します。

   これによって、これらのスレッドのイベントのみを含めるようフィルタが設定されます。「スレッド」ビューで右クリックして、フィルタを選択することもできます。

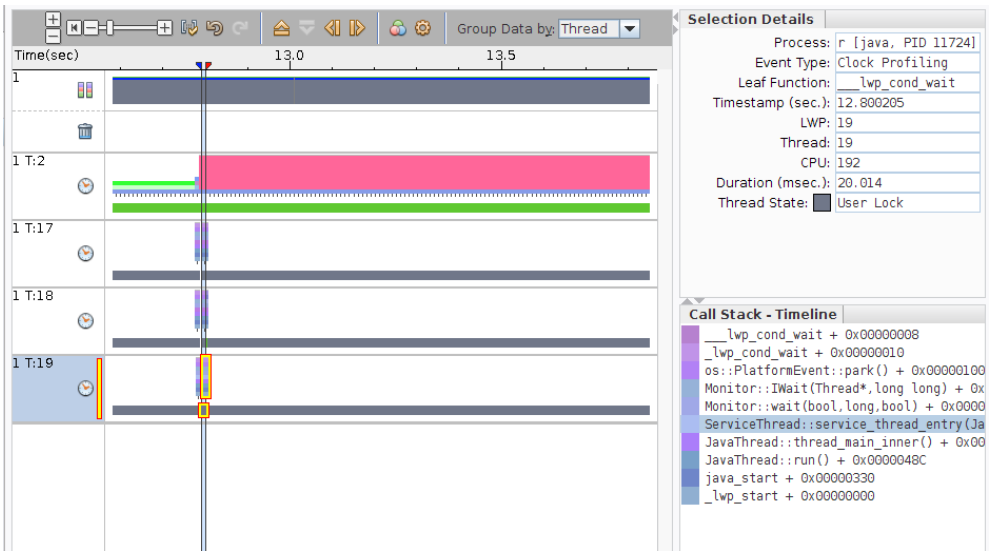
6. 「タイムライン」ビューに戻って、パターンを見やすくするために横方向ズームをリセットします。

7. スレッド 17 および 18 内のイベントをクリックします。

   「呼び出しスタック」パネルには `CompileBroker::compiler_thread_loop()` が表示されます。これらのスレッドは、HotSpot コンパイラに使用されるスレッドです。



スレッド 19 は、`ServiceThread::service_thread_entry()` が含まれている呼び出しスタックを示しています。



複数のイベントがこれらのスレッドで発生するのは、ユーザーコードが新しいメソッドを呼び出してそのメソッドでかなりの時間を費やすたびに、そのメソッドのマシンコードを生成する

ために HotSpot コンパイラが呼び出されるためです。HotSpot コンパイラは、実行するスレッドがユーザー CPU 時間をあまり使用しないくらいに十分に高速です。

HotSpot コンパイラが呼び出される正確な方法の詳細はこのチュートリアルの範囲外です。

マルチスレッドプログラムでのハードウェアカウンタプロファイリング

この章では、次の内容について説明します。

ハードウェアカウンタプロファイリングのチュートリアルについて

このチュートリアルでは、mttest という名前のマルチスレッドプログラムに対してパフォーマンスアナライザを使用し、クロックプロファイリングおよびハードウェアカウンタプロファイリングのデータを収集および理解する方法を示します。

「概要」ページを調査し、表示するメトリックを変更し、「関数」ビュー、「呼び出し元-呼び出し先」ビュー、および「ソース」と「逆アセンブリ」ビューを検証し、フィルタを適用します。

まず、クロックプロファイルデータを調査します。次に、サポートされているすべてのシステムで利用可能な「命令の実行」カウンタを使用して、ハードウェアカウンタプロファイルデータを調査します。そのあと、「命令の実行」と「CPU サイクル」(一部を除いた、サポートされているほとんどのシステムで利用可能)、および「データキャッシュミス」(サポートされている一部のシステムで利用可能) を調査します。

データキャッシュミス (dcm) の正確なハードウェアカウンタを備えたシステムで実行する場合、IndexObject および MemoryObject ビューの使用方法と、キャッシュ行の偽共有を検出する方法についても学びます。

mttest プログラムは、ダミーデータに対してさまざまな同期オプションを実行する単純なプログラムです。プログラムはさまざまなタスクを実装し、各タスクは次の基本アルゴリズムを使用します。

- 複数のワークブロック (デフォルトで 4 つ) をキューに入れます。各ブロックは構造体 Workblk のインスタンスです。
- 作業を処理する複数のスレッド (これもデフォルトで 4 つ) を生成します。スレッドのプライベートなワークブロックを各スレッドに渡します。
- 各タスクで、特定の同期プリミティブを使用して、ワークブロックへのアクセスを制御します。
- 同期のあと、ブロックに対する作業を処理します。

記録する実験で目にするデータは、ここで示されているものとは異なります。チュートリアルのスクリーンショットに使用した実験は、Oracle Solaris 11.3 を実行する SPARC T5 システムで記録されました。Oracle Solaris または Linux を実行する x86 システムからのデータは異なります。さらに、データ収集には統計的な性質があり、同じシステムおよび OS で実行する場合でも実験ごとに異なります。

パフォーマンスアナライザの実際のウィンドウ構成は、スクリーンショットと厳密に一致しない場合があります。パフォーマンスアナライザでは、ウィンドウのコンポーネント間の区切りバーをドラッグしたり、コンポーネントを縮小したり、ウィンドウのサイズを変更したりできます。パフォーマンスアナライザはその構成を記録し、次回の実行時に同じ構成を使用します。チュートリアルで示しているスクリーンショットの取得過程で、多くの構成を変更しました。

mttest サンプルコードの設定

始める前に:

コードの取得および環境の設定については、次を参照してください。

- 10 ページの「チュートリアルのサンプルコードの入手」
- 11 ページの「環境をチュートリアル用に設定」

パフォーマンスアナライザについてよく理解するために、「C プロファイリングの概要」の入門チュートリアルを詳細に検討することもできます。

1. 次のコマンドで、mttest ディレクトリの内容をプライベート作業領域にコピーします。

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

   mydirectory は使用する作業ディレクトリです。
2. その作業ディレクトリのコピーを変更します。

   ```
   % cd directory/mttest
   ```
3. ターゲットの実行可能ファイルをビルドします。

```
% make clobber

% make
```

注記 - clobber サブコマンドコマンドは、このディレクトリで以前に make を実行した場合にのみ必要ですが、どの場合にも安全に使用できます。

make を実行すると、このディレクトリにはチュートリアルで使用するターゲットアプリケーションである、mttest という名前の C プログラムが含まれます。

ヒント - 必要に応じて、次のために Makefile を編集できます。デフォルトの Oracle Developer Studio コンパイラではなく GNU コンパイラを使用します。デフォルトの 64 ビットではなく 32 ビットでビルドします。各種のコンパイラフラグを追加します。

ハードウェアカウンタプロファイリングのチュートリアル用に mttest からデータを収集する

データを収集するもっとも簡単な方法は、mttest ディレクトリで次のコマンドを実行することです。

```
% make hwcperf
```

Makefile の hwcperf ターゲットは、collect コマンドを起動し、実験を記録します。

注記 - このチュートリアルでは、コンパイルおよびデータ収集に前出の入門チュートリアルよりも時間がかかることがあります。

実験はデフォルトで test.1.er という名前であり、次の 3 つのカウンタのクロックプロファイリングデータおよびハードウェアカウンタプロファイリングデータを含みます。inst (命令)、cycles (サイクル)、dcm (データキャッシュミス)。

システムが cycles カウンタまたは dcm カウンタをサポートしない場合、collect コマンドは失敗します。その場合、Makefile を編集して # 記号を適切な行に移動し、システムでサポートされているカウンタのみを指定する HWC_OPT 変数を有効化します。省略されたカウンタのデータは実験で取得されません。

ヒント - collect -h コマンドを使用すると、システムがサポートするカウンタを調べることができます。ハードウェアカウンタについては、『Oracle Developer Studio 12.5: パフォーマンスアナライザ』の「ハードウェアカウンタリスト」を参照してください。

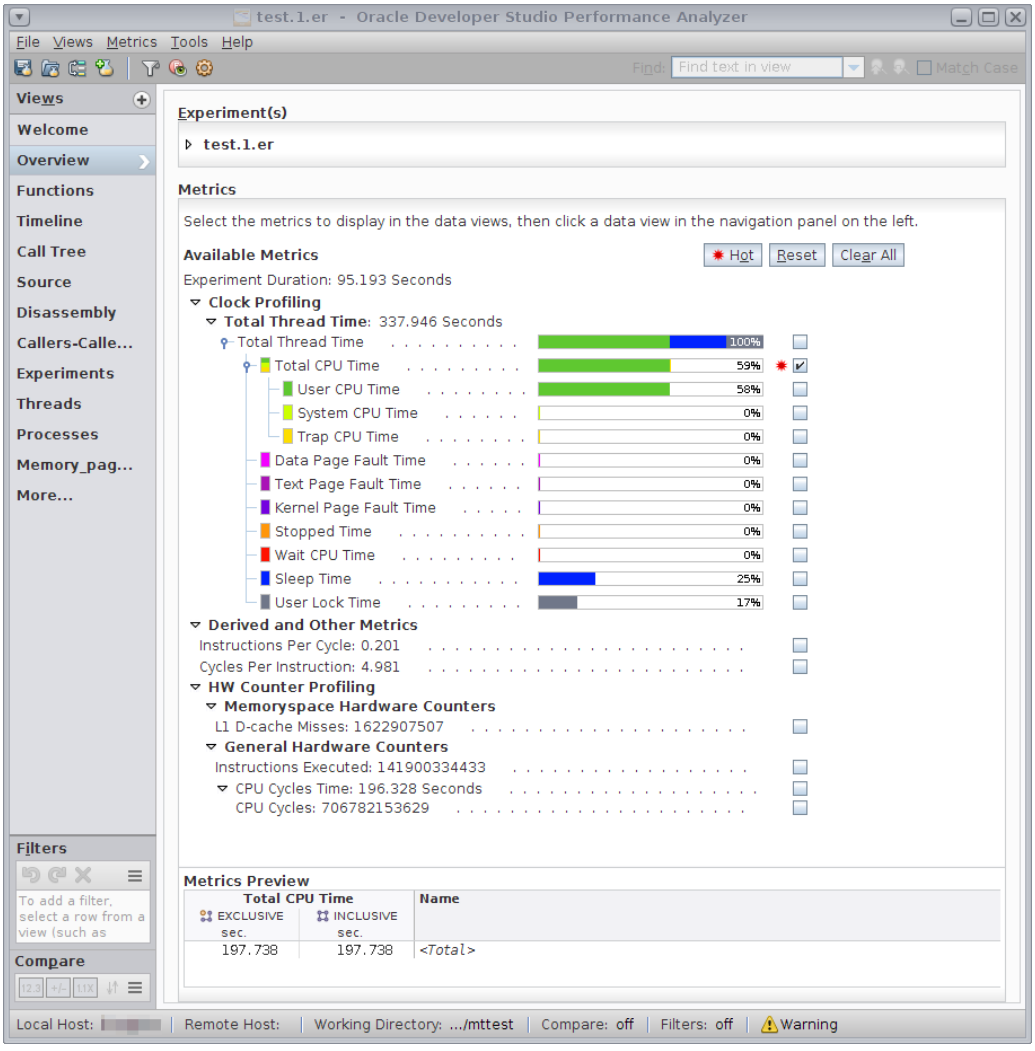
mttest のハードウェアカウンタプロファイリング実験の検査

このセクションでは、前のセクションの mttest サンプルコードで作成した実験のデータを調査する方法を示します。

次のようにして、mttest ディレクトリからパフォーマンスアナライザを起動し、実験をロードします。

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

実験が開くと、パフォーマンスアナライザの「概要」ページが表示されます。



「クロックプロファイリング」メトリックが最初に表示され、色付きのバーが含まれます。大部分のスレッド時間が「ユーザー CPU 時間」で消費されます。「休眠時間」または「ユーザーロック時間」で一定の時間が消費されます。

「派生およびその他のメトリック」グループは、`cycles` カウンタと `insts` カウンタの両方を記録した場合に存在します。派生メトリックは、それら 2 つのカウンタからのメトリックの比率を表します。「サイクル当たりの命令数」の値が高い、または「命令当たりのサイクル数」の値が低いことは、コードが比較的効率的であることを示唆します。反対に、「サイクル当たりの命令数」の値が低い、または「命令当たりのサイクル数」の値が高いことは、コードが比較的非効率であることを示唆します。

この実験では、「HW カウンタプロファイリング」グループは 2 つのサブグループ「メモリー領域ハードウェアカウンタ」および「一般ハードウェアカウンタ」を示します。「一般ハードウェアカウンタ」には「命令の実行」カウンタ (`insts`) が含まれます。収集したデータに `cycles` カウンタが含まれていた場合、「CPU サイクル」も「一般ハードウェアカウンタ」に含まれます。*正確な* `dcm` カウンタを備えるマシンでデータが収集された場合、「メモリー領域ハードウェアカウンタ」には「L1 データキャッシュミス」が含まれます。`dcm` カウンタが利用可能であったが正確なカウンタでない場合、「L1 データキャッシュミス」は「一般ハードウェアカウンタ」に含まれます。正確なカウンタとは、オーバーフローを発生させる命令の実行時に、そのオーバーフロー割り込みが伝達されるカウンタのことです。正確でないカウンタは、オーバーフローを発生させる命令を過ぎてから、可変量の「滑り」を伴って伝達されます。正確でないカウンタは、メモリー関連のカウンタであってもメモリー領域プロファイリングには使用できません。メモリー領域プロファイリングの詳細については、『Oracle Developer Studio 12.5: パフォーマンスアナライザ』の「データ領域プロファイリングとメモリー領域プロファイリング」を参照してください。

システムが `dcm` をサポートせず、`Makefile` を編集して `-h dcm` を削除した場合、「命令の実行」および「CPU サイクル」カウンタが表示されます。`Makefile` を編集して `-h dcm` と `-h cycles` の両方を削除した場合、「命令の実行」カウンタのみが表示されます。

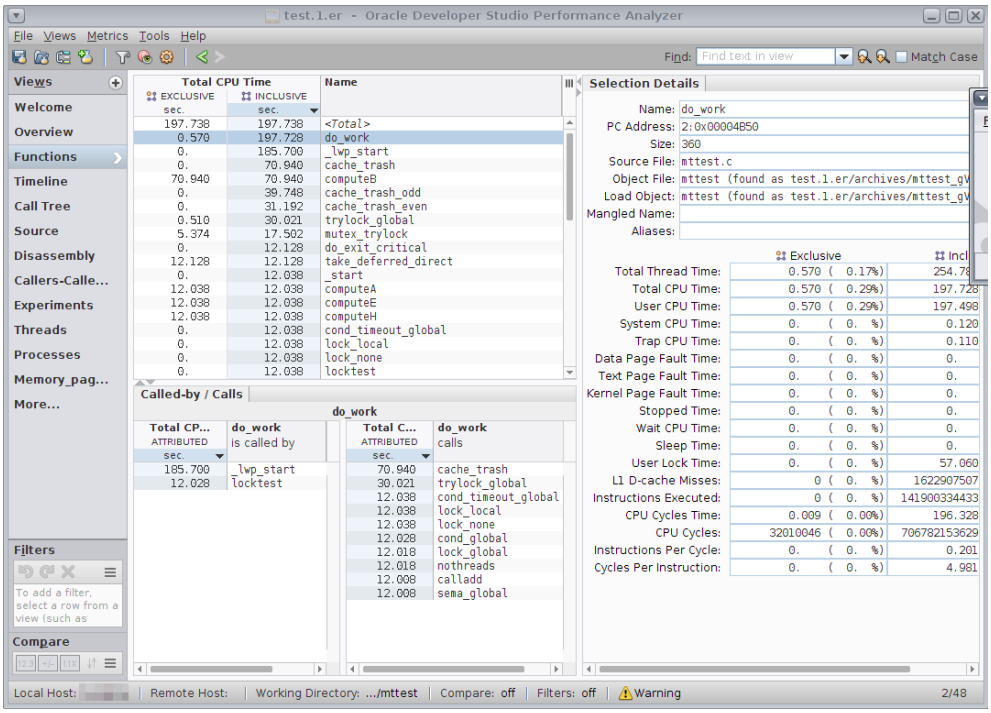
チュートリアルの以降のセクションで、これらのメトリックとその実装を調査します。

クロックプロファイリングデータの調査

このセクションでは、「概要」ページと「関数」ビューを「呼び出し元/呼び出し回数」パネルとともに使用してクロックプロファイリングデータを調査します。

1. 「概要」ページで、3 つの HW カウンタメトリックのチェックボックスを選択解除し、「CPU 時間合計」チェックボックスのみを選択したままにします。
2. 「関数」ビューに移動し、「包括的 CPU 合計時間」の列見出しを 1 回クリックして、包括的 CPU 合計時間に従ってソートします。

   この時点で、関数 `do_work()` がリストの先頭にあります。



3. `do_work()` 関数を選択し、「関数」ビューの下部にある「呼び出し元/呼び出し回数」パネルに注目します。

   `do_work()`は 2 つの場所から呼び出され、10 個の関数を呼び出します。

   `do_work()` が呼び出す 10 個の関数は 10 個の異なるタスクを表し、プログラムが実行した同期メソッドは関数ごとに異なります。`mttest` で作成される一部の実験では、わずかな時間を使用してほかのタスクからワークブロックをフェッチする 11 番目の関数が記録されることがあります。この関数はスクリーンショットには示されていません。

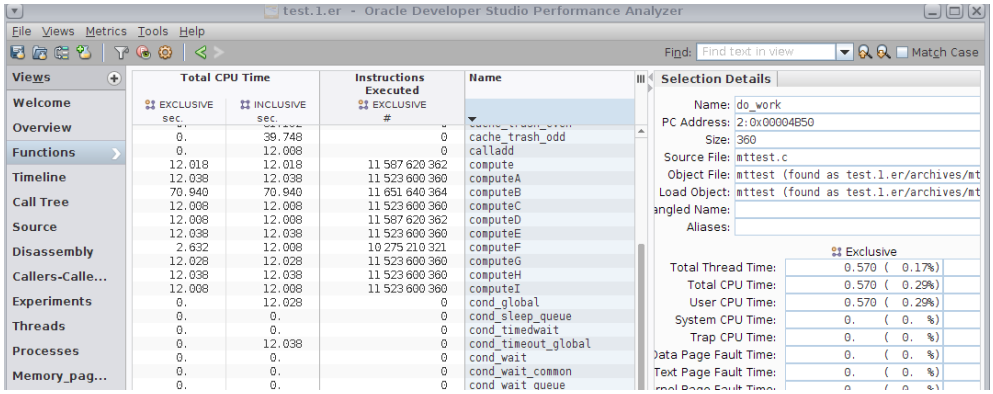
   `do_work()` はほとんどの場合、データを処理するスレッドの作成時に呼び出され、`_lwp_start()` から呼び出されるものとして示されます。`do_work()` は場合によっては、`locktest()` から呼び出されたあと、`nothreads()` という名前の 1 つのシングルスレッドタスクを呼び出します。

   パネルの「呼び出し」側では、最初の 2 つを除くすべての呼び出し先が、ほぼ同じ時間の「属性 CPU 合計」(12 秒以下) を示しています。

ハードウェアカウンタ命令プロファイリングのメトリックの理解

このセクションでは、一般ハードウェアカウンタを使用して、関数で実行される命令数を確認する方法を示します。

1. 「概要」ページを選択し、「一般ハードウェアカウンタ」の下にある「命令の実行」という名前の「HW カウンタプロファイリング」メトリックを有効にします。
2. 「関数」ビューに戻り、「名前」列見出しをクリックして、アルファベット順でソートします。
3. 下にスクロールして、`compute()`、`computeA()`、`computeB()` などの関数を探します。



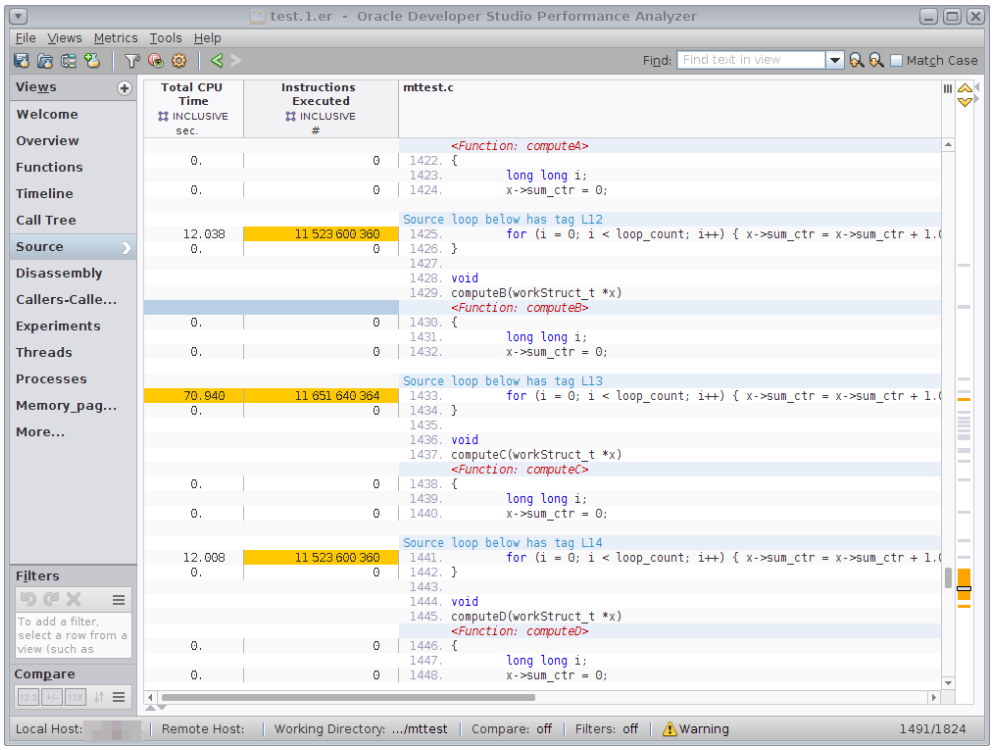
`computeB()` および `computeF()` を除くすべての関数で、「排他的 CPU 合計時間」および排他的命令の実行の数値がほとんど同じです。

4. `computeF()` を選択して「ソース」ビューに切り替えます。`computeF()` をダブルクリックすると、ワンステップでこれを実行できます。

computeF() は計算カーネルが異なり、1 を加算するために関数 addone() を呼び出す一方で、ほかの compute*() 関数は加算を直接実行します。ほかの関数とパフォーマンスが異なっているのは、そのことが原因です。

5. 「ソース」ビューを上下にスクロールして、すべての compute*() 関数を見てみます。

   computeB() を含むすべての compute*() 関数が、ほとんど同じ命令実行数を示しています。その中で、computeB() は特異な CPU 時間コストを示しています。



次のセクションでは、`computeB()` の CPU 時間合計がそのような高値を示す理由を探ります。

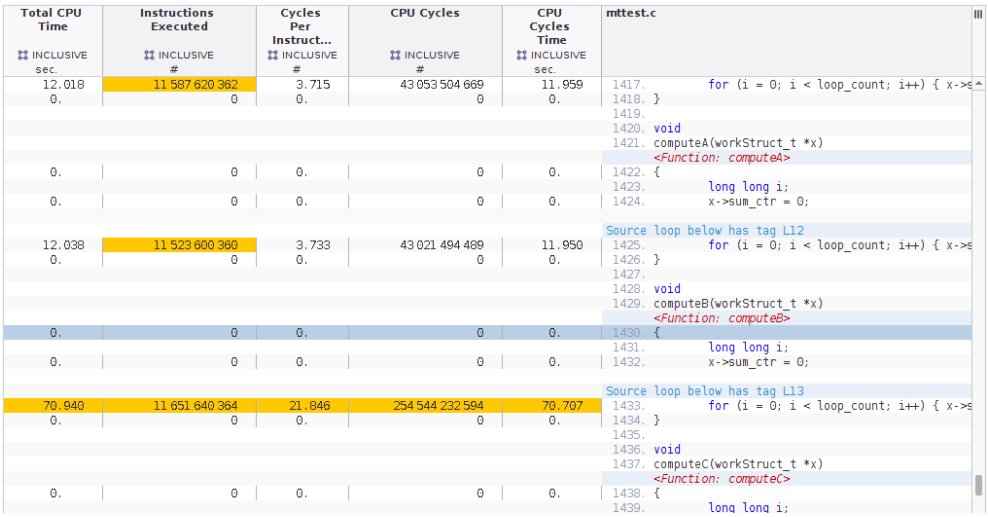
ハードウェアカウンタ CPU サイクルプロファイリングのメトリックの理解

チュートリアルのこのパートでは、`cycles` カウンタからデータが得られた実験が必要です。このカウンタをサポートしないシステムで行われた実験は、このセクションで使用できません。次のセクションである 81 ページの「キャッシュ競合メトリックとキャッシュプロファイリングメトリックの理解」に進んでください。

1. 「概要」ページを選択し、派生メトリック「命令当たりのサイクル数」と、「一般ハードウェアカウンタ」メトリックの「CPU サイクル時間」を有効にします。

   「CPU 時間合計」および「命令の実行」は選択したままにします。

2. `computeB()` の「ソース」ビューに戻ります。



包括的 CPU サイクルの時間と「包括的 CPU 合計」の時間はそれぞれの `compute*()` 関数でほぼ同じです。これは、クロックプロファイリングと「CPU サイクル」ハードウェアカウンタプロファイリングで同様のデータが得られていることを示します。

スクリーンショットでは、包括的 CPU サイクルおよび「包括的 CPU 合計」の時間は、`computeB()` を除いたそれぞれの `compute*()` 関数でおよそ 12 秒です。また実験では、包括的命令当たりのサイクル数 (CPI) が、`computeB()` に関してほかの `compute*()` 関数よりもずっと高くなります。これが示すように、`computeB()` は同数の命令を実行するために必要な CPU サイクルが多いことになり、ほかの関数よりも非効率です。

ここまでに観察したデータは、`computeB()` 関数とほかの関数の違いを示していますが、関数間で相違が生じる理由は示していません。このチュートリアルの次のパートでは、`computeB()` が異なっている理由を調査します。

キャッシュ競合メトリックとキャッシュプロファイリングメトリックの理解

チュートリアルのこのセクション以降では、正確な dcm ハードウェアカウンタからデータが得られた実験が必要です。正確な dcm カウンタをサポートしないシステムで行われた実験は、これ以降のチュートリアルで使用できません。

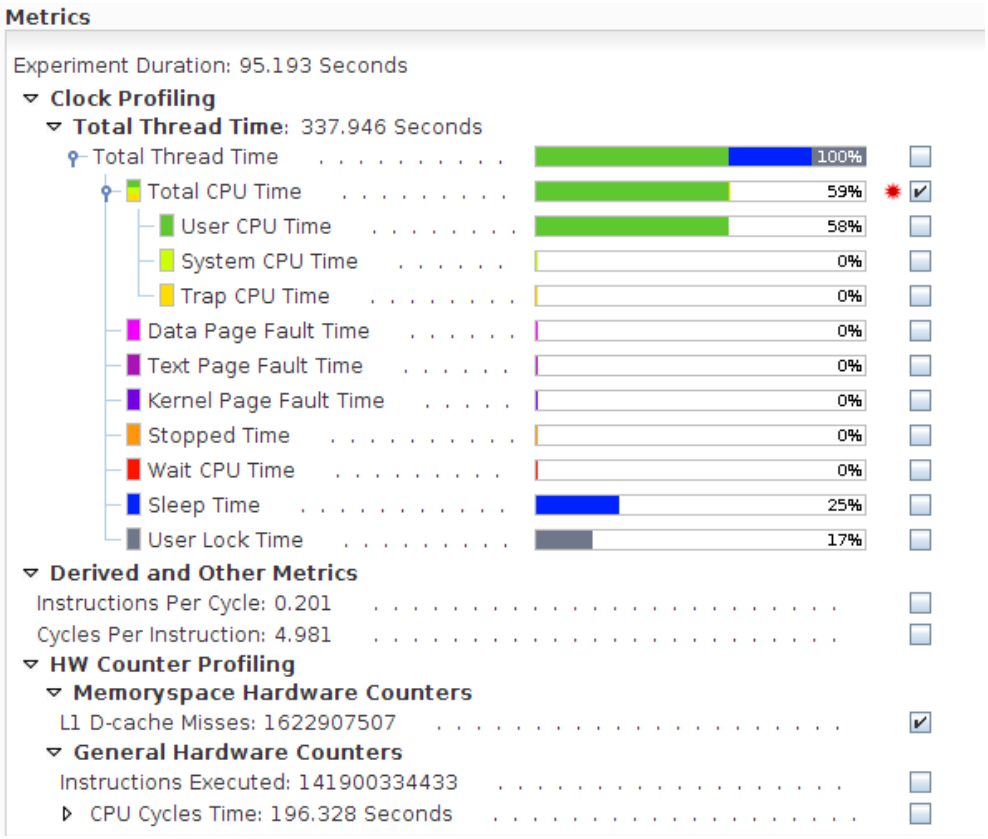
dcm カウンタはキャッシュミスをカウントしています。キャッシュミスとは、キャッシュにないメモリーアドレスを参照するロードおよびストアのことです。

次のいずれかの理由により、アドレスがキャッシュに存在しない場合があります。

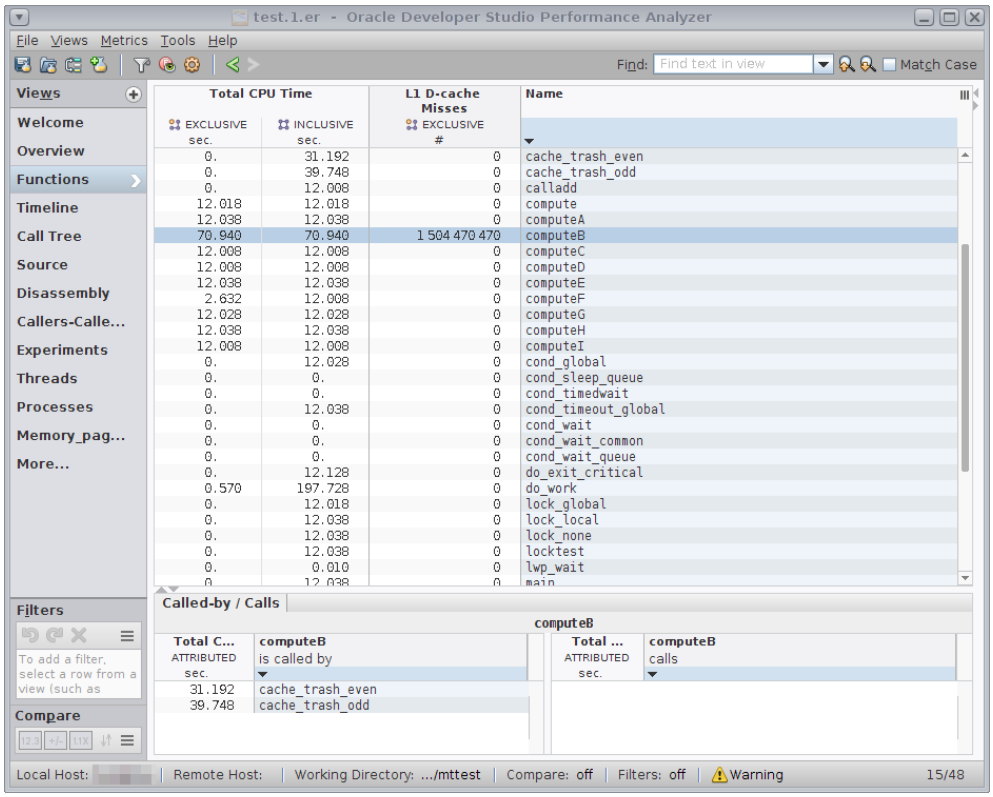
- 現在の命令が、その CPU からそのメモリー位置への最初の参照であるため。より正確には、現在の命令が、キャッシュ行を共有するメモリー位置のいずれかへの最初の参照であるため。
- スレッドがほかのメモリーアドレスを多く参照しすぎて、現在のアドレスがキャッシュからフラッシュされたため。これはキャパシティミスです。
- 同じキャッシュ行にマップするほかのメモリーアドレスをスレッドが参照したことにより、現在のアドレスがフラッシュされるため。これは競合ミスです。
- キャッシュ行の内部にあるアドレスに別のスレッドが書き込んだことにより、現在のスレッドのキャッシュ行がフラッシュされるため。これは共有ミスであり、2 種類の共有ミスのいずれかである可能性があります。
  - *真の共有*。現在のスレッドが参照しているアドレスと同じアドレスに、ほかのスレッドが書き込みました。真の共有が原因であるキャッシュミスは回避不可能です。
  - *偽共有*。現在のスレッドが参照しているアドレスとは異なるアドレスに、ほかのスレッドが書き込みました。偽共有が原因のキャッシュミスが発生する理由は、データ-ワードの粒度ではなくキャッシュ行の粒度でキャッシュハードウェアが動作することです。偽共有を回避するには、適切なデータ構造を変更し、各スレッドで参照される異なったアドレスが異なったキャッシュ行に配置されるようにします。

この手順では、関数 computeB() に影響を及ぼす偽共有のケースを検証します。

1. 「概要」に戻り、「L1 データキャッシュミス」のメトリックを有効にし、「命令当たりのサイクル数」のメトリックを無効にします。



2. ふたたび「関数」ビューに切り替え、`compute*()` ルーチンに注目します。

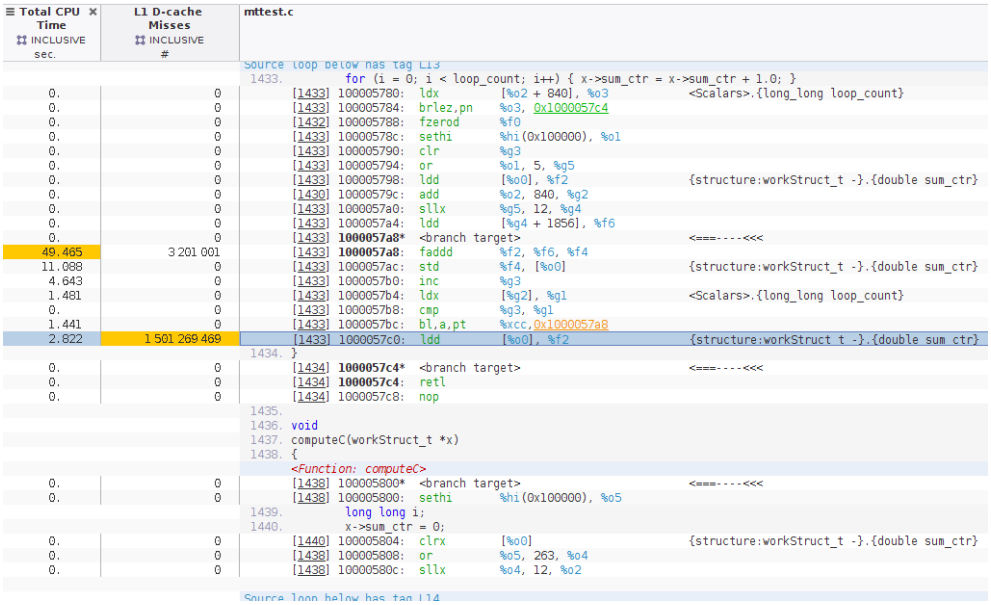


すべての `compute*()` 関数はほとんど同じ命令カウントを示しますが、`computeB()` はより高い「CPU 時間合計」を示し、排他的 L1 データキャッシュミスのカウントが大きい唯一の関数であることを思い出してください。

3. 「ソース」ビューに戻ると、`computeB()` ではキャッシュミスが 1 つの行ループに含まれています。
4. ナビゲーションパネルに「逆アセンブリ」タブがまだ表示されていない場合は、ナビゲーションパネルの上部で、「ビュー」ラベルの横にある「+」ボタンをクリックし、「逆アセンブリ」のチェックボックスにチェックマークを付けて、このビューを追加します。

   「L1 データキャッシュミス」の数値が高いロード命令の行が見つかるまで、「逆アセンブリ」ビューをスクロールします。

ヒント - 「逆アセンブリ」などのビューの右マージンに含まれるショートカットをクリックすると、メトリックの高い行 (ホットライン) にジャンプできます。マージンの上部にある「次のホットライン」下向き矢印、またはゼロ以外のメトリックのマーカーをクリックすると、メトリックの値が顕著である行にすばやくジャンプできます。



SPARC システムでは、-xhwcprof でコンパイルした場合、ロードおよびストアには、命令がダブルワード (workStruct_t データ構造内の sum_ctr) を参照していることを示す構造情報が注釈として付けられます。アドレスが次の行と同じであり、命令が <branch target> である行も見えます。そのような行は次のアドレスが分岐のターゲットであることを示しており、<branch target> より上の命令を実行することなく、ホットとして指摘された命令にコードが到達した可能性があります。

-xhwcprof は x86 ではサポートされていないため、x86 システムではロードおよびストアに注釈が付かず、<branch target> 行は表示されません。

5. 「関数」ビューと「逆アセンブリ」ビューを切り替えながら、さまざまな compute*() 関数を選択します。

   すべての compute*() 関数で、「命令の実行」のカウントが高い命令は同じ構造フィールドを参照します。

computeB() は、同じ数の命令を実行するにもかかわらず、ほかの関数よりも非常に長い時間がかかっており、キャッシュミスが発生する唯一の関数であることがわかりました。キャッシュミスのあるロードはキャッシュヒットするロードよりも計算に要するサイクルが多いため、命令の実行サイクル数の増加はキャッシュミスが原因です。

computeB() を除いたすべての compute*() 関数について、各スレッドから引数で指される構造体 workStruct_t 内のダブルワードフィールド sum_ctr は、そのスレッドに対する Workblk の内部に含まれています。Workblk 構造体は隣接して割り当てられますが、十分な大きさを持っているため、各構造体内のダブルワードはキャッシュ行を共有するには離れすぎています。

computeB() については、スレッドからの workStruct_t 引数はその構造体の連続的なインスタンスであり、その長さは 1 ダブルワード long しかありません。結果として、異なるスレッドによって使用されるダブルワードがキャッシュ行を共有し、あるスレッドからのストアが別のスレッド内のキャッシュ行を無効化する原因になります。キャッシュミスカウントの高さはそのことが原因であり、「CPU 時間合計」および「CPU サイクル」メトリックの高さはキャッシュ行の遅延リフィルが原因です。

この例では、スレッドによって格納されるデータワードは重複しませんが、キャッシュ行を共有します。このパフォーマンスの問題は「偽共有」と呼ばれます。スレッドが同じデータワードを参照する場合、それは真の共有です。これまでに見てきたデータは、偽共有と真の共有を区別しません。

偽共有と真の共有の違いについては、このチュートリアルの最後のセクションで調査します。

偽共有の検出

チュートリアルのこのパートは、「L1 データキャッシュミス」dcm カウンタが正確であるシステムのみが対象です。そのようなシステムには SPARC-T4、SPARC-T5、SPARC-M5、SPARC-M6 などが含まれます。正確な dcm カウンタを備えていないシステムで記録された実験は、このセクションで使用できません。

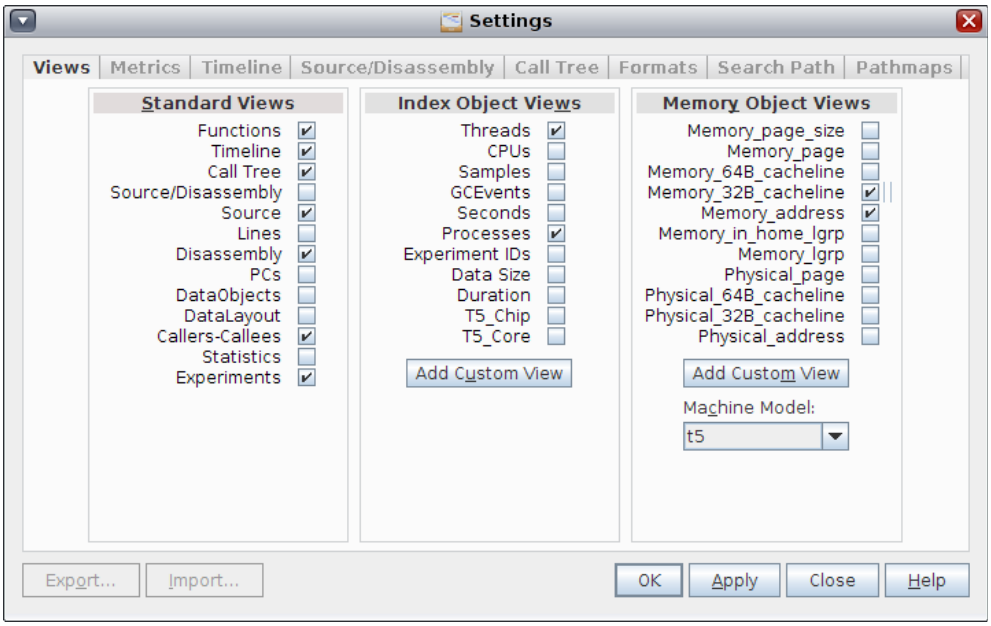
この手順では、フィルタリングとともに「インデックスオブジェクト」ビューおよび「メモリーオブジェクト」ビューを使用する方法を示します。

正確なメモリー関連カウンタを備えたシステムで実験を作成すると、*マシンモデル*が実験に記録されます。マシンモデルは、そのマシンのメモリーサブシステム内のさまざまなコンポーネントへのアドレスのマッピングを表します。パフォーマンスアナライザまたは er_print で実験をロードすると、マシンモデルが自動的にロードされます。

このチュートリアルのスクリーンショットで使用する実験は SPARC T5 システムで記録されたものであり、そのマシン用の t5 マシンモデルが自動的に実験とともにロードされます。マシンモデルはインデックスオブジェクトおよびメモリーオブジェクトのデータビューを追加します。

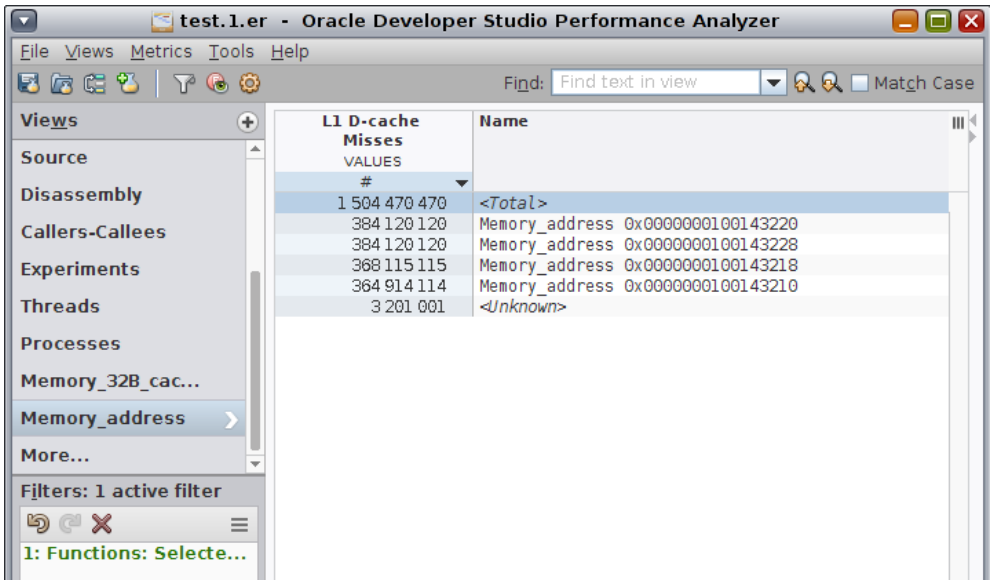
1. 「関数」ビューに移動して computeB() を選択し、右クリックして「フィルタの追加: 選択した関数を含むスタックのみを含める」を選択します。

   フィルタリングによって、computeB() 関数のパフォーマンスと、その関数で発生するプロファイルイベントに焦点を当てることができます。
2. ツールバーの「設定」ボタンをクリック、または「ツール」->「設定」を選択して「設定」ダイアログを開き、そのダイアログで「ビュー」タブを選択します。



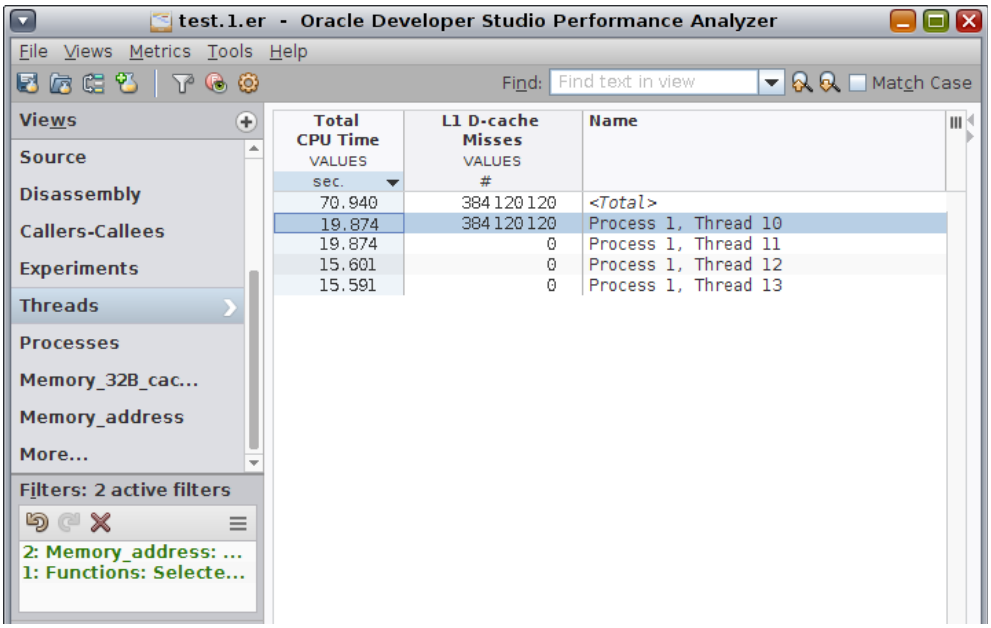
右側のパネルには「メモリーオブジェクトビュー」というラベルが付いており、SPARC T5 マシンのメモリーサブシステム構造を表すデータビューのリストを示します。

3. 「Memory_address」および「Memory_32B_cacheline」のチェックボックスを選択して「了解」をクリックします。
4. 「ビュー」ナビゲーションパネルで「Memory_address」ビューを選択します。



この実験では、4 つの異なるアドレスでキャッシュミスが発生していることがわかります。

5. アドレスのうちの 1 つを選択し、右クリックして「フィルタの追加: 選択した項目を含むイベントのみを含める」を選択します。
6. 「スレッド」ビューを選択します。



前のスクリーンショットでわかるように、そのアドレスでキャッシュミスが発生しているスレッドは 1 つだけです。

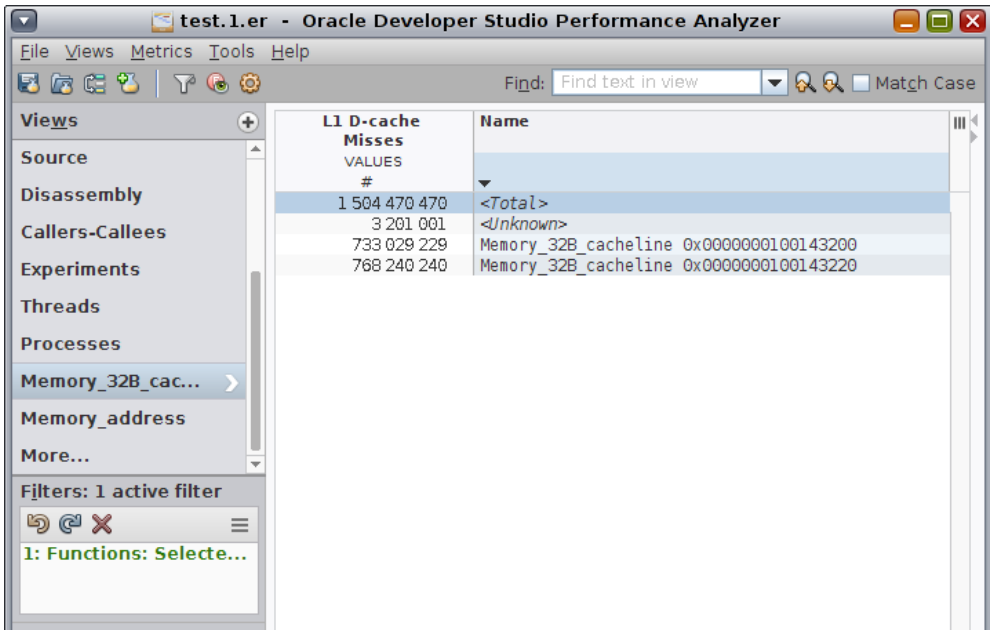
7. ビュー内で右クリックし、コンテキストメニューから「フィルタアクションを元に戻す」を選択して、アドレスフィルタを削除します。

   または、「アクティブなフィルタ」パネル内の「フィルタアクションを元に戻す」ボタンを使用すると、フィルタを削除できます。

8. 「Memory_address」ビューに戻り、ほかのアドレスを選択してフィルタを適用し、関連するスレッドを「Threads」ビューでチェックします。

   このようにして、フィルタを適用および適用解除する、また「Memory_address」ビューと「Threads」ビューを切り替えることによって、4 つのスレッドと 4 つのアドレスの間に 1 対 1 の関係があることを確認できます。つまり、4 つのスレッドはアドレスを共有しません。

9. 「ビュー」ナビゲーションパネルで「Memory_32B_cacheline」ビューを選択します。



「アクティブなフィルタ」パネルで、関数 `computeB()` のフィルタのみがアクティブであることを確認します。フィルタは「関数: 選択した関数」と表示されます。この時点で、どのアドレスフィルタもアクティブではありません。

2 つの 32 バイトキャッシュ行で 4 つのスレッドのキャッシュミスが発生していること、また 4 つのスレッドそれぞれのアドレスを確認できます。4 つのスレッドがアドレスを共有しないことは以前に確認しましたが、ここでは 4 つのスレッドがキャッシュ行を共有することを確認できます。

偽共有は診断が非常に難しい問題であり、SPARC T5 チップでは、Oracle Developer Studio パフォーマンスアナライザを使用することでその診断が可能になります。

マルチスレッドプログラムでの同期トレース

このチュートリアルの内容は次のとおりです。

同期トレースのチュートリアルについて

このチュートリアルでは、マルチスレッドプログラムに対してパフォーマンスアナライザを使用し、クロックプロファイリングおよび同期トレースのデータを検証する方法を示します。

「概要」ページを使用して、どのパフォーマンスメトリックがハイライトされているかをすばやく確認したり、データビューに表示するメトリックを変更したりします。「関数」ビュー、「呼び出し元-呼び出し先」ビュー、および「ソース」ビューを使用してデータを調査します。チュートリアルでは 2 つの実験を比較する方法も示します。

チュートリアルは同期トレースデータの理解に役立ち、そのデータをクロックプロファイリングデータと関連付ける方法を説明します。

記録する実験で目にするデータは、ここで示されているものとは異なります。チュートリアルのスクリーンショットに使用した実験は、Oracle Solaris 11.3 を実行する SPARC T5 システムで記録されました。Oracle Solaris または Linux を実行する x86 システムからのデータは異なります。さらに、データ収集には統計的な性質があり、同じシステムおよび OS で実行する場合でも実験ごとに異なります。

パフォーマンスアナライザの実際のウィンドウ構成は、スクリーンショットと厳密に一致しない場合があります。パフォーマンスアナライザでは、ウィンドウのコンポーネント間の区切りバーをドラッグしたり、コンポーネントを縮小したり、ウィンドウのサイズを変更したりできます。パフォーマンスアナライザはその構成を記録し、次回の実行時に同じ構成を使用します。チュートリアルで示しているスクリーンショットの取得過程で、多くの構成を変更しました。

mttest プログラムについて

mttest プログラムは、ダミーデータに対してさまざまな同期オプションを実行する単純なプログラムです。プログラムはさまざまなタスクを実装し、各タスクは次の同じ基本アルゴリズムを使用します。

- 複数のワークブロック (デフォルトで 4 つ) をキューに入れます。
- 作業を処理する複数のスレッド (これもデフォルトで 4 つ) を生成します。
- 各タスクで、特定の同期プリミティブを使用して、ワークブロックへのアクセスを制御します。
- 同期のあと、ブロックに対する作業を処理します。

各タスクは異なる同期メソッドを使用します。mttest のコードは各タスクを順番に実行します。

同期トレースについて

同期トレースは、mutex_lock()、pthread_mutex_lock()、sem_wait() など、同期用のさまざまなライブラリ関数に割り込むことによって実装されます。pthread および Oracle Solaris 同期呼び出しの両方がトレースされます。

ターゲットプログラムがこれらの関数のいずれかを呼び出すと、呼び出しはデータコレクタによってインターセプトされます。現在の時間、ロックのアドレス、その他のデータがキャプチャーされたあと、割り込みルーチンが実際のライブラリルーチンを呼び出します。実際のライブラリルーチンが戻ると、データコレクタがふたたび時間を読み取り、終了時間と開始時間の差を計算します。その差がユーザー指定のしきい値を超える場合、イベントが記録されます。時間がしきい値を超えない場合、イベントは記録されません。どちらの場合も、実際のライブラリルーチンからの戻り値が呼び出し元に返されます。

イベントを記録するかどうかの決定に使用されるしきい値は、collect コマンドの -s オプションを使用して設定できます。パフォーマンスアナライザを使用して実験を収集する場合、「アプリケーションのプロファイル」ダイアログの「同期待ちトレース」の「最小遅延」でしきい値を指定できます。しきい値はマイクロ秒数に、またはキーワード calibrate または on に設定できます。calibrate または on を使用する場合、データコレクタは、競合しない相互排他ロックを獲得するのにかかる時間を調べ、しきい値をその値の 5 倍に設定します。0 または all のしきい値を指定すると、すべてのイベントが記録されます。

このチュートリアルでは、2 つの実験で同期待ちトレースを記録します。一方の実験では測定に基づいたしきい値を使用し、もう一方の実験ではしきい値をゼロにします。どちらの実験にもクロックプロファイリングが含まれます。

mttest サンプルコードの設定

始める前に:

コードの取得および環境の設定については、次を参照してください。

- 10 ページの「チュートリアルのサンプルコードの入手」
- 11 ページの「環境をチュートリアル用に設定」

パフォーマンスアナライザについてよく理解するために、「C プロファイリングの概要」の入門チュートリアルを詳細に検討することもできます。

このチュートリアルでは、「マルチスレッドプログラムでのハードウェアカウンタプロファイリング」のチュートリアルと同じ mttest のコードを使用します。このチュートリアル用の独立したコピーを作成してください。

1. 次のコマンドで、mttest ディレクトリの内容をプライベート作業領域にコピーします。

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

   mydirectory は使用する作業ディレクトリです。
2. その作業ディレクトリのコピーを変更します。

   ```
   % cd directory/mttest
   ```
3. ターゲットの実行可能ファイルをビルドします。

   ```
   % make clobber

   % make
   ```

   注記 - clobber サブコマンドコマンドは、このディレクトリで以前に make を実行した場合にのみ必要ですが、どの場合にも安全に使用できます。

   make を実行すると、このディレクトリにはチュートリアルで使用するターゲットアプリケーションである、mttest という名前の C プログラムが含まれます。

ヒント - 必要に応じて、次のために Makefile を編集できます。デフォルトの Oracle Developer Studio コンパイラではなく GNU コンパイラを使用します。デフォルトの 64 ビットではなく 32 ビットでビルドします。各種のコンパイラフラグを追加します。

同期トレースのチュートリアル用に mttest からデータを収集する

データを収集するもっとも簡単な方法は、mttest ディレクトリで次のコマンドを実行することです。

```
% make syncperf
```

Makefile の syncperf ターゲットは、collect コマンドを 2 回起動し、2 つの実験を作成します。

注記 - このチュートリアルでは、コンパイルおよびデータ収集に前出の入門チュートリアルよりも時間がかかることがあります。

2 つの実験は test.1.er および test.2.er という名前であり、それぞれ、同期トレースデータおよびクロックプロファイルデータを含みます。最初の実験では、collect の -s on オプションを指定し、測定に基づいたしきい値を使用してイベントを記録します。2 番目の実験では、collect の -s all オプションを指定し、しきい値をゼロに設定してすべてのイベントを記録します。どちらの実験でも、-p on オプションによってクロックプロファイリングを有効にします。

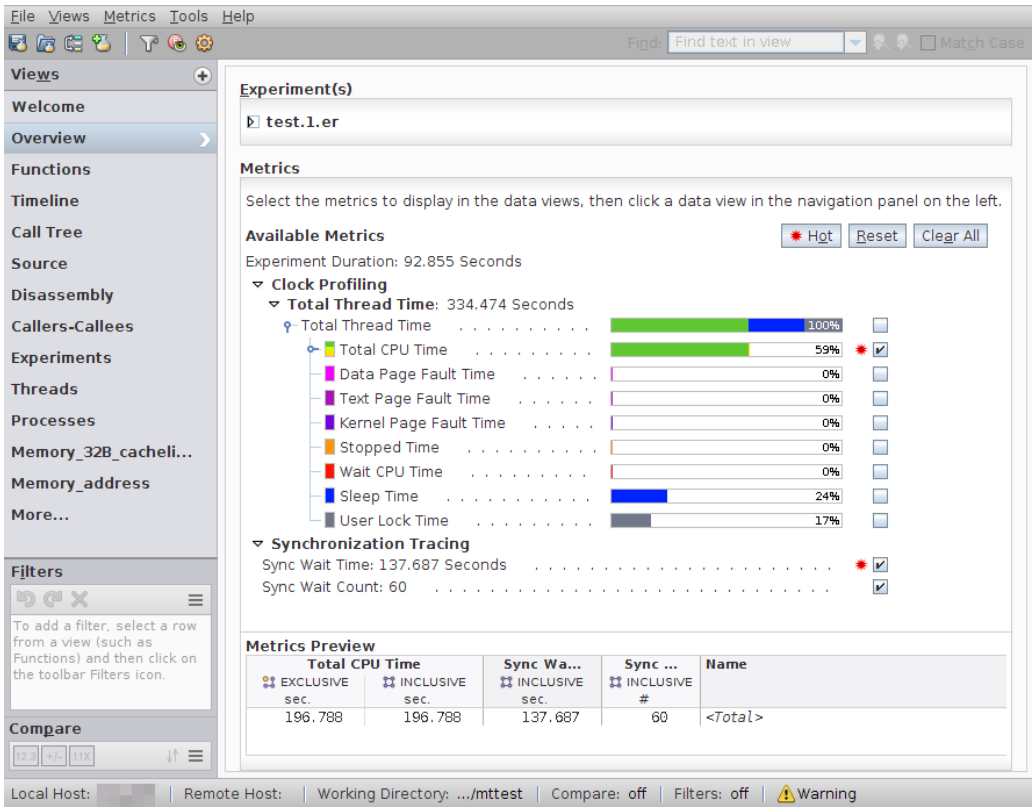
mttest での同期トレース実験の検証

このセクションでは、前のセクションの mttest サンプルコードで作成した実験のデータを調査する方法を示します。

次のようにして、mttest ディレクトリからパフォーマンスアナライザを起動し、最初の実験をロードします。

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

実験が開くと、パフォーマンスアナライザの「概要」ページが表示されます。

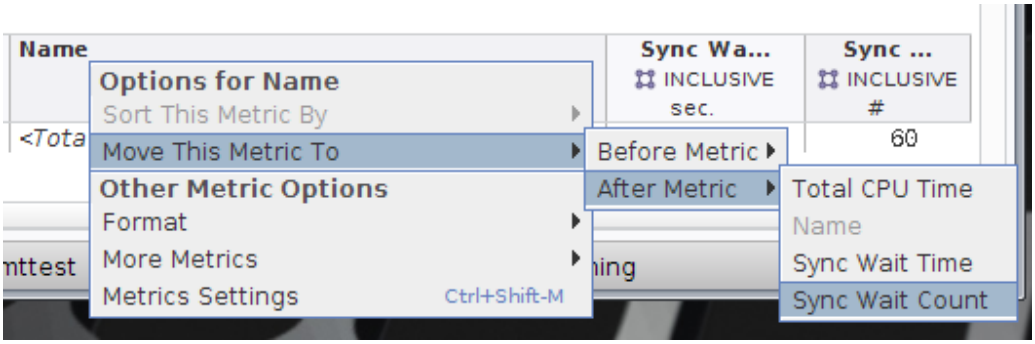


「クロックプロファイリング」メトリックが最初に表示され、色付きのバーが含まれます。大部分のスレッド時間が「ユーザー CPU 時間」で消費されます。「休眠時間」または「ユーザーロック時間」で一定の時間が消費されます。

「同期トレース」メトリックは、「同期待ち時間」および「同期待ちカウント」の 2 つのメトリックを含む 2 番目のグループに表示されます。

注記 - 「同期待ち時間」および「同期待ちカウント」メトリックが表示されていない場合は、右へスクロールしてこれらの列を表示する必要があります。メトリックの列ヘッダーを右クリックし、「このメトリックの移動先」を選択し、ほかのメトリックとの関連でこのメトリックを表示するのに便利な場所を選択することによって、任意の列をより便利な場所に移動できます。

次の例では、「名前」列を「同期待ちカウント」メトリックのあとに移動します。

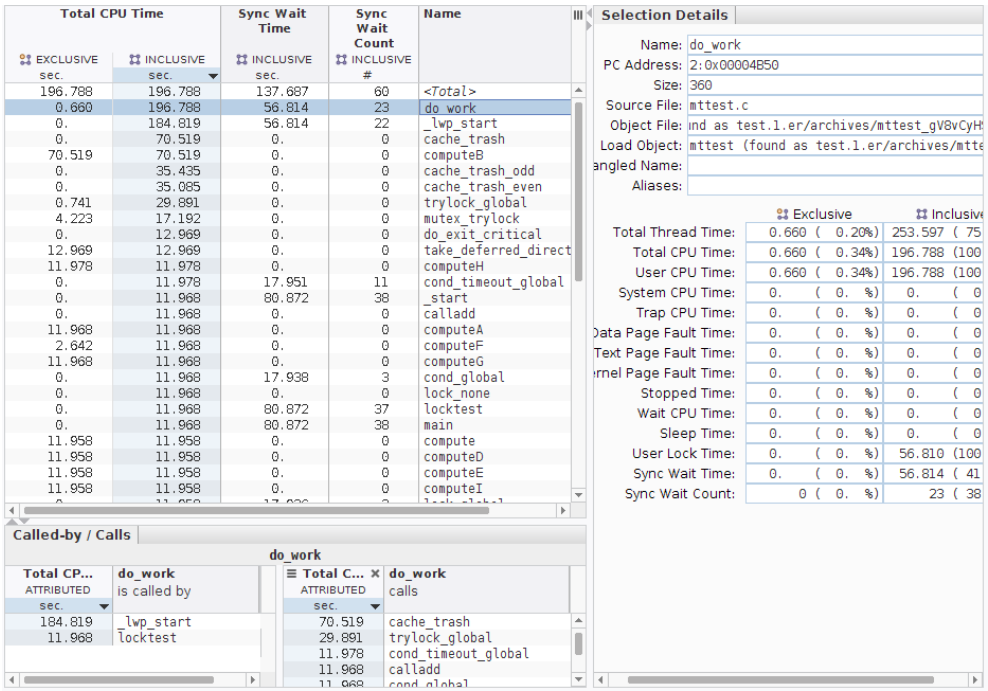


チュートリアルの以降のセクションで、これらのメトリックとその実装を調査できます。

同期トレースの理解

このセクションでは、同期トレースのデータと、そのデータをクロックプロファイリングデータと関連付ける方法について説明します。

1. 「関数」ビューに移動し、列見出し「包括的 CPU 合計」をクリックし、包括的 CPU 合計時間に従ってソートします。
2. リストの先頭にある `do_work()` 関数を選択します。

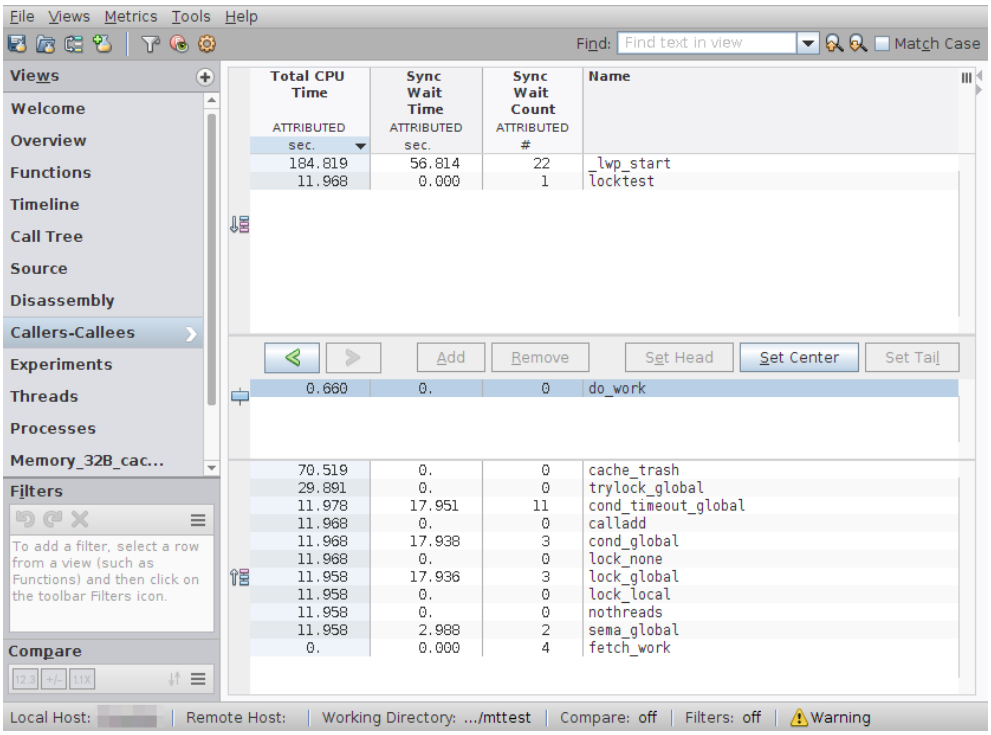


3. 「関数」ビューの下部にある「呼び出し元/呼び出し回数」パネルを見ると、`do_work()` は 2 つの場所から呼び出され、10 個の関数を呼び出しています。

   `do_work()` はほとんどの場合、データを処理するスレッドの作成時に呼び出され、`_lwp_start()` から呼び出されるものとして示されます。`do_work()` は場合によっては、`locktest()` から呼び出されたあと、`nothreads()` という名前の 1 つのシングルスレッドタスクを呼び出します。

   `do_work()` が呼び出す 10 個の関数は 10 個の異なるタスクを表し、各タスクはプログラムが実行した異なる同期メソッドを使用します。`mttest` で作成される一部の実験では、わずかな時間を使用してほかのタスクからワークブロックをフェッチする 11 番目の関数が記録されることがあります。この関数 `fetch_work()` は、前のスクリーンショットで「呼び出し」パネルに表示されています。

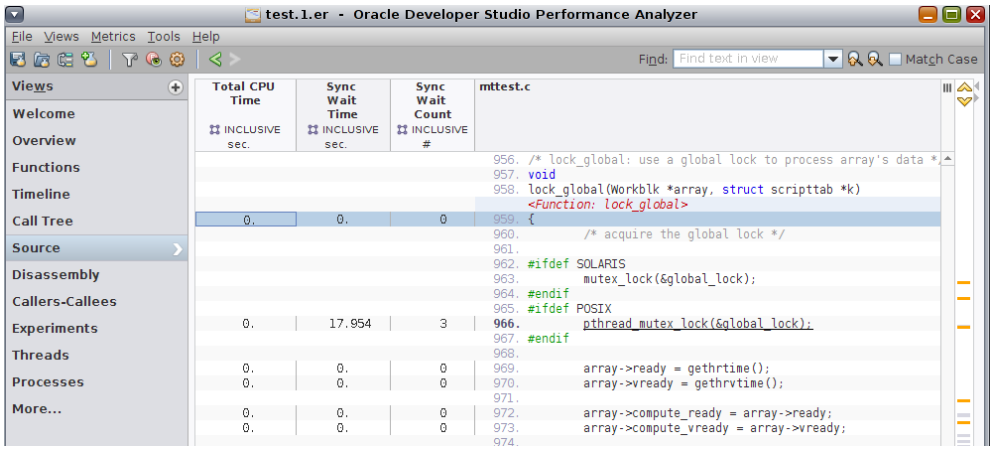
   「呼び出し」パネルの最初の 2 つの呼び出し先を除き、すべての呼び出し先はほぼ同じ時間の「属性 CPU 合計時間」(10.6 秒以下) を示しています。
4. 「呼び出し元-呼び出し先」ビューに切り替えます。



「呼び出し元-呼び出し先」ビューには「呼び出し元/呼び出し回数」パネルと同じ呼び出し元と呼び出し先が表示されますが、「属性同期待ち時間」など、「概要」ページで選択されたほかのメトリックも表示されます。

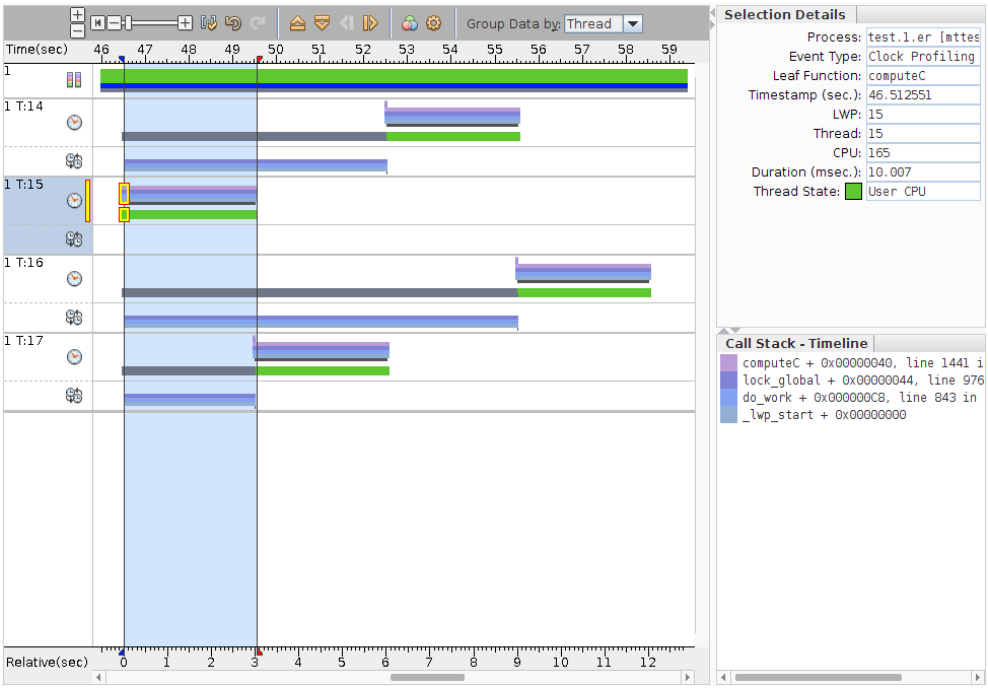
2 つの関数 `lock_global()` および `lock_local()` を探すと、これらの関数が示している「属性 CPU 合計時間」はほとんど同じですが、「属性同期待ち時間」は大きく異なっています。

5. `lock_global()` 関数を選択して「ソース」ビューに切り替えます。



すべての「同期待ち時間」は、「CPU 時間合計」が 0 である `pthread_mutex_lock (&global_lock)` の呼び出しがある行で費やされています。関数名から推測されるように、このタスクを実行する 4 つのスレッドはすべて、グローバルロックを 1 つずつ獲得するときに各自の処理を実行します。

6. 「関数」ビューに戻って `lock_global()` を選択し、「フィルタ」アイコン  をクリックして「フィルタの追加: 選択した関数を含むスタックのみを含める」を選択します。
7. 「タイムライン」ビューを選択すると、4 つのスレッドが見えます。
8. タイムライン内でイベントの発生した領域を強調表示し、右クリックして「ズーム」->「選択した時間範囲にズーム」を選択することによって、関心のある領域にズームインします。
9. 4 つのスレッドを調べて、待機に費やされた時間と計算に費やされた時間を比較します。



注記 - 実験では、複数のスレッドが異なる時点で実行中および待機中になっている可能性があります。

ロックを取得するための最初のスレッド (スクリーンショットの T:15) は約 2.97 秒を費やしたあと、ロックを諦めます。そのスレッドのスレッド状態バーは緑色であり、その時間のすべてが「ユーザー CPU 時間」で消費され、「ユーザーロック時間」では消費されなかったことがわかります。また、同期トレースの呼び出しスタックを表す 2 番目のバーには マークが付き、このスレッドの呼び出しスタックがないことを示しています。

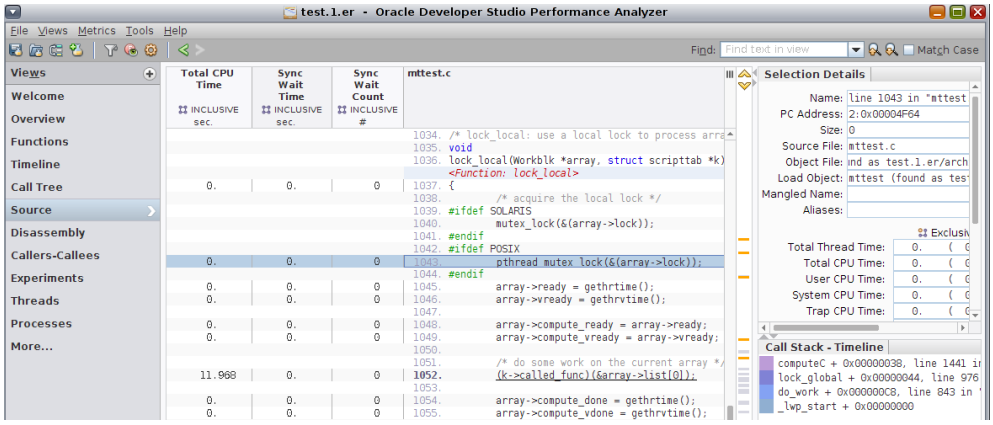
2 番目のスレッド (スクリーンショットの T:17) は「ユーザーロック時間」で 2.99 秒待ったあと、2.90 秒にわたって計算し、ロックを諦めます。同期トレースの呼び出しスタックは「ユーザーロック時間」と同時期です。

3 番目のスレッド (T:14) は「ユーザーロック時間」で 5.98 秒待ったあと、2.95 秒にわたって計算し、ロックを諦めます。

最後のスレッド (T:16) は「ユーザーロック時間」で 8.98 秒待ったあと、2.84 秒にわたって計算します。合計の計算は 2.97+2.90+2.95+2.84、つまり 11.7 秒でした。

合計の同期待ちは 2.99 + 5.98 + 8.98、つまり 17.95 秒でした。このことは「関数」ビューで確認できます (17.954 秒と報告されています)。

10. 「アクティブなフィルタ」パネルの「X」をクリックしてフィルタを削除します。
11. 「関数」ビューに戻り、関数 `lock_local()` を選択して「ソース」ビューに切り替えます。



`pthread_mutex_lock(&array->lock)` の呼び出しがある行 (スクリーンショットの行 1043) で、「同期待ち時間」は 0 です。その理由は、ロックがワークブロックにローカルであるため、競合がなく、4 つのスレッドすべてが同時に計算することです。

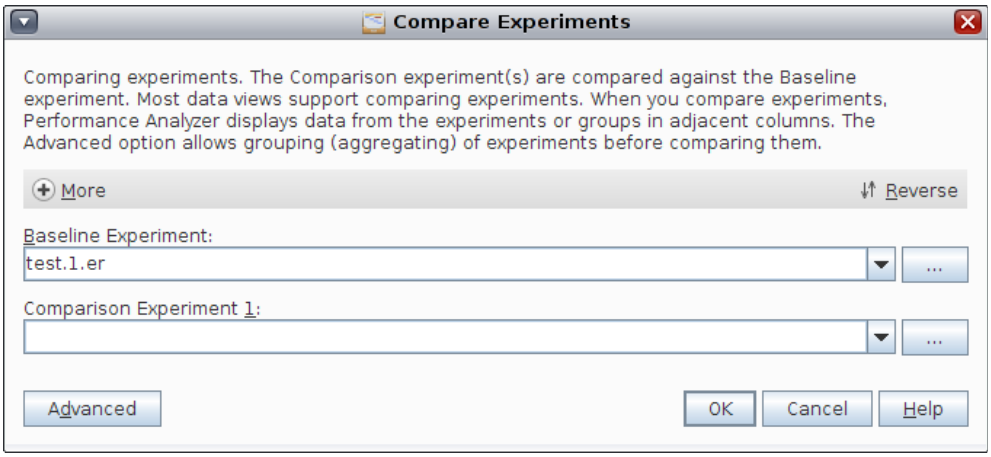
ここまで見てきた実験は、測定に基づくしきい値を使用して記録されたものです。次のセクションでは、`make` コマンドの実行時にしきい値をゼロにして記録された 2 番目の実験と比較します。

同期トレースのある 2 つの実験の比較

このセクションでは 2 つの実験を比較します。`test.1.er` 実験は、測定に基づくイベント記録しきい値を使用して記録されました。`test.2.er` 実験は、`mttest` プログラムの実行で発生したすべての同期イベントが含まれるよう、しきい値をゼロにして記録されました。

1. ツールバーにある「実験を比較」ボタン  をクリック、または「ファイル」->「実験を比較」を選択します。

   「実験を比較」ダイアログボックスが開きます。

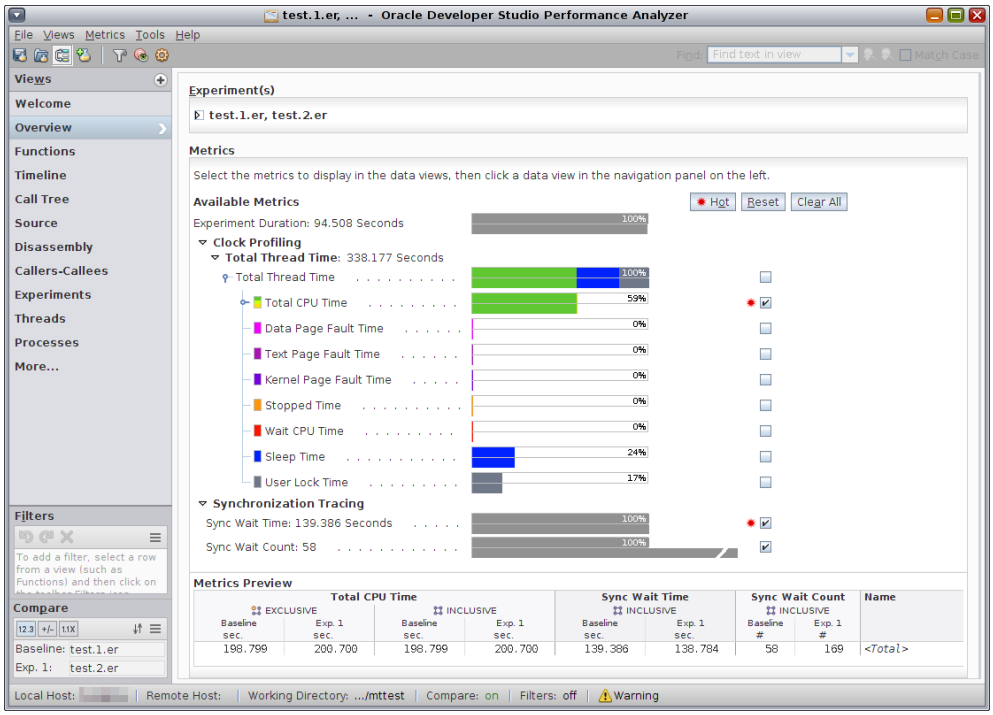


すでに開いている `test.1.er` 実験は「ベースライン」グループに表示されます。ベースライン実験と比較する実験を「比較グループ」パネルに追加する必要があります。

実験の比較、およびベースラインと比較する複数の実験の追加に関する詳細は、ダイアログボックスの「ヘルプ」ボタンをクリックしてください。

2. 「比較実験 1」の隣にある「...」ボタンをクリックし、「実験の選択」ダイアログで `test.2.er` 実験を開きます。
3. 「実験を比較」ダイアログで「了解」をクリックし、2 番目の実験をロードします。

   両方の実験のデータが含まれた状態で、「概要」ページがふたたび開きます。



「クロックプロファイリング」メトリックは、それぞれの実験に対応する 2 本の色付きバーをメトリックごとに表示します。`test.1.er` ベースライン実験のデータは上側です。

データバーにマウスカーソルを合わせると、ベースライングループおよび比較グループのデータと、数値および % で表されたグループ間の差がポップアップテキストで表示されます。

2 番目の実験では、記録された「CPU 時間合計」がわずかに多い一方、「同期待ちカウント」はほぼ 3 倍になっています。

4. 「関数」ビューに切り替え、「包括的同期待ち時間カウント」列の下の「ベースライン」というサブ列ヘッダーをクリックして、最初の実験のイベント数で関数をソートします。

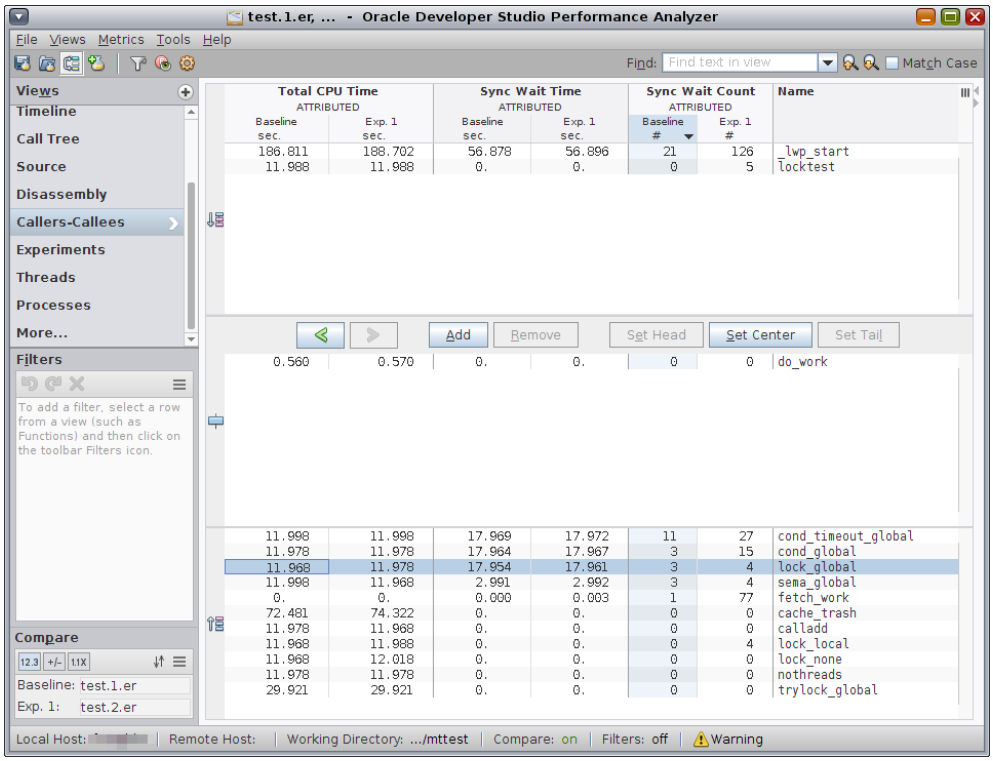
Total CPU Time				Sync Wait Time		Sync Wait Count		Name
EXCLUSIVE		INCLUSIVE		INCLUSIVE		INCLUSIVE		
Baseline sec.	Exp. 1 sec.	Baseline sec.	Exp. 1 sec.	Baseline sec.	Exp. 1 sec.	Baseline #	Exp. 1 #	
198.799	200.700	198.799	200.700	139.386	138.784	58	169	<Total>
0.560	0.570	198.799	200.690	56.878	56.896	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 と test.2.er の差がもっとも大きいのは do_work() であり、これには、lock_global() や lock_local() など、この関数が直接または間接的に呼び出すすべての関数で生じた差が含まれます。

ヒント - 比較形式を変更すると、さらに簡単に差を比較できます。ツールバーの「設定」ボタンをクリックし、「書式」タブを選択し、「比較形式」から「デルタ」を選択します。変更を適用したあと、test.2.er のメトリックは、test.1.er のメトリックからのプラスまたはマイナスの差として表示されます。前のスクリーンショットで、選択された pthread_mutex_lock() 関数は、test.2.er の「包括的同期待ちカウント」列で +88 を示します。

---

5. 「呼び出し元-呼び出し先」ビューを選択します。



2 つの呼び出し元 `lock_global()` および `lock_local()` に注目します。

`lock_global()` 関数は、`test.1.er` では「属性同期待ちカウント」に対して 3 つのイベントを示しますが、`test.2.er` では 4 つのイベントを示します。理由は、`test.1.er` ではロックを獲得するための最初のスレッドがストールしなかったため、イベントが記録されなかったことです。`test.2.er` 実験では、すべてのイベントを記録するようにしきい値が記録されていたため、最初のスレッドのロック獲得も記録されました。

同様に、最初の実験では、ロックの競合がなかったため `lock_local()` に対してはイベントが記録されませんでした。2 番目の実験では 4 つのイベントがありましたが、遅延は合算しても無視できる程度でした。

パフォーマンスアナライザのその他の調査

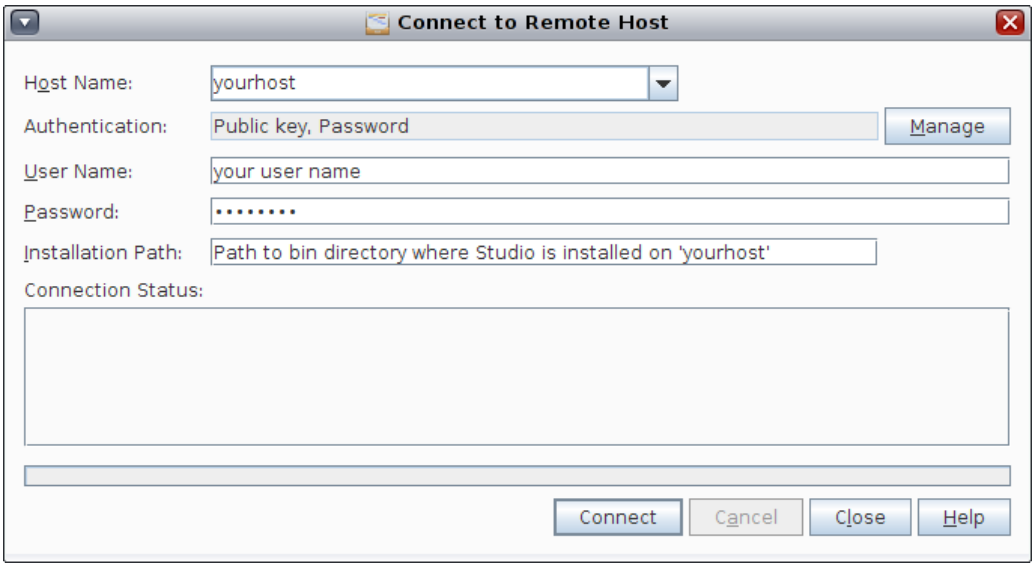
この章では、パフォーマンスアナライザで実行できるその他のチュートリアルとタスクを調査し、追加情報の参照先も示します。

リモートパフォーマンスアナライザの使用

サポートされるシステム、または Mac OS や Windows などの Oracle Developer Studio をインストールできないシステムのいずれかからリモートパフォーマンスアナライザを使用できます。この特殊なバージョンのパフォーマンスアナライザのインストールおよび使用については、『Oracle Developer Studio 12.5: パフォーマンスアナライザ』の「パフォーマンスアナライザのリモートでの使用」を参照してください。

パフォーマンスアナライザをリモートで起動すると、同じ「ようこそ」ページが表示されますが、実験を作成および表示するためのオプションは無効になっており、グレー表示されます。

「リモートホストに接続」をクリックすると、パフォーマンスアナライザで接続ダイアログが開きます。



接続先のシステムの名前、そのシステムのユーザー名とパスワード、およびそのシステムでの Oracle Developer Studio インストールへのインストールパスを入力します。「接続」をクリックすると、パフォーマンスアナライザが名前とパスワードを使用してリモートシステムにログインして、接続を検証します。

この時点から、下部にあるステータス領域には接続先のリモートホストの名前が表示される点を除き、「ようこそ」ページの外観は、ローカルパフォーマンスアナライザと同じようになります。ここから上の手順 2 に進みます。

追加のチュートリアル

「パフォーマンスアナライザのチュートリアルの概要」で説明したように、10 ページの「チュートリアルのサンプルコードの入手」でダウンロードしたサンプル zip ファイルの `PerformanceAnalyzer` サブディレクトリには、ほかのチュートリアルがいくつかあります。次のリストに、これらの各チュートリアルの詳細を示します。

`cachetest` — `cachetest` チュートリアルでは、コードのパフォーマンスに対するコンパイラの最適化の影響、および Oracle Developer Studio コンパイラのコンパイラ解説の表示方法について調査し、最適化の理解に役立てます。

`ksynprog` — `ksynprog` チュートリアルでは、カーネルからパスをトリガーするタスクの実行、および結果として生成されたプログラムのパフォーマンスデータの収集について調査します。

`omptest`	`omptest` チュートリアルでは、OpenMP 並列化ディレクティブの使用、およびその結果生じる、パフォーマンスアナライザで調べることができるパフォーマンス特性について調査します。
`synprog`	`synprog` チュートリアルでは、多数のタスクを実行してパフォーマンスアナライザの一部のパフォーマンス特性や機能を示す、単純なプログラムについて調査します。

追加情報

パフォーマンスアナライザおよび関連するデータ収集ツールの追加情報は、次のリソースから入手できます。

- パフォーマンスアナライザの統合ヘルプシステム
- 『Oracle Developer Studio 12.5: パフォーマンスアナライザ』
- Oracle Developer Studio の開発者ポータル (`http://www.oracle.com/technetwork/server-storage/solarisstudio/`) で提供されている記事およびホワイトペーパー。
- 『Oracle Developer Studio 12.5 リリースの新機能』の 第 5 章,「パフォーマンス解析ツール」
- 『Oracle Developer Studio 12.5: リリースノート』の「パフォーマンスアナライザおよびer_print ユーティリティの制限事項」