Oracle® Solaris Studio 12.2: パフォーマンスアナライザ

Part No: 821–2491
2010 年 9 月

ORACLE

110412@25097

目次

はじめに

このマニュアルでは、Oracle Solaris Studio 12.2 ソフトウェアのパフォーマンス解析ツールについて説明します。コレクタおよびパフォーマンスアナライザという 2 つのツールを併用することによって、広範囲のパフォーマンスデータの統計的プロファイリングと多数のコールのトレースを行い、そのデータを関数、ソース行、および命令レベルでプログラム構造に関連付けます。

対象読者

このマニュアルは、Fortran、C、C++、Java のいずれかのプログラミング言語に関する実用的な知識を持つアプリケーション開発者を対象にしています。パフォーマンスツールのユーザーは、Solaris オペレーティングシステムまたは Linux オペレーティングシステムと、UNIX® オペレーティングシステムのコマンドを理解している必要があります。パフォーマンス解析についての知識があると役立ちますが、ツールを使用する上では必須ではありません。

サポートされるプラットフォーム

この Oracle Solaris Studio リリースは、SPARC および x86 ファミリ (UltraSPARC、SPARC64、AMD64、Pentium、Xeon EM64T) プロセッサアーキテクチャーを使用するシステムをサポートしています。使用の Solaris オペレーティングシステムのバージョンに対するシステムのサポート状況は、ハードウェア互換性リスト (`http://www.sun.com/bigadmin/hcl`) を参照してください。ここには、すべてのプラットフォームごとの実装の違いについて説明されています。

このドキュメントでは、x86 関連の用語は次のものを指します。

- 「x86」は、64 ビットおよび 32 ビットの x86 互換製品を指します。
- 「x64」は、AMD 64 または EM64T システムで、特定の 64 ビット情報を指します。
- 「32 ビット x86」は、x86 ベースシステムで特定の 32 ビット情報を指します。

サポートされるシステムについては、ハードウェアの互換性に関するリストを参照してください。

関連ドキュメント

スレッドアプリケーションに対してパフォーマンスアナライザを使用する場合は、『Oracle Solaris Studio 12.2: スレッドアナライザユーザーズガイド』を参照してください。

パフォーマンスアナライザを使用してカーネルをプロファイルする場合は、Solaris 10 Software Developer Collection (`http://docs.sun.com/app/docs/coll/45.20`) にある『DTrace User Guide』を参照してください。

関連するサードパーティの Web サイトリファレンス

このマニュアルには、詳細な関連情報を提供するサードパーティの URL が記載されています。

注 – このマニュアルで紹介するサードパーティ Web サイトが使用可能かどうかについては、Oracle は責任を負いません。このようなサイトやリソース上、またはこれらを経由して利用できるコンテンツ、広告、製品、またはその他の資料についても、Oracle は保証しておらず、法的責任を負いません。また、このようなサイトやリソースから直接あるいは経由することで利用できるコンテンツ、商品、サービスの使用または依存が直接のあるいは関連する要因となり実際に発生した、あるいは発生するとされる損害や損失についても、Oracle は一切の法的責任を負いません。

Oracle Solaris Studio マニュアルへのアクセス方法

マニュアルには、次の場所からアクセスできます。

- マニュアルは、次に示すマニュアル索引のページからアクセスできます。`http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation/index.html`
- IDE のオンラインヘルプは、IDE 内の「ヘルプ」メニューだけでなく、F1 キー、および多くのウィンドウやダイアログにある「ヘルプ」ボタンを使用してアクセスできます。
- パフォーマンスアナライザおよびスレッドアナライザのオンラインヘルプは、これらのツール内の「ヘルプ」メニューだけでなく、F1 キー、および多くのウィンドウやダイアログボックスにある「ヘルプ」ボタンを使用してアクセスできます。
- dbxtool および DLight のオンラインヘルプは、これらのツール内のの「ヘルプ」メニューだけでなく、F1 キー、および多くのダイアログボックスにある「ヘルプ」ボタンを使用してアクセスできます。

アクセシブルな製品マニュアル

マニュアルは、技術的な補足をすることで、ご不自由なユーザーの方々にとって読みやすい形式のマニュアルを提供しております。アクセシブルなマニュアルは次の表に示す場所から参照することができます。

マニュアルの種類	アクセシブルな形式と格納場所
マニュアル	HTML 形式。docs.sun.com にある Oracle Solaris Studio 12.2 Collection - Japanese から選択
『Oracle Solaris Studio 12.2 リリースの新機能』(以前のリリースのコンポーネントの Readme ファイルに含まれていた情報)	HTML 形式。docs.sun.com にある Oracle Solaris Studio 12.2 Collection - Japanese から選択
マニュアルページ	インストールされた製品で `man` コマンドを実行
オンラインヘルプ	HTML 形式。IDE、`dbxtool`、DLight、パフォーマンスアナライザを使用時に「ヘルプ」メニュー、「ヘルプ」ボタン、および F1 キーから表示
リリースノート	HTML 形式の『Oracle Solaris Studio 12.2 リリースノート』

マニュアル、サポート、およびトレーニング

追加リソースについては、次の Web サイトを参照してください。

- マニュアル (`http://docs.sun.com`)
- サポート (`http://www.oracle.com/us/support/systems/index.html`)
- トレーニング (`http://education.oracle.com`) – 左のナビゲーションバーにある Sun リンクをクリックします。

ご意見の送付先

マニュアルの品質や使いやすさに関するご意見やご提案をお待ちしています。間違いやその他の改善すべき箇所がありましたら、`http://docs.sun.com` で「Feedback」をクリックしてお知らせください。ドキュメント名とドキュメントの Part No.、および、可能な場合は章、節、ページ番号を記載してください。返答が必要な場合はお知らせください。

Oracle 技術ネットワーク (http://www.oracle.com/technetwork/index.html) では、Oracle ソフトウェアに関するさまざまなリソースを提供しています。

- 技術上の問題やソリューションについては、ディスカッションフォーラム (http://forums.oracle.com) を参照してください。
- 実践的なステップ・バイ・ステップのチュートリアルについては、Oracle By Example (http://www.oracle.com/technology/obe/start/index.html) を参照してください。
- サンプルコードのダウンロードについては、サンプルコード (http://www.oracle.com/technology/sample_code/index.html) を参照してください。

表記上の規則

このマニュアルでは、次のような字体や記号を特別な意味を持つものとして使用します。

表 P–1 表記上の規則

字体または記号	意味	例
AaBbCc123	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コード例を示します。	.login ファイルを編集します。 ls -a を使用してすべてのファイルを表示します。 system%
AaBbCc123	ユーザーが入力する文字を、画面上のコンピュータ出力と区別して示します。	system% **su** password:
AaBbCc123	変数を示します。実際に使用する特定の名前または値で置き換えます。	ファイルを削除するには、rm *filename* と入力します。
『 』	参照する書名を示します。	『コードマネージャ・ユーザーズガイド』を参照してください。
「 」	参照する章、節、ボタンやメニュー名、強調する単語を示します。	第 5 章「衝突の回避」を参照してください。 この操作ができるのは、「スーパーユーザー」だけです。
\	枠で囲まれたコード例で、テキストがページ行幅を超える場合に、継続を示します。	sun% **grep '^#define ** **XV_VERSION_STRING'**

コード例は次のように表示されます。

- C シェル

  ```
  machine_name% command y|n [filename]
  ```

- C シェルのスーパーユーザー

  ```
  machine_name# command y|n [filename]
  ```

- Bourne シェルおよび Korn シェル

  ```
  $ command y|n [filename]
  ```

- Bourne シェルおよび Korn シェルのスーパーユーザー

  ```
  # command y|n [filename]
  ```

[ ] は省略可能な項目を示します。上記の例は、*filename* は省略してもよいことを示しています。

| は区切り文字 (セパレータ) です。この文字で分割されている引数のうち 1 つだけを指定します。

キーボードのキー名は英文で、頭文字を大文字で示します (例: Shift キーを押します)。ただし、キーボードによっては Enter キーが Return キーの動作をします。

ダッシュ (-) は 2 つのキーを同時に押すことを示します。たとえば、Ctrl-D は Control キーを押したまま D キーを押すことを意味します。

◆ ◆ ◆ 第 1 章

1 パフォーマンスアナライザの概要

高性能なアプリケーションを開発するには、コンパイラのさまざまな機能、最適化された関数のライブラリ、およびパフォーマンス解析のためのツールを組み合わせる必要があります。この『パフォーマンスアナライザ』マニュアルでは、コードのパフォーマンス評価、潜在的なパフォーマンス上の問題の特定、および問題が発生するコード部分の特定に役立つツールについて説明します。

この章では、次の内容について説明します。

パフォーマンス解析ツール

このマニュアルでは、コレクタとパフォーマンスアナライザについて説明します。これらは、アプリケーションのパフォーマンスデータを収集および解析するために使用する 1 組のツールです。また、このマニュアルでは、収集したパフォーマンスデータをテキスト形式で表示し解析するためのコマンド行ツールである `er_print` ユーティリティーについても説明します。アナライザと `er_print` ユーティリティーは、ほぼ同じデータを表示しますが、ユーザーインタフェースが異なります。

スポットという名前の追加の Oracle Solaris Studio ツールを使用すると、アプリケーションについてのパフォーマンスレポートを作成できます。これは、パフォーマンスアナライザの補完ツールになります。詳細は、`spot(1)` のマニュアルページを参照してください。

コレクタとパフォーマンスアナライザは、パフォーマンスチューニングが主な担当ではない開発者も含めて、すべてのソフトウェア開発者を対象として設計されています。これらのツールは、一般的に使用されているプロファイリングツールの `prof` および `gprof` に比べて柔軟性が高く、詳細で正確な解析が可能になります。また、`gprof` に見られる時間の因果関係の判定の誤りもありません。

これらのツールは、次のような疑問の解決に役立ちます。

- 使用可能なリソースのうちのどのぐらいがプログラムによって消費されるのか。
- どの関数またはロードオブジェクトが特に多くのリソースを消費するのか。
- どのソース行と命令がリソースを消費するのか。
- 特定の地点に達するまでにプログラムはどのような実行過程を経ているのか。
- 関数またはロードオブジェクトはどのようなリソースを消費しているのか。

コレクタツール

コレクタツールは、プロファイリングと呼ばれる統計手法を使用し、関数呼び出しをトレースすることによって、パフォーマンスデータを収集します。データの内容には、呼び出しスタック、マイクロステートアカウンティング情報 (Solaris プラットフォームのみ)、スレッド同期遅延データ、ハードウェアカウンタのオーバーフローデータ、Message Passing Interface (MPI) 関数呼び出しデータ、メモリー割り当てデータ、およびオペレーティングシステムとプロセスの概要情報が含まれています。コレクタは C、C++、および Fortran に関するプログラムのあらゆる種類のデータを収集できるとともに、Java プログラミング言語で書かれたアプリケーションに関するプロファイルデータを収集できます。また、動的に生成される関数と派生プロセスに関するデータも収集できます。収集対象のデータについては第 2 章「パフォーマンスデータ」を、コレクタの詳細については第 3 章「パフォーマンスデータの収集」を参照してください。コレクタは、パフォーマンスアナライザ GUI、`dbx` コマンド行ツール、および `collect` コマンドを使用して実行できます。

パフォーマンスアナライザツール

パフォーマンスアナライザツールは、ユーザーがパフォーマンスデータを評価できるように、コレクタによって記録されたデータを表示します。パフォーマンスアナライザはデータを処理し、プログラム、関数、ソース行、および命令のレベルでパフォーマンスに関するさまざまなメトリックスを表示します。これらのメトリックスは、次の 5 つのグループに分類されます。

- 時間プロファイルメトリックス
- ハードウェアカウンタメトリックス
- 同期遅延メトリックス
- メモリー割り当てメトリックス
- MPI トレースメトリックス

パフォーマンスアナライザタイムラインは、raw データを時間の関数としてグラフィカル形式で表示することができます。

パフォーマンスアナライザについての詳細は、第 4 章「パフォーマンスアナライザツール」およびパフォーマンスアナライザのオンラインヘルプを参照してください。

第 5 章「er_print コマンド行パフォーマンス解析ツール」では、er_print コマンド行インタフェースを使用し、コレクタが収集したデータを解析する方法について説明しています。

第 6 章「パフォーマンスアナライザとそのデータについて」では、パフォーマンスアナライザとそのデータに関連する、データ収集の動作、パフォーマンスメトリックスの解釈、呼び出しスタックとプログラムの実行、注釈付きコードリストなどのトピックについて説明しています。注釈付きソースコードリストおよび逆アセンブリコードリストに、コンパイラのコメントが含まれており、パフォーマンスデータは含まれていない場合、er_src ユーティリティーで表示できます。詳細は、第 7 章「注釈付きソースと逆アセンブリデータについて」を参照してください。

第 7 章「注釈付きソースと逆アセンブリデータについて」では、注釈付きソースと逆アセンブリについて説明し、パフォーマンスアナライザが表示する各種インデックス行とコンパイラのコメントを解説します。

第 8 章「実験の操作」では、実験のコピー、移動、削除、アーカイブ、およびエクスポート方法について説明します。

第 9 章「カーネルプロファイリング」では、Solaris オペレーティングシステム (Solaris OS) が負荷を実行中に、Oracle Solaris Studio のパフォーマンスツールを使用してカーネルのプロファイリングを行う方法について説明しています。

er_print ユーティリティー

er_print ユーティリティーは、パフォーマンスアナライザにより提示されるすべての情報のうち、タイムライン、MPI タイムライン、および MPI グラフを除く情報を、標準テキストで出力します。これらの表示は本質的にグラフィカルであり、テキスト表示にはできません。

「パフォーマンスアナライザ」ウィンドウ

注 – 次に、「パフォーマンスアナライザ」ウィンドウの簡単な概要を示します。ここで説明するタブの機能および特徴の詳細は、第 4 章「パフォーマンスアナライザツール」とオンラインヘルプを参照してください。

「パフォーマンスアナライザ」ウィンドウは、複数のタブで構成されており、メニューバーとツールバーが付いています。パフォーマンスアナライザの起動時に表示されるタブには、各関数の排他的メトリックスと包括的メトリックスをまとめた、プログラムの関数の一覧が表示されます。これらのデータは、ロードオブジェクト、スレッド、軽量プロセス (LightWeight Process、LWP)、CPU、およびタイムスライスでフィルタ処理を実行できます。

関数を選択すると、その関数の呼び出し元と呼び出し先が別のタブに表示されます。このタブでは、呼び出しツリーをたどり、たとえば、メトリック値の大きなものを探すことができます。

このほか、ソースコードと逆アセンブリコードの 2 つのタブがあります。ソースコードのタブには、行単位でパフォーマンスメトリックス付きのソース行と、コンパイラのコメントがインタリーブされ、逆アセンブリコードのタブには、各命令のメトリックス付きの逆アセンブリコードと、可能であればソースコードおよびコンパイラのコメントがインタリーブされます。

パフォーマンスデータは、時間の関数として別のタブに表示されます。

MPI トレースデータのうち、プロセス、メッセージ、および関数が 1 つのタブのタイムラインに、グラフが別のタブに表示されます。

OpenMP 並列領域は 1 つのタブに表示され、OpenMP タスクは別のタブに表示されます。

このほか、実験とロードオブジェクトの詳細、関数の概要情報、メモリーリーク、およびプロセスの統計を表示するタブもあります。

そのほかのタブには、インデックスオブジェクト、メモリオブジェクト、データオブジェクト、データレイアウト、および PC が表示されます。各タブについては、95 ページの「アナライザデータ表示」を参照してください。

また、スレッドアナライザのデータを記録した実験用に、データの競合とデッドロックのタブもあります。タブは、読み込んだ実験がそれらをサポートするデータを持つ場合にのみ表示されます。

スレッドアナライザについての詳細は、『Oracle Solaris Studio 12.2: スレッドアナライザユーザーズガイド』を参照してください。

パフォーマンスアナライザは、マウスの使用だけでなく、キーボードから操作することもできます。

◆ ◆ ◆ 第 2 章

2 パフォーマンスデータ

パフォーマンスツールは、プログラムの実行中に特定のイベントに関するデータを記録し、メトリックスと呼ばれるプログラムパフォーマンスの測定基準にデータを変換します。メトリックスは、関数、ソース行、および命令に対して表示されます。

この章では、パフォーマンスツールによって収集したデータをどのように処理して表示するか、またどのようにパフォーマンス解析に使用するかについて説明します。パフォーマンスデータを収集するツールは複数あり、どのツールも「コレクタ」という用語で呼ばれています。同様に、パフォーマンスデータを解析するツールも複数あり、どのツールも「解析ツール」という用語で呼ばれています。

この章では、次の内容について説明します。

- 23 ページの「コレクタが収集するデータの内容」
- 37 ページの「プログラム構造へのメトリックスの対応付け」

パフォーマンスデータの収集と格納については、第 3 章「パフォーマンスデータの収集」を参照してください。

コレクタが収集するデータの内容

コレクタは、プロファイルデータ、トレースデータ、および大域データの 3 種類のデータを収集します。

- プロファイルデータの収集は、一定の間隔でプロファイルイベントを記録することによって行います。間隔は、システム時間を使用して取得した時間間隔、または特定のタイプのハードウェアイベントの数です。指定の間隔に達すると、シグナルがシステムに送られ、次の機会にデータが記録されます。
- トレースデータの収集は、さまざまなシステム関数をラッパー関数で割り込み、それによってシステム関数の呼び出しをインターセプトし、呼び出しに関するデータを記録することによって行います。

- 大域データの収集は、さまざまなシステムルーチンを呼び出して情報を取得することによって行います。大域データパケットのことを標本と呼びます。

プロファイルデータとトレースデータの両方が特定のイベントに関する情報を含んでおり、いずれのデータの種類もパフォーマンスメトリックスに変換されます。大域データはメトリックスに変換されませんが、プログラムの実行を複数のタイムセグメントに分割するために使用できるマーカーを提供します。大域データは、タイムセグメントにおけるプログラム実行の概要を示します。

それぞれのプロファイルイベントやトレースイベントで収集されたデータパケットには、次の情報が含まれます。

- データ識別用のヘッダー
- 高分解能のタイムスタンプ
- スレッド ID
- 軽量プロセス (LWP) ID
- プロセッサ (CPU) ID (オペレーティングシステムから提供できる場合)
- 呼び出しスタックのコピー。Java プログラムの場合、マシン呼び出しスタックと Java 呼び出しスタックの 2 つの呼び出しスタックが記録されます。
- OpenMP プログラムの場合、現在の並行領域の識別子と、OpenMP の状態も収集されます。

スレッドと軽量プロセスについての詳細は、第 6 章「パフォーマンスアナライザとそのデータについて」を参照してください。

こうした共通の情報のほかに、各イベントに固有のデータパケットには、データの種類に固有の情報が含まれます。コレクタが記録できるデータは、次の 5 種類です。

- 時間プロファイルデータ
- ハードウェアカウンタのオーバーフロープロファイルデータ
- 同期待ちトレースデータ
- ヒープトレース (メモリー割り当て) データ
- MPI トレースデータ

これらの 5 種類のデータ、それらから派生したメトリックス、およびメトリックスの使用方法については、あとの項で説明します。6 種類目のデータ、大域標本データには呼び出しスタック情報が含まれないため、メトリックスに変換できません。

時間データ

時間ベースのプロファイル時に収集されるデータは、オペレーティングシステムが提供するメトリックスによって異なります。

Solaris OS での時間ベースのプロファイル

Solaris OS での時間ベースのプロファイルでは、各 LWP の状態が定期的な間隔で格納されます。この時間間隔は、プロファイル間隔と呼ばれます。この情報は整数の配列に格納されます。その配列の 1 つの要素は、カーネルによって維持される 10 個のマイクロアカウンティングの各状態に使用されます。収集されたデータは、各状態で消費された、プロファイル間隔の分解能を持つ時間値に、パフォーマンスアナライザによって変換されます。デフォルトのプロファイル間隔は、約 10 ミリ秒 (10 ms) です。コレクタは、約 1 ミリ秒の高分解能プロファイル間隔と、約 100 ミリ秒の低分解能プロファイル間隔を提供し、OS で許されれば任意の間隔を許可します。引数を付けずに `collect` コマンドを実行すると、このコマンドが実行されるシステム上で許される範囲と分解能が出力されます。

次の表に、時間ベースのデータから計算されるメトリックスの定義を示します。

表 2–1 Solaris タイミングメトリックス

メトリック	定義
ユーザー CPU 時間	CPU のユーザーモードで実行中に使用される LWP 時間。
時計時間	LWP 1 で使用される LWP 時間。通常は、「実経過時間」です。
LWP 合計時間	LWP 時間の総合計。
システム CPU 時間	CPU のカーネルモードまたはトラップ状態で実行中に使用される LWP 時間。
CPU 待ち時間	CPU の待機中に使用される LWP 時間。
ユーザーロック時間	ロックの待機中に使用される LWP 時間。
テキストページフォルト時間	テキストページの待機中に使用される LWP 時間。
データページフォルト時間	データページの待機中に使用される LWP 時間。
ほかの待ち時間	カーネルページ待機中に使用される LWP 時間、またはスリープ中か停止中に使用される時間。

マルチスレッドの実験では、すべての LWP にまたがって実経過時間以外の時間が集計されます。定義した時計時間は、MPMD (Multiple-Program Multiple-Data) プログラムには意味がありません。

タイミングメトリックスは、プログラムがいくつかのカテゴリで時間を費やした部分を示し、プログラムのパフォーマンス向上に役立てることができます。

- ユーザー CPU 時間が大きいということは、その場所で、プログラムが仕事の大半を行なっていることを示します。この情報は、アルゴリズムを再設計することによって特に有益となる可能性があるプログラム部分を見つけるのに使用できます。

- システム CPU 時間が大きいということは、プログラムがシステムルーチンに対する呼び出しで多くの時間を使用していることを示します。
- CPU 待ち時間が大きいということは、使用可能な CPU 以上に実行可能なスレッドが多いか、ほかのプロセスが CPU を使用していることを示します。
- ユーザーロック時間が大きいということは、要求対象のロックをスレッドが取得できないことを示します。
- テキストページフォルト時間が大きいということは、リンカーによって指定されたコードが、多数の呼び出しまたは分岐で新しいページの読み込みが発生するようなメモリー上の配置になることを意味します。
- データページフォルト時間が大きいということは、データへのアクセスによって新しいページの読み込みが発生していること示します。この問題は、プログラムのデータ構造またはアルゴリズムを変更することによって解決できます。

Linux OS での時間ベースのプロファイル

Linux OS で利用できるメトリックは、ユーザー CPU 時間だけです。報告される合計 CPU 使用時間は正確ですが、アナライザは Solaris OS の場合ほど正確に実際のシステム CPU 時間の割合を判別できない場合があります。アナライザは軽量プロセス (LightWeight Process、LWP) のデータであるかのように情報を表示しますが、実は Linux OS 上に LWP はなく、表示される LWP ID は実際にはスレッド ID です。

MPI プログラム対応の時間ベースのプロファイル

時間プロファイリングデータは、Oracle Message Passing Toolkit (以前の Sun HPC ClusterTools) で実行される MPI 実験で収集できます。Oracle Message Passing Toolkit はバージョン 8.1 またはそれ以降である必要があります。

Linux で Oracle Message Passing Toolkit 8.2 または 8.2.1 を使用する場合、回避策が必要になる場合があります。バージョン 8.1 または 8.2.1c の場合、または Oracle Solaris Studio コンパイラを使用している場合はどのバージョンでも回避策は必要ありません。回避策については、docs.sun.com の Oracle Solaris Studio 12.2 Collection - Japanese 内の『Oracle Solaris Studio 12.2 リリースの新機能』を参照してください。

MPI 実験で時間プロファイリングデータを収集すると、次の 2 つのメトリックスが追加されます。

- MPI 作業: 要求やメッセージの処理など、プロセスが MPI ランタイム実行作業の内部にある場合に蓄積されます。
- MPI 待機: プロセスが MPI ランタイムの内部にあり、イベント、バッファー、またはメッセージを待機している場合に蓄積されます。

Solaris OS では、MPI 作業は作業が直列または並列に実行される場合に蓄積されます。MPI 待機は、MPI ランタイムが同期化を待機している間に蓄積され、待機が CPU 時間ないしスリーピングのいずれかを使用しているか、または作業が並列実行中であるがスレッドは CPU 上にスケジュールされていない場合に、蓄積されます。

Linux OS では、MPI 作業および MPI 待機は、プロセスがユーザーモードまたはシステムモードでアクティブになっている場合にのみ蓄積されます。MPI がビジーウェイトを行う必要があるものとして指定しないかぎり、Linux での MPI 待機は有用ではありません。

OpenMP プログラム対応の時間ベースのプロファイル

時間ベースのプロファイルが OpenMP プログラムで実行される場合は、OpenMP 作業および OpenMP 待機という 2 つの追加メトリックスが提供されます。

Solaris OS では、OpenMP 作業は作業が直列または並列に実行される場合に蓄積されます。OpenMP 待機は、OpenMP ランタイムが同期化を待機している場合に蓄積し、待機が CPU 時間かスリーピングを使用しているか、または作業は並行してなされるがスレッドが CPU 上でスケジュールされていない場合に蓄積します。

Linux OS では、OpenMP 作業および OpenMP 待機は、プロセスがユーザーモードまたはシステムモードでアクティブになっている場合にのみ蓄積されます。OpenMP でビジーウェイトを行う必要があるものとして指定しないかぎり、Linux での OpenMP は有用ではありません。

ハードウェアカウンタオーバーフローのプロファイルデータ

ハードウェアカウンタは、キャッシュミス、キャッシュストールサイクル、浮動小数点演算、分岐予測ミス、CPU サイクル、および実行対象命令といったイベントの追跡に使用されます。ハードウェアカウンタオーバーフローのプロファイルでは、LWP が動作している CPU の特定のハードウェアカウンタがオーバーフローしたときに、コレクタはプロファイルパケットを記録します。この場合、そのカウンタはリセットされ、カウントを続行します。プロファイルパケットには、オーバーフロー値とカウンタタイプが入っています。

各種の CPU ファミリが 2 潤ォ 18 個の同時ハードウェアカウンタレジスタをサポートしています。コレクタは、複数のレジスタ上でデータを収集できます。コレクタではレジスタごとに、オーバーフローを監視するカウンタの種類を選択し、カウンタのオーバーフロー値を設定することができます。ハードウェアカウンタには、任意のレジスタを使用できるものと、特定のレジスタしか使用できないものがあります。このことは、1 つの実験であらゆるハードウェアカウンタの組み合わせを選択できるわけではないことを意味します。

パフォーマンスアナライザは、ハードウェアカウンタのオーバーフロープロファイルデータをカウントメトリックスに変換します。循環型のカウンタの場合、報告されるメトリックスは時間に変換されます。非循環型のカウンタの場合は、イベントの発生回数になります。複数の CPU を搭載したマシンの場合、メトリックスの変換に使用されるクロック周波数が個々の CPU のクロック周波数の調和平均となりま

す。プロセッサのタイプごとに専用のハードウェアカウンタセットがあり、また
ハードウェアカウンタの数が多いため、ハードウェアカウンタメトリックスはここ
に記載していません。次の項で、どのような種類のハードウェアカウンタがあるか
について調べる方法を説明します。

ハードウェアカウンタの用途の 1 つは、CPU に出入りする情報フローに伴う問題を
診断することです。たとえば、キャッシュミス回数が多いということは、プログラ
ムを再構成してデータまたはテキストの局所性を改善するか、キャッシュの再利用
を増やすことによってプログラムのパフォーマンスを改善できることを意味しま
す。

ハードウェアカウンタはほかのカウンタと関連する場合があります。たとえば、分
岐予測ミスが発生すると、間違った命令が命令キャッシュに読み込まれ、これらの
命令を正しい命令と置換しなければならなくなるため、分岐予測ミスと命令
キャッシュミスが関連付けられることがよくあります。置換により、命令
キャッシュミス、命令変換索引バッファー (Instruction Translation Look aside
Buffer、ITLB) ミス、またはページフォルトが発生する可能性があります。

ハードウェアカウンタのオーバーフローは、イベントを発生させて対応するイベン
トのカウンタをオーバーフローにした命令のあとに、1 つ以上の命令で実現される傾
向があります。これは「滑り止め」と呼ばれ、カウンタオーバーフローのプロ
ファイルを解釈しにくくする可能性があります。原因となる命令を正確に識別する
ためのハードウェアサポートがないと、候補の原因となる命令の適切なバックト
ラッキング検索が行われる場合があります。

そのようなバックトラッキングが収集中にサポートされて指定されると、ハード
ウェアカウンタプロファイルパケットにはさらに、ハードウェアカウンタイベント
に適した候補の、メモリー参照命令の PC (プログラムカウンタ) と EA (有効アドレス)
が組み込まれます。解析中の以降の処理は、候補のイベント PC と EA を有効にする
のに必要です。このメモリー参照イベントに関する追加情報により、データ空間プ
ロファイリングと呼ばれるさまざまなデータ指向解析が容易になります。バックト
ラッキングは、Oracle Solaris オペレーティングシステムを実行している SPARC
ベースのプラットフォームでのみサポートされます。

候補のイベント PC および EA のバックトラッキングと記録は、時間プロファイルに
対しても指定できますが、解釈しにくい場合があります。

ハードウェアカウンタのリスト

ハードウェアカウンタはプロセッサ固有であるため、どのカウンタを利用できるか
は、使用しているプロセッサによって異なります。パフォーマンスツールには、よ
く使われると考えられるいくつかのカウンタの別名が用意されています。コレクタ
から特定システム上で利用できるハードウェアカウンタの一覧を取り出すには、引
数を付けないで `collect` をそのシステム上の端末ウィンドウに入力します。プロ
セッサとシステムがハードウェアカウンタプロファイルをサポートしている場
合、`collect` コマンドは、ハードウェアカウンタに関する情報が入った 2 つのリスト

を出力します。最初のリストには一般的な名称に別名が設定されたハードウェアカウンタが含まれ、2 番目のリストには raw ハードウェアカウンタが含まれます。パフォーマンスカウンタサブシステムも `collect` コマンドも特定システムのカウンタの名前を知らない場合、各リストは空になります。ただしほとんどの場合、カウンタは数値で指定できます。

次に、カウンタリストに含まれるエントリの表示例を示します。別名が設定されたカウンタがリストの最初に表示され、続いて raw ハードウェアカウンタリストが表示されます。この例の出力における各行は、印刷用の形式になっています。

```
Aliased HW counters available for profiling:
cycles[/{0|1}],9999991 ('CPU Cycles', alias for Cycle_cnt; CPU-cycles)
insts[/{0|1}],9999991 ('Instructions Executed', alias for Instr_cnt; events)
dcrm[/1],100003 ('D$ Read Misses', alias for DC_rd_miss; load events)
...
Raw HW counters available for profiling:
Cycle_cnt[/{0|1}],1000003 (CPU-cycles)
Instr_cnt[/{0|1}],1000003 (events)
DC_rd[/0],1000003 (load events)
```

別名が設定されたハードウェアカウンタリストの形式

別名が設定されたハードウェアカウンタリストでは、最初のフィールド (たとえば、`cycles`) は、`collect` コマンドの `-h` *counter...* 引数で使用できる別名を示します。この別名は、 `er_print` コマンド内で使用する識別子でもあります。

リストの 2 番目のフィールドには、そのカウンタに使用可能なレジスタ、たとえば、`[/{0|1}]` が示されます。

3 番目のフィールドは、たとえば `9999991` など、カウンタのデフォルトのオーバーフロー値です。別名が設定されたカウンタの場合は、合理的なサンプルレートを提供するためにデフォルト値が選択されています。実際のレートは、かなり変化するため、デフォルト以外の値を指定する必要がある場合もあります。

4 番目のフィールドは、括弧で囲まれ、タイプ情報を含んでいます。これは、簡単な説明 (`CPU Cycles` など)、raw ハードウェアカウンタ名 (`Cycle_cnt` など)、およびカウントされる単位の種類 (`CPU-cycles` など) を提供します。

タイプ情報の最初のワードが、

- load、store、または load-store のいずれかである場合、そのカウンタはメモリーに関連したものです。collect -h コマンド内でカウンタ名の前に + 符号を付ける (たとえば、+dcrm) ことにより、イベントの原因となった正確な命令と仮想アドレスを検索できます。+ 符号を使用すると、データ空間プロファイリングも使用可能になります。詳細については、106 ページの「「データオブジェクト」タブ」、107 ページの「「データレイアウト」タブ」、および109 ページの「メモリーオブジェクトのタブ」を参照してください。
- not-program-related である場合、カウンタはほかのプログラムによって開始されたイベント、たとえば CPU 対 CPU のキャッシュスヌープなどを取り込みます。プロファイリングにカウンタを使用すると、警告が生成され、プロファイリングで呼び出しスタックが記録されません。

タイプ情報の 2 番目または唯一のワードが、

- CPU-cycles である場合は、そのカウンタを使用して時間ベースのメトリックを提供できます。そのようなカウンタについて報告されるメトリックスは、デフォルトでは包括的時間および排他的時間へ変換されますが、イベントカウントとして表示することもできます。
- events である場合、メトリックは包括的および排他的イベントカウントであり、時間へ変換できません。

この例の別名が設定されたハードウェアカウンタリストでは、タイプ情報に 1 ワードが含まれており、最初のカウンタの場合は CPU-cycles で、2 番目のカウンタの場合は、events となっています。3 番目のカウンタでは、タイプ情報に load events という 2 ワードが含まれています。

raw ハードウェアカウンタリストの形式

raw ハードウェアカウンタリストに含まれる情報は、別名設定されたハードウェアカウンタリストに含まれる情報のサブセットです。raw ハードウェアカウンタリスト内の各行には、cpu-track(1) によって使用された内部カウンタ名、そのカウンタを使用できるレジスタ番号 (単数または複数)、デフォルトのオーバーフロー値、およびカウンタ単位が含まれており、カウンタ単位は CPU-cycles か Events です。

カウンタがプログラムの実行に関連のないイベントを測定する場合、タイプ情報の最初のワードは not-program-related になります。そのようなカウンタの場合、プロファイリングで呼び出しスタックが記録されませんが、その代わりに、擬似関数 collector_not_program_related で使用された時間が示されます。スレッドと LWP ID は記録されますが、意味がありません。

raw カウンタのデフォルトのオーバーフロー値は 1000003 です。この値はほとんどの raw カウンタで最適でないため、raw カウンタを指定する際にオーバーフロー値を指定する必要があります。

同期待ちトレースデータ

マルチスレッドプログラムでは、たとえば1つのスレッドによってデータがロックされていると、別のスレッドがそのアクセス待ちになることがあります。このため、複数のスレッドが実行するタスクの同期を取るために、プログラムの実行に遅延が生じることがあります。これらのイベントは同期遅延イベントと呼ばれ、Solarisまたはpthreadのスレッド関数の呼び出しをトレースすることによって収集されます。同期遅延イベントを収集して記録するプロセスを同期待ちトレースと言います。また、ロック待ちに費やされる時間を待ち時間と言います。

ただし、イベントが記録されるのは、その待ち時間がしきい値(ミリ秒単位)を超えた場合だけです。しきい値0は、待ち時間に関係なく、あらゆる同期遅延イベントをトレースすることを意味します。デフォルトでは、同期遅延なしにスレッドライブラリを呼び出す測定試験を実施して、しきい値を決定します。こうして決定された場合、しきい値は、それらの呼び出しの平均時間に任意の係数(現在は6)を乗算して得られた値です。この方法によって、待ち時間の原因が本当の遅延ではなく、呼び出しそのものにあるイベントが記録されないようになります。この結果として、同期イベント数がかなり過小評価される可能性がありますが、データ量は大幅に少なくなります。

同期トレースはJavaプログラムに対してはサポートされていません。

同期待ちトレースデータは、次のメトリックスに変換されます。

表2–2 同期待ちトレースメトリックス

メトリック	定義
同期遅延イベント	待ち時間が所定のしきい値を超えたときの同期ルーチン呼び出し回数。
同期待ち時間	所定のしきい値を超えた総待ち時間。

この情報から、関数またはロードオブジェクトが頻繁にブロックされるかどうか、または同期ルーチンを呼び出したときの待ち時間が異常に長くなっているかどうかを調べることができます。同期待ち時間が大きいということは、スレッド間の競合が発生していることを示します。競合は、アルゴリズムの変更、具体的には、ロックする必要があるデータだけがスレッドごとにロックされるように、ロックを構成し直すことで減らすことができます。

ヒープトレース(メモリー割り当て)データ

正しく管理されていないメモリー割り当て関数やメモリー割り当て解除関数を呼び出すと、データの使い方の効率が低下し、プログラムパフォーマンスが低下する可能性があります。ヒープトレースでは、C標準ライブラリメモリー割り当て関数

malloc、realloc、valloc、memalign、および割り当て解除関数 free で割り込み処理を行うことによって、コレクタはメモリーの割り当てと割り当て解除の要求をトレースします。mmap への呼び出しはメモリー割り当てとして扱われ、これによって Java メモリー割り当てのヒープトレースイベントを記録することが可能になります。Fortran 関数 allocate および deallocate は C 標準ライブラリ関数を呼び出すため、これらのルーチンは間接的にトレースされます。

Java プログラムのヒーププロファイリングはサポートされません。

ヒープトレースデータは、次のメトリックスに変換されます。

表 2–3 メモリー割り当て (ヒープトレース) メトリックス

メトリック	定義
割り当て	メモリー割り当て関数の呼び出し回数。
割り当てバイト数	メモリー割り当て関数の呼び出しごとに割り当てられるバイト数の合計。
リーク	対応するメモリー割り当て解除関数が存在しなかったメモリー割り当て関数の呼び出し回数。
リークバイト数	割り当てられたが割り当て解除されなかったバイト数。

ヒープトレースデータの収集は、プログラム内のメモリーリークを特定したり、メモリーの割り当てが不十分な場所を見つける上で役立ちます。

dbx デバッグツールなどで使用されることの多い、メモリーリークの別の定義では、メモリーリークとは、プログラムのデータ空間内のいずれかを指しているポインタを持たない、動的に割り当てられるメモリーブロックです。ここで使用されているリークの定義にはこの代替定義が含まれますが、ポインタが存在するメモリーも含まれます。

MPI トレースデータ

コレクタは、Message Passing Interface (MPI) ライブラリの呼び出しの際のデータを収集できます。

MPI トレースは、オープンソースの VampirTrace 5.5.3 リリースを使用して実装されます。これは次の VampirTrace 環境変数を認識します。

VT_STACKS　呼び出しスタックを記録するかどうかを制御します。デフォルトの設定は 1 です。VT_STACKS を 0 に設定すると、呼び出しスタックが無効になります。

VT_BUFFER_SIZE	MPI API トレースコレクタの内部バッファーのサイズを制御します。デフォルト値は 64M (64M バイト) です。
VT_MAX_FLUSHES	MPI トレースの終了前に行うバッファーのフラッシュ回数を制御します。デフォルト値は 0 です。この場合、バッファーがいっぱいになるとディスクにフラッシュされます。VT_MAX_FLUSHES を正数に設定すると、バッファーがフラッシュされる回数が制限されます。
VT_VERBOSE	さまざまなエラーメッセージや状態メッセージをオンにします。デフォルト値は 1 で、重大なエラーメッセージや状態メッセージをオンにします。問題が生じる場合は、この変数を 2 に設定してください。

これらの変数については、Technische Universität Dresden Web サイトにある『Vampirtrace User Manual』を参照してください。

バッファーの制限に達したあとに発生する MPI イベントはトレースファイルに書き込まれないため、トレースが不完全になります。

この制限を撤廃してアプリケーションのトレースを完全なものにするには、VT_MAX_FLUSHES 環境変数を 0 に設定します。この設定を行うと、MPI API トレースコレクタは、バッファーがいっぱいになるたびにバッファーをディスクにフラッシュします。

バッファーのサイズを変更するには、VT_BUFFER_SIZE 環境変数を設定します。この変数の最適値は、トレース対象のアプリケーションによって異なります。小さな値を設定すると、アプリケーションに利用できるメモリーは増えますが、MPI API トレースコレクタによるバッファーのフラッシュが頻繁に行われるようになります。このようなバッファーのフラッシュによって、アプリケーションの動作が大幅に変化する可能性があります。その一方、大きな値 (2G など) を設定すると、MPI API トレースコントローラによるバッファーのフラッシュは最小限に抑えられますが、アプリケーションに使用できるメモリーは少なくなります。バッファーやアプリケーションデータを保持するために十分なメモリーを利用できない場合、アプリケーションの一部がディスクにスワップされて、アプリケーションの動作が大幅に変化する可能性があります。

次のリストに、データが収集される関数を示します。

MPI_Abort	MPI_Accumulate	MPI_Address
MPI_Allgather	MPI_Allgatherv	MPI_Allreduce

MPI_Alltoall	MPI_Alltoallv	MPI_Alltoallw
MPI_Attr_delete	MPI_Attr_get	MPI_Attr_put
MPI_Barrier	MPI_Bcast	MPI_Bsend
MPI_Bsend-init	MPI_Buffer_attach	MPI_Buffer_detach
MPI_Cancel	MPI_Cart_coords	MPI_Cart_create
MPI_Cart_get	MPI_Cart_map	MPI_Cart_rank
MPI_Cart_shift	MPI_Cart_sub	MPI_Cartdim_get
MPI_Comm_compare	MPI_Comm_create	MPI_Comm_dup
MPI_Comm_free	MPI_Comm_group	MPI_Comm_rank
MPI_Comm_remote_group	MPI_Comm_remote_size	MPI_Comm_size
MPI_Comm_split	MPI_Comm_test_inter	MPI_Dims_create
MPI_Errhandler_create	MPI_Errhandler_free	MPI_Errhandler_get
MPI_Errhandler_set	MPI_Error_class	MPI_Error_string
MPI_File_close	MPI_File_delete	MPI_File_get_amode
MPI_File_get_atomicity	MPI_File_get_byte_offset	MPI_File_get_group
MPI_File_get_info	MPI_File_get_position	MPI_File_get_position_shared
MPI_File_get_size	MPI_File_get_type_extent	MPI_File_get_view
MPI_File_iread	MPI_File_iread_at	MPI_File_iread_shared
MPI_File_iwrite	MPI_File_iwrite_at	MPI_File_iwrite_shared
MPI_File_open	MPI_File_preallocate	MPI_File_read
MPI_File_read_all	MPI_File_read_all_begin	MPI_File_read_all_end
MPI_File_read_at	MPI_File_read_at_all	MPI_File_read_at_all_begin
MPI_File_read_at_all_end	MPI_File_read_ordered	MPI_File_read_ordered_begin
MPI_File_read_ordered_end	MPI_File_read_shared	MPI_File_seek
MPI_File_seek_shared	MPI_File_set_atomicity	MPI_File_set_info
MPI_File_set_size	MPI_File_set_view	MPI_File_sync
MPI_File_write	MPI_File_write_all	MPI_File_write_all_begin
MPI_File_write_all_end	MPI_File_write_at	MPI_File_write_at_all
MPI_File_write_at_all_begin	MPI_File_write_at_all_end	MPI_File_write_ordered

MPI_File_write_ordered_begin	MPI_File_write_ordered_end	MPI_File_write_shared
MPI_Finalize	MPI_Gather	MPI_Gatherv
MPI_Get	MPI_Get_count	MPI_Get_elements
MPI_Get_processor_name	MPI_Get_version	MPI_Graph_create
MPI_Graph_get	MPI_Graph_map	MPI_Graph_neighbors
MPI_Graph_neighbors_count	MPI_Graphdims_get	MPI_Group_compare
MPI_Group_difference	MPI_Group_excl	MPI_Group_free
MPI_Group_incl	MPI_Group_intersection	MPI_Group_rank
MPI_Group_size	MPI_Group_translate_ranks	MPI_Group_union
MPI_Ibsend	MPI_Init	MPI_Init_thread
MPI_Intercomm_create	MPI_Intercomm_merge	MPI_Irecv
MPI_Irsend	MPI_Isend	MPI_Issend
MPI_Keyval_create	MPI_Keyval_free	MPI_Op_create
MPI_Op_free	MPI_Pack	MPI_Pack_size
MPI_Probe	MPI_Put	MPI_Recv
MPI_Recv_init	MPI_Reduce	MPI_Reduce_scatter
MPI_Request_free	MPI_Rsend	MPI_rsend_init
MPI_Scan	MPI_Scatter	MPI_Scatterv
MPI_Send	MPI_Send_init	MPI_Sendrecv
MPI_Sendrecv_replace	MPI_Ssend	MPI_Ssend_init
MPI_Start	MPI_Startall	MPI_Test
MPI_Test_cancelled	MPI_Testall	MPI_Testany
MPI_Testsome	MPI_Topo_test	MPI_Type_commit
MPI_Type_contiguous	MPI_Type_extent	MPI_Type_free
MPI_Type_hindexed	MPI_Type_hvector	MPI_Type_indexed
MPI_Type_lb	MPI_Type_size	MPI_Type_struct
MPI_Type_ub	MPI_Type_vector	MPI_Unpack
MPI_Wait	MPI_Waitall	MPI_Waitany
MPI_Waitsome	MPI_Win_complete	MPI_Win_create

```
MPI_Win_fence                MPI_Win_free                MPI_Win_lock
MPI_Win_post                 MPI_Win_start               MPI_Win_test
MPI_Win_unlock
```

MPI トレースデータは、次のメトリックスに変換されます。

表 2–4 MPI トレースメトリックス

メトリック	定義
MPI 送信数	開始された MPI ポイントツーポイント送信数
MPI 送信バイト数	MPI で送信されるバイト数
MPI 受信数	完了した MPI ポイントツーポイント受信数
MPI 受信バイト数	MPI で受信されるバイト数
MPI 時間	MPI 関数へのすべての呼び出しにかかった時間
そのほかの MPI イベント	2 点間のメッセージの送受信を行わない MPI 関数の呼び出しの数

MPI 時間は MPI 関数でかかった LWP 時間の合計です。MPI 状態の時間も収集される場合、MPI_Init および MPI_Finalize 以外のすべての MPI 関数については、MPI 作業時間と MPI 待機時間の合計が MPI 作業時間にほぼ等しくなるはずです。Linux では、MPI 待機および MPI 作業はユーザー CPU 時間とシステム CPU 時間の合計に基づきますが、MPI 時間は実際の時間に基づくため、数値は一致しません。

MPI のバイトおよびメッセージのカウントは、現在のところ 2 点間のメッセージについてのみ収集され、集合的な通信機能に関しては記録されません。MPI 受信バイト数は、すべてのメッセージで実際に受信したバイト数をカウントします。MPI 送信バイト数は、すべてのメッセージで実際に送信したバイト数をカウントします。MPI 送信数は送信したメッセージの数をカウントし、MPI 受信数は受信したメッセージの数をカウントします。

MPI トレースデータの収集は、MPI 呼び出しが原因となる可能性のある、MPI プログラム内のパフォーマンスの問題を抱えている場所を特定する上で役立ちます。パフォーマンスの問題となる可能性のある例としては、負荷分散、同期遅延、および通信のボトルネックがあります。

大域 (標本収集) データ

大域データは、コレクタによって標本パケットと呼ばれるパケット単位で記録されます。各パケットには、ヘッダー、タイムスタンプ、ページフォルトや I/O データなどのカーネルからの実行統計情報、コンテキストスイッチ、および各種のページの

常駐性 (ワーキングセットおよびページング) 統計情報が含まれます。標本パケットに記録されるデータはプログラムに対して大域的であり、パフォーマンスメトリックスには変換されません。標本パケットを記録するプロセスを標本収集と言います。

標本パケットは、次の状況で記録されます。

- IDE のデバッグ中や `dbx` で、プログラムが何らかの理由 (ブレークポイントなど) により停止したとき (このためのオプションが設定されている場合)。
- 標本収集の間隔の終了時 (定期的な標本収集を選択している場合)。標本収集の間隔は整数値 (秒単位) で指定します。デフォルト値は 1 秒です。
- `dbx collector sample record` コマンドを使用し、標本を手動で記録したとき。
- このルーチンに対する呼び出しがコードに含まれている場合に `collector_sample` を呼び出したとき (49 ページの「データ収集のプログラム制御」を参照)。
- `collect` コマンドで `-l` オプションが使用されている場合に指定したシグナルが送信されたとき (`collect`(1) のマニュアルページを参照)。
- 収集が開始および終了したとき。
- `dbx collector pause` コマンドで収集を一時停止したとき (一時停止の直前)、および `dbx collector resume` コマンドで収集を再開したとき (再開の直後)。
- 派生プロセスが作成される前後。

パフォーマンスツールは、標本パケットに記録されたデータを使用して、時間期間別に分類します。この分類されたデータを標本と呼びます。特定の標本セットを選択することによってイベントに固有のデータをフィルタできるので、特定の期間に関する情報だけを表示させることができます。各標本の大域データを表示することもできます。

パフォーマンスツールは、標本ポイントのさまざまな種類を区別しません。標本ポイントを解析に利用するには、1 種類のポイントだけを記録対象として選択してください。特に、プログラム構造や実行シーケンスに関する標本ポイントを記録する場合は、定期的な標本収集を無効にし、`dbx` がプロセスを停止したとき、`collect` コマンドによってデータ記録中のプロセスにシグナルが送られたとき、あるいはコレクタ API 関数が呼び出されたときのいずれかの状況で記録された標本を使用します。

プログラム構造へのメトリックスの対応付け

メトリックスは、イベント固有のデータとともに記録される呼び出しスタックを使用して、プログラムの命令に対応付けられます。情報を利用できる場合には、あらゆる命令がそれぞれ 1 つのソースコード行にマップされ、その命令に割り当てられたメトリックスも同じソースコード行に対応付けられます。この仕組みについての詳細は、第 6 章「パフォーマンスアナライザとそのデータについて」を参照してください。

メトリックスは、ソースコードと命令のほかに、より上位のオブジェクト (関数とロードオブジェクト) にも対応付けられます。呼び出しスタックには、プロファイルが取られたときに記録された命令アドレスに達するまでに行われた、一連の関数呼び出しに関する情報が含まれます。パフォーマンスアナライザは、この呼び出しスタックを使用し、プログラム内の各関数のメトリックスを計算します。こうして得られたメトリックスを関数レベルのメトリックスといいます。

関数レベルのメトリックス: 排他的、包括的、属性

パフォーマンスアナライザが計算する関数レベルのメトリックスには、排他的メトリックス、包括的メトリックス、および属性メトリックスの 3 種類があります。

- 関数の排他的メトリックスは、関数自体の内部で発生するイベントにより計算されます。これには、ほかの関数の呼び出しから発生したメトリックスは含まれません。
- 包括的メトリックスは、関数の内部で発生するイベントと、その関数から呼び出す関数により計算されます。これには、ほかの関数の呼び出しにから発生したメトリックが含まれます。
- 属性メトリックスは、どれだけの包括的メトリックが、別の関数からの呼び出しによるものか、または別の関数の呼び出しによるものかを示します。つまり、属性メトリックスは別の関数に起因するメトリックです。

呼び出しスタックの一番下にのみ現れる関数 (リーフ関数) では、その関数の排他的および包括的メトリックスは同じになります。

排他的および包括的メトリックスは、ロードオブジェクトについても計算されます。ロードオブジェクトの排他的メトリックスは、そのロードオブジェクト内の全関数の関数レベルのメトリックスを集計することによって計算されるメトリックスです。ロードオブジェクトの包括的メトリックスは、関数に対するのと同じ方法で計算されるメトリックスです。

関数の排他的および包括的メトリックスは、その関数を通るすべての記録経路に関する情報を提供します。属性メトリックスは、関数を通る特定の経路に関する情報を提供します。その情報は、どれだけのメトリックが特定の関数呼び出しが原因で発生したかを示します。その呼び出しに関係する 2 つの関数は、呼び出し元および呼び出し先として表されます。呼び出しツリー内のそれぞれの関数では次のことが言えます。

- 関数の呼び出し元の属性メトリックスは、その関数の包括的メトリックのうち、各呼び出し元からの呼び出しが原因になっているメトリックスを示します。呼び出し元の属性メトリックスを合計したものが、関数の包括的メトリックです。

- 関数の呼び出し先の属性メトリックスは、その関数の包括的メトリックのうち、各呼び出し先への呼び出しが原因になっているメトリックスを示します。それらの合計に、その関数の排他的メトリックを加えたものが、その関数の包括的メトリックに等しくなります。

メトリックス間の関係は、次の等式で表すことができます。

$$\sum_{\text{呼び出し元}} \text{属性メトリック} = \text{包括的メトリック} = \left(\sum_{\text{呼び出し先}} \text{属性メトリック} + \text{排他的メトリック} \right)$$

呼び出し元または呼び出し先の属性メトリックスと包括的メトリックスを比較すると、さらに情報が得られます。

- 呼び出し元の属性メトリックとその包括的メトリックの差は、ほかの関数への呼び出しおよびその呼び出し元自体の動作が原因で発生したメトリックを示します。
- 呼び出し先の属性メトリックとその包括的メトリックの差は、その呼び出し先の包括的メトリックのうち、ほかの関数からのその呼び出し先への呼び出しが原因で発生したメトリックを示します。

プログラムのパフォーマンス改善が可能な場所を見つける方法には、次のものがあります。

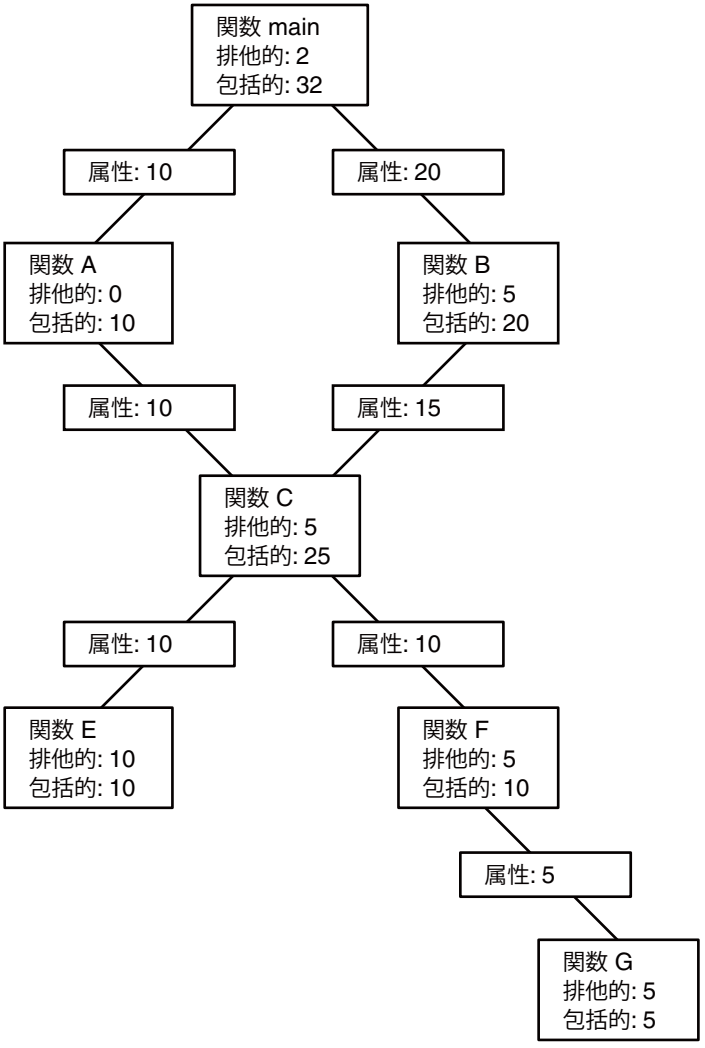
- 排他的メトリックスを使用して、メトリック値が大きい関数を見つけます。
- 包括的メトリックスを使用して、プログラム内のどの呼び出しシーケンスが大きなメトリック値の原因になっているかを判定します。
- 属性メトリックスを使用して、大きなメトリック値の原因になっている特定の 1 つまたは複数の関数に対する呼び出しシーケンスをトレースします。

属性メトリックスの解釈: 例

図 2–1 は、排他的メトリックス、包括的メトリックス、属性メトリックスを完全な呼び出しツリーで表しています。ここでは、中央の関数の関数 C に注目します。

プログラムの擬似コードは、図のあとに示されています。

図 2–1 排他的、包括的、属性メトリックスを示す呼び出しツリー



Main 関数は、関数 A および関数 B を呼び出し、Main 関数の包括的メトリックのうちの関数 A の 10 単位と関数 B の 20 単位の原因となっています。これらは Main 関数の呼び出し先の属性メトリックです。その合計 (10+20) に Main 関数の排他的メトリックを加算すると、Main 関数の包括的メトリック (32) に等しくなります。

関数 A は関数 C の呼び出しにすべての時間を使用するため、排他的メトリックスは 0 単位です。

関数 C は、関数 A および関数 B の 2 つの関数によって呼び出され、関数 C の包括的メトリックのうちの関数 A の 10 単位と関数 B の 15 単位の原因になっています。これらは呼び出し元の属性メトリックスです。その合計 (10+15) は、関数 C の包括的メトリック (25) に等しくなります。

呼び出し元の属性メトリックは、関数 A および関数 B の包括的メトリックスと排他的メトリックスの差に等しくなり、それぞれが関数 C のみを呼び出すことを意味します (実際のところ、各関数はほかの関数を呼び出すことがありますが、その時間はかなり短いため、実験には現れません)。

関数 C は、関数 E および関数 F の 2 つの関数を呼び出し、関数 C の包括的メトリックのうちの関数 E の 10 単位と関数 F の 10 単位の原因となっています。これらは呼び出し先の属性メトリックスです。その合計 (10+10) に関数 C の排他的メトリック (5) を加算すると、関数 C の包括的メトリック (25) に等しくなります。

呼び出し先の属性メトリックと呼び出し先の包括的メトリックは、関数 E でも関数 F でも同じです。これは、関数 E と関数 F のどちらも、関数 C によってのみ呼び出されることを意味します。排他的メトリックおよび包括的メトリックは、関数 E では同じですが、関数 F では異なります。この理由は、関数 F は別の関数である関数 G を呼び出しますが、関数 E は関数 G を呼び出さないためです。

このプログラムの擬似コードを次に示します。

```
main() {
   A();
   /Do 2 units of work;/
   B();
}

A() {
   C(10);
}

B() {
   C(7.5);
   /Do 5 units of work;/
   C(7.5);
}

C(arg) {
      /Do a total of "arg" units of work, with 20% done in C itself,
      40% done by calling E, and 40% done by calling F./
}
```

関数レベルのメトリックスに再帰が及ぼす影響

直接または間接のどちらの場合も、再帰関数呼び出しがあると、メトリックスの計算が複雑になります。パフォーマンスアナライザは、関数の呼び出しごとのメトリックスではなく、関数全体としてのメトリックスを表示します。このため、一連

の再帰的な呼び出しのメトリックスを 1 つのメトリックに要約する必要があります。この要約によって、呼び出しスタックの最後の関数 (リーフ関数) から計算される排他的メトリックスが影響を受けることはありませんが、包括的および属性メトリックスは影響を受けます。

包括的メトリックスは、イベントのメトリックと呼び出しスタック内の関数の包括的メトリックを合計することによって計算されます。再帰呼び出しスタックでメトリックが複数回カウントされないようにするには、イベントのメトリックが、同じ関数の包括的メトリックに 1 回だけ加算されるようにします。

属性メトリックスは、包括的メトリックスから計算されます。もっとも簡単な再帰では、再帰関数は、それ自身ともう 1 つの関数 (呼び出しを開始する関数) の 2 つの呼び出し元を持ちます。最後の呼び出しですべての動作を終えた場合、再帰関数の包括的メトリックの原因になるのは、その再帰関数であり、呼び出しを開始した関数ではありません。この現象が起こるのは、メトリックが複数回カウントされるのを避けるために、再帰関数の上位の呼び出しすべてについて、包括的メトリックがゼロと見なされるためです。ただし、呼び出しを開始した関数は、再帰呼び出しであるために、呼び出し先としての再帰関数の包括的メトリックの一部の原因になります。

◆ ◆ ◆ 第 3 章

パフォーマンスデータの収集

パフォーマンス解析の第一段階は、データ収集です。この章では、データ収集のための準備、収集データの格納場所、データの収集方法、およびデータ収集の管理方法について説明します。データそのものの詳細については、第 2 章「パフォーマンスデータ」を参照してください。

カーネルからデータを収集するには、`er_kernel` という別のツールが必要です。詳細は、第 9 章「カーネルプロファイリング」を参照してください。

この章では、次の内容について説明します。

プログラムのコンパイルとリンク

プログラムのコンパイル時には、ほとんどのコンパイラオプションを使用してデータの収集および解析を行うことができますが、収集対象とパフォーマンスアナライザでの表示対象に影響するオプションがいくつかあります。プログラムのコンパイルとリンクを行う際に考慮すべき事柄について、このあとの各項で説明します。

ソースコード情報

注釈付きの「ソース」および「逆アセンブリ」の解析にソースコードを表示し、「行」解析にソース行を表示するには、`-g` コンパイラオプション (C++ でフロントエンドインライン化を有効にするには `-g0`) で対象のソースファイルをコンパイルし、デバッグシンボル情報を作成します。デバッグシンボル情報の形式は、`-xdebugformat=(dwarf|stabs)` によって、DWARF2 またはスタブに指定することができます。デフォルトのデバッグ形式は `dwarf` です。

データ空間プロファイルを許可するデバッグ情報を含むコンパイルオブジェクトを準備するには、`-xhwcprof` と任意の最適化レベルを指定してコンパイルします。この方法は現在のところ、SPARC プロセッサだけが対象です。現在は、最適化を行わないと、この機能は使用できません。「データオブジェクト」解析でプログラムデータオブジェクトを表示するには、`-g` (または C++ の場合は `-g0`) も追加して十分なシンボル情報を取得します。

DWARF 形式のデバッグ用シンボルで構築された実行可能ファイルやライブラリには、構成要素である各オブジェクトファイルのデバッグシンボルのコピーが自動的に取り込まれます。スタブ形式のデバッグ用シンボルを使用して構築された実行可能ファイルとライブラリの場合、デバッグシンボルのリンク時に `-xs` オプションが指定され、各種のオブジェクトファイルおよび実行可能ファイル内にスタブシンボルが残されていれば、構成要素である各オブジェクトファイルのデバッグシンボルのコピーが取り込まれます。この情報の取り込みは、オブジェクトファイルを移動したり、削除したりする必要がある場合に特に有用です。すべてのデバッグ用シンボルが実行可能ファイルとライブラリ自体にあるので、実験とプログラム関連ファイルを別の場所に容易に移動できます。

静的リンク

プログラムをコンパイルするときに、`-dn` および `-Bstatic` コンパイラオプションを使用して動的リンクを無効にしないでください。完全に静的にリンクされたプログラムのデータを収集しようとしても、コレクタからエラーメッセージが返され、データは収集されません。このエラーが発生する原因は、コレクタを実行したときに、コレクタライブラリが動的に読み込まれるためです。

システムライブラリを静的リンクしないでください。システムライブラリを静的リンクしてしまうと、トレースデータを何も収集できなくなることがあります。また、コレクタライブラリ `libcollector.so` へもリンクしないでください。

共有オブジェクトの処理

通常、`collect` コマンドではターゲットのアドレス空間に含まれているすべての共有オブジェクトのデータが、初期ライブラリリストに含まれているものか、`dlopen()`

によって明示的に読み込まれたものかにかかわらず、収集されます。ただし、特定の条件では一部の共有オブジェクトのプロファイリングが行われないことがあります。

- 遅延読み込みにより指定プログラムが呼び出された場合。この場合、起動時にはライブラリが読み込まれず、`dlopen()` の呼び出しによって明示的にも読み込まれないため、共有オブジェクトは実験に含まれず、共有されたオブジェクトからの PC はすべて `<Unknown>` 関数にマップされます。対策として、環境変数 `LD_BIND_NOW` を設定すると、起動時にライブラリが強制的に読み込まれます。
- 実行可能ファイルが `-B` オプションを指定して構築された場合。この場合、そのオブジェクトは、`dlopen()` の動的リンカーエントリポイントに固有の呼び出しによって読み込まれ、libcollector 割り込みはバイパスされます。共有オブジェクト名は実験には含まれず、共有されたオブジェクトからの PC はすべて `<Unknown>()` 関数にマップされます。対策は、`-B` オプションを使用しないことです。

コンパイル時の最適化

何らかのレベルの最適化を有効にしてプログラムをコンパイルすると、コンパイラにより実行順序が変更され、プログラム内の行の順序と実行順序が厳密に一致しなくなることがあります。この場合、パフォーマンスアナライザは、最適化されたコードについて収集された実験データを解析できますが、しばしば、逆アセンブリレベルでパフォーマンスアナライザが提供するデータを元のソースコード行に対応付けることが困難になります。また、コンパイラが末尾呼び出しの最適化を行う場合には、呼び出しシーケンスが予想とは異なっているように見えることがあります。詳細は、174 ページの「末尾呼び出しの最適化」を参照してください。

Java プログラムのコンパイル

`javac` コマンドによる Java プログラムのコンパイルに特別なアクションは不要です。

データ収集と解析のためのプログラムの準備

データ収集と解析の準備のために、プログラムに対して特別な作業を行う必要はありません。次の処理のうち、いずれか 1 つでも行うプログラムの場合には、後述の該当する節を読んでください。

- シグナルハンドラをインストールする
- システムライブラリを明示的かつ動的に読み込む
- 関数を動的にコンパイルする
- プロファイルする派生プロセスを作成する
- 非同期 I/O ライブラリを使用する
- プロファイルタイマーまたはハードウェアカウンタ API を直接使用する

- `setuid`(2) を呼び出すか、`setuid` ファイルを実行する

また、データ収集をプログラムから制御する場合も、該当する節を読んでください。

動的割り当てメモリーの利用

多くのプログラムは、次のような機能を使用して、動的に割り当てられたメモリーに依存しています。

- `malloc`、`valloc`、`alloca` (C/C++)
- `new` (C++)
- スタック局所変数 (Fortran)
- `MALLOC`、`MALLOC64` (Fortran)

メモリー割り当ての方式の説明で、初期値が設定されることが明示的に記述されている場合を除き、動的に割り当てられるメモリーの初期値にプログラムが依存しないよう注意する必要があります。例としては、`malloc` (3C) のマニュアルページにある `calloc` と `malloc` の説明を比較してください。

動的に割り当てられるメモリーを使用するプログラムを単独で実行すると、正常に機能しているように見えることがありますが、パフォーマンスデータの収集を有効にした状態で実行すると、問題が起きることがあります。この場合、浮動小数点演算の予期しない動作、セグメント例外、アプリケーション固有のエラーメッセージなどが発生する可能性があります。

こうした症状は、アプリケーションが単独で実行されたときには、初期化されていないメモリーの値が動作に影響ないものであっても、パフォーマンスデータの収集ツールとの組み合わせで実行されたときに別の値が設定されることによって、発生する場合があります。この場合は、パフォーマンスツールの問題ではありません。動的に割り当てられるメモリーの内容に依存するアプリケーションにはすべて、潜在的なバグがあります。オペレーティングシステムにより動的に割り当てられるメモリーの内容は、ドキュメントに明確に記載されている場合を除いて、どのような値をとる可能性もあります。現在のオペレーティングシステムが動的に割り当てられるメモリーに必ず特定の値を設定するようになっていたとしても、将来オペレーティングシステムのリリースが変わったとき、あるいはプログラムを別のオペレーティングシステムに移植した場合には、こうした潜在的な問題によって、予期しない動作が発生する可能性があります。

次のツールが、こうした潜在的な問題の発見に役立ちます。

- `f95 -xcheck=init_local`

  詳細は、『Fortran ユーザーズガイド』または `f95`(1) のマニュアルページを参照してください。

- `lint` ユーティリティー

詳細は、『C ユーザーズガイド』または lint(1) のマニュアルページを参照してください。

- dbx 下での実行時チェック

  詳細は、『dbx コマンドによるデバッグ』または dbx(1) のマニュアルページを参照してください。

- Rational Purify

システムライブラリの使用

さまざまなコレクタは、システムライブラリの関数に割り込んでトレースデータを収集し、データ収集の整合性を保証します。コレクタがライブラリ関数の呼び出しに割り込む状況を次に示します。

- 同期待ちトレースデータの収集。Solaris 10 OS では、コレクタは Solaris C ライブラリ libc.so からの関数に割り込みます。
- ヒープトレースデータの収集。コレクタは、malloc、realloc、memalign、free の関数に割り込みます。これらの関数は、C 標準ライブラリ libc.so のほか、libmalloc.so や libmtmalloc.so などのライブラリにあります。
- MPI トレースデータの収集。コレクタは、Solaris MPI ライブラリからの関数に割り込みます。
- 時間データの完全性の確保。コレクタは setitimer に割り込み、プログラムがプロファイルタイマーを使用しないようにします。
- ハードウェアカウンタデータの完全性の確保。コレクタは、ハードウェアカウンタライブラリ libcpc.so からの関数に割り込み、プログラムがカウンタを使用しないようにします。プログラムからこのライブラリの関数への呼び出しは、値 -1 を返します。
- 派生プロセスに対するデータ収集の有効化。コレクタは、fork(2)、fork1(2)、vfork(2)、fork(3F)、posix_spawn(3p)、posix_spawnp(3p)、system(の関数とそのバリアントに割り込みます。vfork の呼び出しは、内部で fork1 の呼び出しに置き換えられます。これらの割り込み処理が行われるのは、collect コマンドの場合だけです。
- コレクタによる SIGPROF シグナルおよび SIGEMT シグナルの処理の保証。コレクタは sigaction に割り込み、シグナルハンドラがこれらのシグナル用のプライマリシグナルハンドラであるかどうかを確認します。

次のような環境では、割り込みが成功しません。

- 割り込み対象関数が入っているライブラリとプログラムを静的にリンクした場合。
- コレクタライブラリが事前読み込みされていない実行中アプリケーションに dbx を接続した場合。

- これらのライブラリのいずれか 1 つを動的に読み込み、このライブラリの中でだけ検索することによってシンボルを解決する場合。

コレクタが割り込み処理を行えなかった場合には、パフォーマンスデータが消去されたり無効になったりする可能性があります。

`er_sync.so`、`er_heap.so`、および `er_mpview`*n*`.so` (*n* は MPI バージョンを示す) ライブラリは、それぞれ同期待ちトレースデータ、ヒープトレースデータ、または MPI トレースデータが要求された場合のみ読み込まれます。

シグナルハンドラの使用

コレクタは、2 つのシグナルを使用して、プロファイルデータを収集します。`SIGPROF` はすべての実験に使用され、`SIGEMT` (Solaris プラットフォームの場合) または `SIGIO` (Linux プラットフォームの場合) はハードウェアカウンタ実験にのみ使用されます。コレクタは、これらの各シグナルについてシグナルハンドラをインストールします。シグナルハンドラは自身のシグナルをインターセプトして処理しますが、ほかのシグナルは、インストールされているほかのシグナルハンドラに引き渡します。プログラムがこれらのシグナル用に独自のシグナルハンドラをインストールすると、コレクタは自分のシグナルハンドラをプライマリハンドラとして再インストールし、それによって完全なパフォーマンスデータが確保されます。

`collect` コマンドでは、ユーザー指定のシグナルを使用してデータ収集の一時停止と再開、および標本の記録を行えます。それらのシグナルはコレクタによって保護されませんが、ユーザーハンドラがインストールされている場合は、実験に警告が書き出されます。コレクタとアプリケーションによる指定シグナルの使用が互いに競合しないように、ユーザーが責任を持って確認する必要があります。

コレクタによってインストールされたシグナルハンドラは、システムコールがシグナル配信のために中断されないようにするためのフラグを設定します。このフラグ設定方法では、プログラムのシグナルハンドラがシステムコールの中断を許可するようにフラグを設定した場合に、プログラムの動作が変化する可能性があります。動作が変化する重要な例として、非同期キャンセル処理に `SIGPROF` を使用し、システムコールの中断を行う非同期入出力ライブラリ `libaio.so` が挙げられます。コレクタライブラリ `libcollector.so` がインストールされている場合、キャンセルシグナルの到着は常に、非同期入出力操作を取り消すには遅すぎます。

コレクタライブラリを事前読み込みしないままプロセスに `dbx` を接続してパフォーマンスデータ収集を有効にし、そのあとでプログラムが自分のシグナルハンドラをインストールすると、コレクタは自分のシグナルハンドラを再インストールしません。この場合、プログラムのシグナルハンドラは、パフォーマンスデータが失われないように、`SIGPROF` および `SIGEMT` シグナルが確実に渡されるようにする必要があります。プログラムのシグナルハンドラがシステムコールを中断した場合、プログラムの動作とプロファイルの動作は、コレクタライブラリが事前読み込みされた場合の動作と異なります。

setuid と setgid の使用

動的ローダーによって課される制約により、setuid(2) を使用してパフォーマンスデータを収集することが困難になります。プログラムが setuid を呼び出すか、setuid ファイルを実行する場合、コレクタは新しいユーザー ID に必要なアクセス権がないために、実験ファイルに書き込めない可能性が高くなります。

collect コマンドは、ターゲットのアドレス空間 (LD_PRELOAD) に共有ライブラリ libcollector.so を挿入して動作します。実行可能ファイルが setuid または setgid を呼び出す、または setuid または setgid を呼び出す子孫プロセスを作成する場合、その実行可能ファイルから collect コマンドを起動すると、いくつかの問題が発生する可能性があります。root ではないユーザーとして実験を実行すると、共有ライブラリが信頼できるディレクトリにインストールされていないために収集が失敗します。対策としては、root ユーザーとして実験を実行するか、crle(1) を使用してアクセス権を与えます。セキュリティバリアーを迂回するときには十分な注意を払い、各自のリスクで行ってください。

collect コマンドを実行するとき、umask は、自分、exec() によって実行されるプログラムの setuid 属性および setgid 属性によって設定されているユーザーまたはグループ、およびプログラム自体によって設定されるユーザーまたはグループに対して、書き込みアクセスを許可するように設定されている必要があります。マスクが正しく設定されていない場合、一部のファイルが実験に対して書き込まれず、実験が処理不能になることがあります。ログファイルが書き込み可能な場合、実験の処理を試みたときにエラーが示されます。

ターゲット自身が UID または GID を設定する何らかのシステムコールを発行した場合、自分の umask を変更してから fork を行うか、ほかの実行可能ファイルを exec() で実行した場合、または crle を使用してランタイムリンカーが共有オブジェクトを検索する方法が構成された場合には、ほかの問題が発生する可能性があります。

実効 GID を変更するターゲット上で実験が root として開始された場合、実験の終了時に実行される er_archive プロセスが失敗します。これは、このプロセスには「信頼できる」にマークされていない共有ライブラリが必要なためです。この場合、実験の終了直後に、実験が記録されたマシン上で、er_archive ユーティリティー (または er_print ユーティリティーか analyzer コマンド) を手作業で明示的に実行できます。

データ収集のプログラム制御

プログラムからデータ収集を制御するために、コレクタ共有ライブラリ libcollector.so に含まれているいくつかの API 関数を使用できます。これらの関数は C で記述されており、Fortran インタフェースも用意されています。ライブラリとともに提供されるヘッダーファイルに、C インタフェースと Fortran インタフェースの両方が定義されています。

API 関数は、次のように定義されます。

```
void collector_sample(char *name);
void collector_pause(void);
void collector_resume(void);
void collector_terminate_expt(void);
```

CollectorAPI クラスに、Java プログラム用の類似の機能が用意されており、これについては、50 ページの「Java インタフェース」で説明しています。

C/C++ インタフェース

collectorAPI.h を取り込み、配下の libcollector.so API 関数の存在をチェックする実際の関数を含む、-lcollectorAPI にリンクすることにより、コレクタ API の C/C++ インタフェースにアクセスできます。

有効な実験がない場合、API 呼び出しは無視されます。

Fortran インタフェース

Fortran API の libfcollector.h ファイルは、ライブラリへの Fortran インタフェースを定義します。このライブラリを使用するには、アプリケーションを -lcollectorAPI にリンクする必要があります。このライブラリには、下位互換性を維持するため、-lfcollector というもう 1 つの名前も用意されています。動的関数とスレッドによる呼び出しの一時停止と再開を除けば、Fortran API は C/C++ API と同じ機能を提供します。

Fortran の場合、API 関数を使用するには、次の文を挿入します。

```
include "libfcollector.h"
```

注 – どんな言語を使用している場合も、プログラムを -lcollector とリンクしないでください。リンクした場合、コレクタが予期しない動作をすることがあります。

Java インタフェース

次の文を使用して、CollectorAPI クラスをインポートし、Java API にアクセスできます。ただし、アプリケーションは */ installation_directory*/lib/collector.jar (*installation_directory* は Oracle Solaris Studio ソフトウェアがインストールされているディレクトリです) をクラスパスが指している状態で呼び出される必要があります。

```
import com.sun.forte.st.collector.CollectorAPI;
```

Java CollectorAPI メソッドは、次のように定義されます。

```
CollectorAPI.sample(String name)
CollectorAPI.pause()
CollectorAPI.resume()
CollectorAPI.terminate()
```

Java API には、動的関数 API 以外の C および C++ API と同じ関数が含まれています。

C インクルードファイルの `libcollector.h` には、データが収集されていないときには実際の API 関数への呼び出しを迂回するマクロが含まれています。この場合、関数は動的に読み込まれません。ただし、一部の条件では適切に機能しないことがあるため、これらのマクロを使用するのは危険です。`collectorAPI.h` はマクロを使用していないため、このファイルを使用する方が安全です。このファイルでは、関数が直接参照されます。

Fortran API サブルーチンはパフォーマンスデータが収集されているときには C API 関数を呼び出し、そうでないときには復帰します。チェック処理のオーバーヘッドは非常に小さいので、プログラムのパフォーマンスにはあまり影響がないはずです。

パフォーマンスデータを収集するには、この章で後述するように、コレクタを使用してプログラムを実行する必要があります。API 関数への呼び出しを挿入することによって、データ収集が有効になることはありません。

マルチスレッドプログラムで API 関数を使用する場合は、これらの関数が 1 つのスレッドによってのみ呼び出されるようにする必要があります。API 関数は、個別のスレッドではなくプロセスに適用される動作を実行します。各スレッドが API 関数を呼び出すと、記録されたデータが期待したものにならない可能性があります。たとえば、あるスレッドが `collector_pause()` や `collector_terminate_expt()` を呼び出したときに、ほかのスレッドがまだプログラム内のそのポイントに達していない場合、すべてのスレッドについて収集が一時停止または停止され、この API 呼び出しの前にコードを実行していたスレッドのデータが失われる可能性があります。

C、C++、Fortran、および Java API 関数

ここでは、データ収集に関係する API 関数について説明します。

- **C** および **C++**: `collector_sample(char *name)`

  Fortran: `collector_sample(string name)`

  Java: `CollectorAPI.sample(String name)`

  標本パケットを記録し、その標本に指定された名前または文字列をラベルとして付けます。ラベルは、「パフォーマンスアナライザ」の「イベント」タブで表示されます。Fortran の引数 `string` の型は、`character` です。

  標本ポイントに含まれるデータは、プロセスに関するものであり、個々のスレッドに関するものではありません。マルチスレッドアプリケーションの場合、`collector_sample()` API 関数は、標本の記録中に別の呼び出しが行われても、1 つの標本だけが書き込まれるようにします。記録される標本の数は、呼び出しを行うスレッドの数よりも少なくなります。

パフォーマンスアナライザは、別々のメカニズムによって記録された標本同士を区別しません。API 呼び出しによって記録された標本だけを見たい場合には、パフォーマンスデータの記録時にほかのあらゆる標本モードを停止します。

- **C、C++、Fortran**: `collector_pause()`

  Java: `CollectorAPI.pause()`

  実験へのイベント固有データの書き込みを停止します。実験はオープン状態のままであり、大域データの書き込みは続けられます。有効な実験がない場合やデータの記録がすでに停止されている場合には、呼び出しは無視されます。この関数は、たとえすべてのイベント固有データの書き込みが `collector_thread_resume()` 関数によって特定のスレッドに対して有効にされていたとしても、その書き込みを停止します。

- **C、C++、Fortran**: `collector_resume()`

  Java: `CollectorAPI.resume()`

  `collector_pause()` を呼び出したあとに、実験へのイベント固有データの書き込みを再開します。有効な実験がない場合やデータの記録が有効である場合には、呼び出しは無視されます。

- **C、C++、Fortran**: `collector_terminate_expt()`

  Java: `CollectorAPI.terminate`

  データの収集対象である実験を終了します。それ以後のデータは収集されませんが、プログラムの実行は正常に続行されます。有効な実験がない場合は、呼び出しは無視されます。

動的な関数とモジュール

使用している C または C++ プログラムが、関数を動的にコンパイルしてデータ空間に取り込む場合、動的関数やモジュールのデータをパフォーマンスアナライザに表示するには、コレクタに情報を与える必要があります。この情報は、コレクタ API 関数の呼び出しによって渡されます。API 関数の定義は、次のとおりです。

```
void collector_func_load(char *name, char *alias,
    char *sourcename, void *vaddr, int size, int lntsize,
    Lineno *lntable);
void collector_func_unload(void *vaddr);
```

Java HotSpot 仮想マシンによってコンパイルされる Java メソッドには別のインタフェースが使用されるので、これらの API 関数を使用する必要はありません。Java インタフェースは、コンパイルされたメソッドの名前をコレクタに知らせます。Java コンパイル済みメソッドの関数データと注釈付き逆アセンブリのリストを見ることはできますが、注釈付きソースリストを見ることはできません。

次に、API 関数について説明します。

collector_func_load()

実験への記録のため、動的にコンパイルされた関数に関する情報をコレクタに渡します。パラメータリストを次の表に示します。

表 3–1 collector_func_load() のパラメータリスト

パラメータ	定義
name	パフォーマンスツールで使用する、動的にコンパイルされた関数の名前。実際の関数名でなくてもかまいません。この名前は関数の通常の命名規則に従っている必要はありませんが、空白文字や引用符は含めないようにします。
alias	関数の説明に使用する任意の文字列。NULL も使用できます。この文字列が解釈の対象となることはありません。空白文字を含めることができます。アナライザの「概要」タブに表示されます。何の関数であるか、またはなぜ関数が動的に構築されたかを示すために使用できます。
sourcename	関数の構築元であるソースファイルのパス。NULL も使用できます。このソースファイルは、注釈付きソースリストに使用されます。
vaddr	関数が読み込まれたアドレス。
size	バイト数による関数のサイズ。
lntsize	行番号テーブルのエントリの数を示すカウント。行番号情報がない場合には、ゼロとなります。
lntable	lntsize エントリが入っているテーブル。各エントリは、整数対です。第 1 整数はオフセット、第 2 整数は行番号です。あるエントリのオフセットと次のエントリのオフセットとの間の命令はすべて、最初のエントリの行番号に対応します。オフセットは数値の昇順にする必要がありますが、行番号の順序は任意です。lntable が NULL の場合、関数のソースリストは利用できませんが、逆アセンブリリストは利用できます。

collector_func_unload()

アドレス vaddr にある動的関数が読み込み解除されたことをコレクタに通知します。

データ収集に関する制限事項

ここでは、ハードウェア、オペレーティングシステム、プログラムの実行方法、またはコレクタ自体によって課される、データ収集の制限事項について説明します。

異なる種類のデータを同時に収集することについて制限はありません。カウントデータを除いて、任意の種類のデータを同時に収集できます。

コレクタは、最大 16K ユーザースレッドをサポートできます。この数を超えるスレッドからのデータは破棄され、コレクタエラーが生成されます。サポートされるスレッド数を増やすには、SP_COLLECTOR_NUMTHREADS 環境変数に設定されている値を増やします。

デフォルトで、コレクタは、最大 256 フレームの深さまでスタックを収集します。これよりも深いスタックをサポートするには、SP_COLLECTOR_STACKBUFSZ 環境変数に設定されている値を増やします。

時間ベースのプロファイルに関する制限事項

プロファイル間隔の最小値と、プロファイルに使用する時間の分解能は、オペレーティング環境により異なります。最大値は 1 秒に設定されています。プロファイル間隔の値は、時間の分解能のもっとも近い倍数に切り捨てられます。最小値および最大値と時間の分解能を検索するには、引数を付けずに collect コマンドを入力します。

時間プロファイルによる実行時のひずみと実行時間の拡大

時間ベースのプロファイルでは、SIGPROF シグナルがターゲットに送られたときにデータが記録されます。それによってシグナルを処理するための実行時間の拡大が発生し、呼び出しスタックが展開されます。呼び出しスタックが深く、シグナルが頻繁なほど、実行時間の拡大は大きくなります。一定の範囲までは、時間ベースのプロファイルにより、ある程度の実行時間の拡大が生じますが、これはもっとも深いスタックで実行するプログラムの各部分の実行時間の拡大が大きくなることから生まれます。

可能な場合、デフォルト値は正確なミリ秒数でなく、システムクロックとの相関を回避するために、正確な数値と多少異なる値 (たとえば、10.007 ms または 0.997 ms など) に設定されます。システムクロックとの相関はデータのひずみをもたらす場合もあります。SPARC プラットフォームでは、同じ方法でカスタム値を設定してください (Linux プラットフォーム上では不可能)。

トレースデータの収集に関する制限事項

コレクタライブラリ libcollector.so が事前読み込みされていないかぎり、すでに稼働中のプログラムからはトレースデータを収集できません。詳細は、83 ページの「動作中のプロセスからのトレースデータの収集」を参照してください。

トレースによる実行時のひずみと実行時間の拡大

データのトレースは、トレースされるイベント数に比例して実行時間を拡大させます。時間ベースのプロファイルを同時に行うと、イベントのトレースに起因する実行時間の拡大により、時間データにひずみが生じます。

ハードウェアカウンタオーバーフローのプロファイルに関する制限事項

ハードウェアカウンタオーバーフローのプロファイルには、次のような制限があります。

- ハードウェアカウンタオーバーフローデータの収集を行えるのは、ハードウェアカウンタが用意されていてオーバーフロープロファイルをサポートしているプロセッサにおいてだけです。そのほかのシステムでは、ハードウェアカウンタオーバーフローのプロファイルは無効です。UltraSPARC III プロセッサファミリより前の UltraSPARC プロセッサは、ハードウェアカウンタオーバーフローのプロファイルをサポートしません。
- Solaris OS を実行しているシステムで、cpustat(1) コマンドを実行中にハードウェアカウンタオーバーフローのデータを収集することはできません。これは、cpustat がカウンタを制御しており、ユーザープロセスがカウンタを利用できないためです。データ収集中に cpustat を起動すると、ハードウェアカウンタオーバーフローのプロファイルは終了され、実験にエラーが記録されます。
- ハードウェアカウンタオーバーフローのプロファイルを行う場合、独自のコードでハードウェアカウンタを使用することはできません。コレクタは libcpc ライブラリ関数に割り込み、コレクタからの呼び出しではなかった場合、-1 の戻り値で復帰します。ハードウェアカウンタへのアクセスの取得に失敗した場合に正常に機能するようにプログラムをコーディングする必要があります。この処理を行うようにコーディングされていない場合、ハードウェアカウンタプロファイル時に、スーパーユーザーがカウンタを使用するシステム全体のツールを起動した場合、またはそのシステムでカウンタがサポートされていない場合にプログラムが失敗します。
- dbx をプロセスに接続することによって、ハードウェアカウンタライブラリを使用している実行中プログラムのハードウェアカウンタデータを収集しようとすると、実験が壊れることがあります。

注 – 使用可能なすべてのカウンタの一覧を表示するには、引数を指定せずに collect コマンドを実行します。

ハードウェアカウンタオーバーフローのプロファイルによる実行時のひずみと実行時間の拡大

ハードウェアカウンタオーバーフローのプロファイルは、SIGEMT シグナル (Solaris プラットフォームの場合) または SIGIO シグナル (Linux プラットフォームの場合) がターゲットへ配信された時点のデータを記録します。それによってシグナルを処理するための実行時間の拡大が発生し、呼び出しスタックが展開されます。時間ベースのプロファイルと違い、ハードウェアカウンタによっては、プログラムのさ

まざまな部分がその他の部分より高速にイベントを生成する場合があり、そのコード部分に実行時間の拡大が生じます。そのようなイベントを非常に高速に生成するプログラムの一部で大きなひずみが生じる場合があります。同様に、あるスレッドでは、ほかのスレッドと不均等にイベントが生成されるものがあります。

派生プロセスのデータ収集における制限事項

派生プロセスのデータ収集には、いくつかの制限事項があります。

コレクタでの派生プロセスすべてについてデータを収集するには、次のいずれかのオプションとともに `collect` コマンドを使用する必要があります。

- `-F on` オプションを使用すると、`fork` とそのバリアント、および `exec` とそのバリアントへの呼び出しについて、自動的にデータを収集できます。
- `-F all` オプションを使用すると、コレクタは、`system`、`popen`、`posix_spawn`(3p)、`posix_spawnp`(3p)、および `sh` の呼び出しに起因するものを含むすべての派生プロセスを追跡します。
- `-F '=`*regexp*`'` オプションを使用すると、名前または系統が指定した正規表現と一致するすべての派生プロセスに関するデータを収集できます。

`-F` オプションについては、68 ページの「実験制御オプション」を参照してください。

OpenMP プロファイルに関する制限事項

プログラム実行中に OpenMP データを収集すると非常にコストが高くなる可能性があります。このコストを抑制するには、`SP_COLLECTOR_NO_OMP` 環境変数を設定します。この設定を行うと、プログラムの遅延は大幅に減少しますが、スレーブトレッドから呼び出し元へ、最終的には `main()` へ伝搬されるデータは、この変数がない場合には参照可能ですが、この変数を設定すると参照できなくなります。

このリリースでは、デフォルトで OpenMP 3.0 の新しいコレクタが有効になっています。このコレクタは、明示的なタスクを使用するプログラムのプロファイルを実行できます。以前のバージョンのコンパイラで構築されたプログラムは、`libmtsk.so` のパッチ適用済みバージョンが利用可能な場合にのみ、新しいコレクタでプロファイル可能です。このパッチ適用済みバージョンがインストールされていない場合、`SP_COLLECTOR_OLDOMP` 環境変数を設定し、データ収集が古いコレクタを使用するように切り替え可能です。

OpenMP プロファイル機能は Oracle Solaris Studio コンパイラ実行時のランタイムに依存しているため、Oracle Solaris Studio コンパイラでコンパイルされたアプリケーションに対してのみ使用できます。GNU コンパイラでコンパイルされたアプリケーションの場合、マシンレベルの呼び出しスタックのみが表示されます。

Java プロファイルに関する制限事項

Java プログラムのデータ収集には、次の制限事項があります。

- Java 2 Software Development Kit (JDK) の JDK 6、Update 18 またはそれ以降のバージョンを使用する必要があります。コレクタは、`JDK_HOME` 環境変数または環境変数のパスセットに含まれている JDK を最初に探します。`JAVA_PATH`これらの変数がいずれも設定されていない場合、使用されている環境の `PATH` に含まれている JDK を探します。`PATH` にも JDK が存在しない場合、`/usr/java/bin/java` の `java` 実行可能ファイルを探します。コレクタは、見つかった `java` 実行可能ファイルのバージョンが ELF 実行可能ファイルであるかどうかを検証し、そうでない場合は、使用した環境変数またはパスと、試みたフルパス名を示すエラーメッセージを出力します。
- ホットスポットコンパイルコードのソース行マッピングの詳細情報を取得するには、JDK 6、Update 20、JDK 7、build b85 Early Access リリース、またはそれ以降のバージョンを使用する必要があります。
- データを収集するには、`collect` コマンドを使用する必要があります。`dbx collector` サブコマンドは使用できません。
- JVM ソフトウェアを実行する派生プロセスを作成するアプリケーションはプロファイルできません。
- 一部のアプリケーションは純粋な Java ではなく、C または C++ アプリケーションで、`dlopen()` を起動して `libjvm.so` を読み込んでから、その中への呼び出しを行って JVM ソフトウェアを開始します。このようなアプリケーションのプロファイルを行うには、`SP_COLLECTOR_USE_JAVA_OPTIONS` 環境変数を設定し、`collect` コマンド行に `-j on` オプションを追加します。この場合は、`LD_LIBRARY_PATH` 環境変数を設定しないでください。

Java プログラミング言語で書かれたアプリケーションの実行時のひずみと実行時間の拡大

Java のプロファイリングでは、Java Virtual Machine Tools Interface (JVMTI) が使用され、実行のひずみと実行時間の拡大が発生する場合があります。

時間ベースのプロファイリングとハードウェアカウンタオーバーフローのプロファイルリングの場合、データ収集プロセスは JVM ソフトウェアへのさまざまな呼び出しを行い、プロファイリングイベントをシグナルハンドラ内で処理します。これらのルーチンのオーバーヘッドとディスクへの実験の書き込みコストにより、Java プログラムの実行時間の拡大が生じます。そのような実行時間の拡大は通常 10% より少なくなります。

収集データの格納場所

アプリケーションの 1 回の実行で収集されたデータを、実験と呼びます。実験を構成する一連のファイルは、ディレクトリに格納されます。実験の名前は、ディレクトリの名前です。

コレクタは実験データを記録するばかりでなく、プログラムが使用したロードオブジェクトの独自のアーカイブも作成します。これらのアーカイブには、すべてのオブジェクトファイルとそのロードオブジェクト内のすべての関数のアドレス、サイズ、名前、ロードオブジェクトのアドレス、その最終変更日時を示すタイムスタンプが含まれます。

デフォルトでは、実験は現在のディレクトリに格納されます。このディレクトリがネットワーク接続されたファイルシステム上にある場合は、ローカルのファイルシステム上にあるときよりもデータの格納に長い時間がかかり、パフォーマンスデータにひずみが生じる可能性があります。可能であれば、常に、実験はローカルのファイルシステムに記録するようにしてください。コレクタを実行するときに、格納場所を指定することができます。

派生プロセスの実験は、親プロセスの実験の内部に格納されます。

実験名

新しい実験のデフォルト名は、`test.1.er` です。接尾辞 `.er` は必須です。この接尾辞を持たない名前を指定すると、エラーメッセージが表示され、名前は受け付けられません。

experiment `.`*n*`.er` という形式の名前 (*n* は正の整数値) を選択すると、以後の実験ではコレクタによって名前の *n* が自動的に 1 ずつ増やされます。たとえば、`mytest.1.er` の次は `mytest.2.er`、その次は `mytest.3.er`、以下同様に名前が付けられます。コレクタはまた、実験がすでに存在する場合も *n* を増分し、すでに実験名が使用されている場合は、使用されていない実験名が見つかるまで *n* の増分を繰り返します。実験が存在していても実験名に *n* が含まれていない場合、コレクタはエラーメッセージを出力します。

実験はグループにまとめることができます。グループは実験グループファイル内に定義され、このファイルはデフォルトでは現在のディレクトリに格納されます。実験グループファイルは、1 行のヘッダー行のあとに 1 行につき 1 つの実験名が定義されているプレーンテキストファイルです。実験グループファイルのデフォルト名は `test.erg` です。名前の末尾が `.erg` でない場合、エラーが表示され、その名前は受け付けられません。実験グループを作成した後で、そのグループ名で実行したすべての実験は、そのグループに追加されます。

次に示す行を最初に持つプレーンテキストファイルを作成すると、実験グループを手動で作成できます。

```
#analyzer experiment group
```

このあとの行に実験の名前を追加します。ファイルの名前の最後は、.erg でなければなりません。

collect ユーティリティーの -g 引数を使用して、実験グループを作成することもできます。

派生プロセスの実験は、系統によって次のように命名されます。派生プロセスの実験名は、作成元の実験名の根幹部に下線、コード文字、および番号を追加したものになります。コード文字は、fork の場合は f、exec の場合は x、組み合わせの場合は c です。数字は、fork または exec のインデックスで、成功したかどうかには関係ありません。たとえば親プロセスの実験名が test.1.er の場合、3 回目の fork の呼び出しで作成された子プロセスの実験は test.1.er/_f3.er となります。この子プロセスが exec の呼び出しに成功した場合、新しい派生プロセスの実験名は test.1.er/_f3_x1.er となります。

実験の移動

別のコンピュータに実験を移動して解析する場合は、実験が記録されたオペレーティング環境に解析結果が依存することに注意してください。

アーカイブファイルには、関数レベルでメトリックスを計算してタイムラインを表示するのに必要な情報がすべて入っています。ただし、注釈付きソースコードや注釈付き逆アセンブリコードを調べるには、実験の記録時に使用されたものと同じバージョンのロードオブジェクトやソースファイルにアクセスできる必要があります。

パフォーマンスアナライザはソースファイル、オブジェクトファイル、および実行可能ファイルを次の場所で順に検索し、正しいベース名のファイルが見つかると検索を停止します。

- 実験の保管ディレクトリ
- 現在の作業ディレクトリ
- 実行可能ファイルまたはコンパイルオブジェクトに記録されている絶対パス名

検索順序を変更する、またはほかの検索ディレクトリを追加するには、アナライザの GUI を使用するか、setpath (135 ページの「setpath *path_list*」 を参照) および addpath (136 ページの「addpath *path_list*」 を参照) 指示を使用します。pathmap コマンドで検索を拡張することもできます。

プログラムに対応する、正しい注釈付きソースコードと注釈付き逆アセンブリコードが確実に表示されるようにするには、ソースコード、オブジェクトファイル、および実行可能ファイルを実験にコピーしてから、実験の移動やコピーを行います。オブジェクトファイルをコピーしたくない場合は、-xs を使用してプログラムをリンクし、ソース行とファイルの場所に関する情報が実行可能ファイルに挿入さ

れるようにします。collect コマンドの -A copy オプション、または dbx collector archive コマンドを使用すると、ロードオブジェクトを実験に自動的にコピーすることができます。

必要なディスク容量の概算

ここでは、実験を記録するために必要な空きディスク容量を概算するためのいくつかのガイドラインを示します。実験のサイズはデータパケットのサイズとその記録速度、プログラムが使用する LWP の数、およびプログラムの実行時間によって異なります。

データパケットには、イベント固有のデータと、プログラム構造に依存するデータ (呼び出しスタック) が含まれます。データのサイズはデータの種類に依存し、約 50 ～ 100 バイトです。呼び出しスタックのデータは すべての呼び出しの復帰アドレスで構成され、アドレス 1 個あたりのサイズは 4 バイト、64 ビットの実行可能ファイルではアドレス 1 個あたり 8 バイトです。LWP ごとにデータパケットが実験に記録されます。Java プログラムの場合、対象となる呼び出しスタックは Java 呼び出しスタックとマシン呼び出しスタックの 2 つがあるため、ディスクに書き込まれるデータが増えます。

プロファイルデータパケットが記録される頻度は、クロックデータのプロファイル間隔、ハードウェアカウンタデータのオーバーフロー値、および (関数のトレースの場合) トレース対象の関数の出現頻度によって決定されます。プロファイル間隔パラメータの選択は、データの品質や、データ収集オーバーヘッドに起因するプログラムパフォーマンスの偏向にも影響します。これらのパラメータ値が小さければ良い統計値が得られますが、オーバーヘッドは高くなります。プロファイル間隔とオーバーフロー値のデフォルト値は、良好な統計値を得ることとオーバーヘッドを抑えることの折衷点として慎重に選択されています。値が小さい場合、データ量が多いことを意味します。

時間ベースのプロファイル実験、またはハードウェアカウンタオーバーフロープロファイル実験で、プロファイル間隔が毎秒 100 サンプル前後、パケットサイズが小さな呼び出しスタックで 80 バイト、大きな呼び出しスタックで 120 バイトの範囲であれば、記録されるデータはスレッドあたり毎秒 10K バイト程度です。数百という深さを持つ呼び出しスタックを持つプログラムの場合は、この 10 倍以上の速度でデータが記録される可能性があります。

MPI トレース実験では、データ量はトレース対象の MPI 呼び出しごとに 100 - 150 バイトで、送信されるメッセージの数や呼び出しスタックの深さによって異なります。さらに、collect コマンドの -M オプションを使用しているときは、デフォルトで時間プロファイルが有効なため、時間プロファイル実験の推定値を加算します。-p off オプションで時間プロファイルを無効にすると、MPI トレースのデータ量を減らすことができます。

注 – コレクタは、MPI トレースデータを独自の形式 (`mpview.dat3`) と、VampirTrace OTF 形式 (`a.otf`、 `a.*.z`) の両方で格納します。OTF 形式のファイルを削除しても、アナライザに影響はありません。

実験サイズの概算では、アーカイブファイルに使用されるディスク容量も考慮する必要がありますが、通常その量は、必要となるディスク容量全体のごく一部です (前節参照)。必要なディスク容量のサイズを確定できない場合は、実験を短時間だけ行なってみてください。この実験からアーカイブファイルのサイズを取得し (データ収集時間とは無関係)、プロファイルファイルのサイズを確認することによって、完全な実験のサイズを概算できます。

コレクタは、ディスク容量を割り当てるだけでなく、ディスクにプロファイルデータを書き込む前に、そのデータを格納するためのバッファーをメモリー内に確保します。現在、こうしたバッファーのサイズを指定する方法はありません。コレクタがメモリー不足になった場合、収集するデータ量を減らすようにしてください。

現在利用できる容量より実験の格納に必要となる容量の方が大きいと思われる場合には、実行の全体でなく一部だけのデータを収集することを検討してください。実行の一部についてのデータを収集するには、`collect` コマンドを `-y` または `-t` オプションで実行するか、`dbx collector` サブコマンドを使用するか、またはコレクタ API への呼び出しをプログラムに挿入します。`collect` コマンドの `-L` オプションや、`dbx collector` サブコマンドを使用して、収集されるプロファイルおよびトレースデータの総量を制限することもできます。

データの収集

パフォーマンスアナライザでパフォーマンスデータを複数の方法で収集できます。

- コマンド行から `collect` コマンドを使用する (62 ページの「`collect` コマンドによるデータの収集」 および `collect`(1) のマニュアルページを参照)。`collect` コマンド行ツールは `dbx` よりもデータ収集のオーバーヘッドが小さいため、この方法のほうが適切な場合があります。
- Oracle Solaris Studio で、パフォーマンスアナライザの「パフォーマンスコレクタ」ダイアログボックスを使用する (パフォーマンスアナライザのオンラインヘルプの「Oracle Solaris Studio パフォーマンスコレクタダイアログボックスによるパフォーマンスデータの収集」を参照)。
- `dbx` コマンド行から `collector` コマンドを使用する (76 ページの「`dbx collector` サブコマンドによるデータの収集」 を参照)。

次のデータ収集機能は、Oracle Solaris Studio の「収集」ダイアログボックスと、collect コマンドでのみ使用できます。

- Java プログラムに関するデータの収集。dbx の collector コマンドで Java プログラムに関するデータの収集を試みた場合、実際に収集される情報は、Java プログラムではなく JVM ソフトウェアに関する情報です。
- 派生プロセスに関するデータの自動収集。

collect コマンドによるデータの収集

collect コマンドを使用してコマンド行からコレクタを実行するには、次のコマンドを使用します。

```
% collect collect-options program program-arguments
```

ここで、*collect-options* は collect コマンドのオプションで、*program* はデータ収集対象のプログラム名、*program-arguments* はプログラムの引数です。ターゲットプログラムは通常、バイナリ実行可能ファイルです。ただし、環境変数 SP_COLLECTOR_SKIP_CHECKEXEC を設定する場合は、ターゲットとしてスクリプトを指定できます。

collect-options を指定しなかった場合は、デフォルトで時間ベースのプロファイルが有効になり、プロファイル間隔は約 10 ミリ秒になります。

プロファイルに使用可能なオプションとハードウェアカウンタの名前の一覧を表示するには、引数を指定せずに collect コマンドを入力します。

```
% collect
```

ハードウェアカウンタの一覧については、27 ページの「ハードウェアカウンタオーバーフローのプロファイルデータ」を参照してください。55 ページの「ハードウェアカウンタオーバーフローのプロファイルに関する制限事項」も参照してください。

データ収集オプション

これらのオプションは、収集されるデータの種類を制御します。データの種類については、23 ページの「コレクタが収集するデータの内容」を参照してください。

データ収集オプションを指定しなかった場合、デフォルトは -p on で、デフォルトのプロファイル間隔 (約 10 ミリ秒) で時間ベースのプロファイルが行われます。このデフォルト設定は、-h オプションを使用することによってのみ無効にできます。

時間ベースのプロファイルを明示的に無効とし、すべてのトレースとハードウェアカウンタオーバーフロープロファイルを有効にしなかった場合、collect コマンドは警告メッセージを出力し、大域データだけを収集します。

-p *option*

時間ベースのプロファイルデータを収集します。*option* に使用できる値は次のとおりです。

- `off` - 時間ベースのプロファイルを無効にします。
- `on` - デフォルトのプロファイル間隔 (約 10 ミリ秒) で時間ベースのプロファイルを有効にします。
- `lo[w]` - 低分解能プロファイル間隔 (約 100 ミリ秒) で時間ベースのプロファイルを有効にします。
- `hi[gh]` - 高分解能プロファイル間隔 (約 1 ミリ秒) で時間ベースのプロファイルを有効にします。高分解能のプロファイルについては、54 ページの「時間ベースのプロファイルに関する制限事項」を参照してください。
- [+]*value* - 時間ベースのプロファイルを有効にし、プロファイル間隔を *value* に設定します。*value* のデフォルトの単位はミリ秒です。*value* は、整数または浮動小数点数として指定できます。オプションとして、数値の後ろに接尾辞 `m` を付けてミリ秒単位を選択するか、`u` を付けてマイクロ秒単位を選択することができます。プロファイル間隔は、時間の分解能の倍数である必要があります。時間の分解能値よりも大きな値であっても倍数でない場合は、端数が切り捨てられます。時間の分解能値よりも小さな値の場合は、警告メッセージが出力され、時間の分解能に設定されます。

  SPARC プラットフォームでは、ハードウェアカウンタのプロファイルと同様に、値の前に + 記号を付けると、時間ベースのデータ空間プロファイルが有効になります。

`collect` コマンドは、デフォルトで時間ベースのプロファイルデータを収集します。

-h *counter_definition_1*`...[,` *counter_definition_n*`]`

ハードウェアカウンタオーバーフローのプロファイルデータを収集します。カウンタ定義の数はプロセッサによって異なります。

このオプションは、Linux オペレーティングシステムを実行しているシステムでも、`perfctr` パッチをインストールすれば使用できるようになります。このパッチは、`http://user.it.uu.se/~mikpe/linux/perfctr/2.6/` からダウンロードできます。インストール方法の指示は、tar ファイルに含まれています。ユーザーレベルの `libperfctr.so` ライブラリは、`LD_LIBRARY_PATH` 環境変数の値を使用して検索されたあとで、32 ビットバージョンでは `/usr/local/lib`、`/usr/lib`、および `/lib` で、64 ビットバージョンでは `/usr/local/lib64`、`/usr/lib64`、および `/lib64` で検索されます。

使用可能なカウンタの一覧を表示するには、引数なしで `collect` コマンドを端末ウィンドウに入力します。カウンタの一覧については、28 ページの「ハードウェアカウンタのリスト」を参照してください。ほとんどのシステムでは、カウンタが一覧に記載されていない場合でも、16 進数または 10 進数の数値で指定できます。

カウンタ定義には、プロセッサがハードウェアカウンタの属性をサポートしているかどうかに応じて、次のいずれかの形式を使用できます。

[+]*counter_name*[/ *register_number*][,*interval*]

[+]*counter_name*[~ *attribute_1*=*value_1*]...[~*attribute_n* =*value_n*][/ *register_number*][,*interval*]

プロセッサ固有の *counter_name* には、次のいずれかを指定できます。

- カウンタ名の別名
- 生の名前
- 10 進数または 16 進数の数値

複数のカウンタを指定する場合、それらのカウンタは異なるレジスタを使用する必要があります。同じレジスタが指定された場合、`collect` コマンドはエラーメッセージを出力して終了します。

ハードウェアカウンタがメモリーアクセスに関連するイベントをカウントする場合、カウンタ名の前に + 記号を付けて、カウンタのオーバーフローを発生させた命令の実際のプログラムカウンタアドレス (PC) の検索をオンにすることができます。バックトラッキングは SPARC プロセッサ上で、`load`、`store`、`load-store` のいずれかのタイプのカウンタでのみ機能します。検索が成功すると、仮想 PC、物理 PC、および参照された有効アドレスがイベントデータパケットに格納されます。

一部のプロセッサでは、属性オプションをハードウェアカウンタと関連付けることができます。プロセッサが属性オプションをサポートしている場合は、`collect` コマンドを引数リストなしで実行すると、属性名を含むカウンタ定義が一覧表示されます。属性値は、10 進数または 16 進数形式で指定できます。

間隔 (オーバーフロー値) は、ハードウェアカウンタがオーバーフローしてオーバーフローイベントが記録されたときにカウントされたイベントまたはサイクルの数です。間隔は、次のいずれかに設定できます。

- `on` または NULL 文字列 - デフォルトのオーバーフロー値で、`collect` を引数なしで入力することによって判別できます。
- `hi[gh]` - 選択したカウンタの高分解能値で、デフォルトのオーバーフロー値の約 1/10 です。旧バージョンのソフトウェアとの互換を図るため、`h` の省略形もサポートされています。
- `lo[w]` - 選択したカウンタの低分解能値で、デフォルトのオーバーフロー値の約 10 倍です。
- *interval* - 特定のオーバーフロー値で、10 進数または 16 進数形式の正の整数です。

デフォルトでは、各カウンタに定義済みの通常のしきい値が使用されます。これらの値はカウンタの一覧に表示されます。55 ページの「ハードウェアカウンタオーバーフローのプロファイルに関する制限事項」も参照してください。

-p オプションを明示的に指定せずに -h オプションを使用すると、時間ベースのプロファイルが無効となります。ハードウェアカウンタデータと時間ベースデータの両方を収集するには、-h オプションと -p オプションの両方を指定する必要があります。

-s *option*

同期待ちトレースデータを収集します。*option* に使用できる値は次のとおりです。

- `all` - しきい値 0 で同期待ちトレースを有効にします。このオプションは、すべての同期イベントの記録を強制的に有効にします。
- `calibrate` - 同期待ちトレースを有効にし、実行時に測定を行うことによってしきい値を設定します。`on` と等価です。
- `off` - 同期待ちトレースを無効にします。
- `on` - 同期待ちトレースを有効にし、デフォルトのしきい値 (実行時の測定により値を決定) に設定します。`calibrate` と等価です。
- *value* - しきい値を *value* に設定します。この値は正の整数で、単位はマイクロ秒です。

Java プログラムでは同期待ちトレースデータは記録されず、指定するとエラーとして処理されます。

Solaris では、次の関数がトレースされます。

```
mutex_lock()
rw_rdlock()
rw_wrlock()
cond_wait()
cond_timedwait()
cond_reltimedwait()
thr_join()
sema_wait()
pthread_mutex_lock()
pthread_rwlock_rdlock()
pthread_rwlock_wrlock()
pthread_cond_wait()
pthread_cond_timedwait()
pthread_cond_reltimedwait_np()
pthread_join()
sem_wait()
```

Linux では、次の関数がトレースされます。

```
pthread_mutex_lock()
```

```
pthread_cond_wait()
pthread_cond_timedwait()
pthread_join()
sem_wait()
```

-H *option*

ヒープトレースデータを収集します。*option* に使用できる値は次のとおりです。

- on - ヒープの割り当て要求および割り当て解除要求のトレースを有効にします。
- off - ヒープトレースを無効にします。

デフォルトでは、ヒープのトレースは無効です。ヒープトレースは Java プログラムについてはサポートされず、指定するとエラーとして処理されます。

-M *option*

MPI 実験の収集を指定します。collect コマンドのタ–ゲットは mpirun コマンドである必要があります。また、mpirun コマンドのオプションは、-- オプションを使って mpirun コマンドによって実行されるターゲットプログラムと分けて指定されることが必要です。(mpirun コマンドでは常に -- オプションを使用することで、collect コマンドとそのオプションを mpirun コマンド行の先頭に追加し、実験を収集できます。)この実験には通常の名前が付けられ、親の実験と呼ばれます。ディレクトリには MPI プロセスのそれぞれについて、ランクにより命名されたサブ実験が含まれます。

option に使用できる値は次のとおりです。

- *MPI-version* - MPI 実験の収集を有効にします。指定された MPI バージョンが使用されます。MPI バージョンは、OMPT、CT、OPENMPI、MPICH2、MVAPICH2 のいずれかであることが必要です。Oracle Message Passing Toolkit は OMPT または CT を使用して指定できます。
- off - MPI 実験の収集を無効にします。

デフォルトでは、MPI 実験の収集は無効になっています。MPI 実験の収集が有効な場合、-m オプションのデフォルト設定は on に変更されます。

collect コマンドをオプションなしで入力するか、-M オプションで認識されないバージョンを指定すると、MPI のサポートされているバージョンが表示されます。

-m *option*

MPI トレースデータを収集します。*option* に使用できる値は次のとおりです。

- on - MPI トレース情報を有効にします。
- off - MPI トレース情報を無効にします。

MPI トレースは、デフォルトで無効です。ただし -M オプションが有効な場合は、デフォルトで有効になります。通常、MPI 実験は -M オプションで収集され、MPI トレースのユーザー制御は不要です。MPI 実験の収集を行うが、MPI トレースデータは収集しない場合、明示的なオプションの -M *MPI-version* -m off を使用します。

呼び出しがトレースされる MPI 関数とトレースデータをもとに計算されるメトリックスの詳細については、32 ページの「MPI トレースデータ」を参照してください。

-S *option*

標本パケットを定期的に記録します。*option* に使用できる値は次のとおりです。

- off - 定期的標本収集を無効にします。
- on - 定期的標本収集を有効にし、デフォルトの標本収集間隔 (1 秒) を設定します。
- *value* - 定期的標本収集を有効にし、標本収集間隔を *value* に設定します。間隔値は正の値、単位は秒とします。

デフォルトでは、1 秒間隔による定期的標本収集が有効になります。

-c *option*

カウントデータを記録します (Solaris システムのみ)。

注 – この機能を使用するには、Add-on Cool Tools for OpenSPARC に含まれているバイナリインタフェースツール (Binary Interface Tool、BIT) をインストールする必要があります。このツールは、http://cooltools.sunsource.net/ からダウンロードできます。BIT は、Solaris バイナリのパフォーマンスやテストスイートカバレージの測定用ツールです。

---

option には次のいずれかの値を指定できます。

- on - 関数と命令のカウントデータの収集を有効にします。カウントデータと、シミュレートされたカウントデータは、実行可能ファイル、および設置されており実行可能ファイルが静的にリンクされている任意の共有オブジェクトについて、それらの実行可能ファイルと共有オブジェクトが -xbinopt=prepare オプションでコンパイルされたものであれば、記録されます。静的にリンクされていても -xbinopt=prepare オプションでコンパイルされていないそのほかの共有オブジェクトは、データに含まれません。動的に開かれる共有オブジェクトは、シミュレートされたカウントデータに含まれません。

  関数、行などのカウントメトリックスに加えて、パフォーマンスアナライザの「命令頻度」タブ、または er_print ifreq コマンドにより、さまざまな命令の利用率の概要を確認することもできます。
- off - カウントデータの収集を無効にします。

- static - ターゲットの実行可能ファイル、および静的にリンクされている共有オブジェクト内で、すべての命令が 1 回だけ実行されたという前提で実験を生成します。-c on オプションと同様に、-c static オプションも、実行可能ファイルと共有オブジェクトが -xbinopt=prepare フラグを指定してコンパイルされている必要があります。

デフォルトでは、カウントデータの収集を無効にします。ほかの種類のデータについて、カウントデータを収集することはできません。

-I *directory*

bit 計測のディレクトリを指定します。このオプションは Solaris システムでのみ使用可能で、-c オプションも同時に指定されている場合のみ意味を持ちます。

-N *library_name*

bit() 計測から除外するライブラリを指定します。ここで指定したものは、ライブラリが実行可能ファイルにリンクされているか、dlopen() によって読み込まれるかにかかわらず、除外されます。このオプションは Solaris システムでのみ使用可能で、-c オプションも同時に指定されている場合のみ意味を持ちます。-N オプションは複数指定できます。

-r *option*

スレッドアナライザ用に、データ競合検出またはデッドロック検出のデータを収集します。次のいずれかの値を指定できます。

- race - データ競合検出のデータを収集します。
- deadlock - デッドロックと潜在的デッドロックのデータを収集します。
- all - データ競合検出とデッドロック検出のデータを収集します。
- off - スレッドアナライザのデータを無効にします。

collect -r コマンドとスレッドアナライザについての詳細は、『Oracle Solaris Studio 12.2: スレッドアナライザユーザーズガイド』および tha (1) のマニュアルページを参照してください。

実験制御オプション

これらのオプションは、実験データの収集方法を制御します。

-F *option*

派生プロセスのデータを記録するかどうかを制御します。*option* に使用できる値は次のとおりです。

- on - 関数 fork、exec、およびそのバリアントによって作成される派生プロセスについてのみ実験を記録します。

- all - すべての派生プロセスについて実験を記録します。
- off - 派生プロセスの実験を記録しません。
- =*regexp* - 指定された正規表現と名前または系統が一致する、派生プロセスの実験をすべて記録します。

デフォルトでは、-F on オプションが設定されるので、コレクタは、fork(2)、fork1(2)、fork(3F)、vfork(2)、および exec(2) 関数とそのバリアントの呼び出しによって作成されたプロセスを追跡します。vfork への呼び出しは、内部で fork1 への呼び出しに置換されます。

MPI 実験では、デフォルトで派生も追跡されます。

-F all オプションを指定すると、コレクタは、system(3C)、system(3F)、sh(3F)、posix_spawn(3p)、posix_spawnp(3p)、および popen(3C)、および同様の関数の呼び出しによって作成されたものを含むすべての派生プロセス、そして関係する派生プロセスを追跡します。

-F '=*regexp*' オプションを指定すると、コレクタはすべての派生プロセスを追跡します。派生名またはサブ実験名が指定の正規表現と一致する場合、コレクタはサブ実験を作成します。正規表現については、regexp(5) のマニュアルページを参照してください。

派生プロセスのデータを収集するとき、コレクタは、派生プロセスごとに新しい実験を親の実験内に 1 つ開きます。これらの新しい実験は、次のように、下線、文字、および数字を実験接尾辞に追加することで命名されます。

- 文字「f」は fork、「x」は exec、「c」はそのほかの派生プロセスをそれぞれ表します。
- 数字は、fork または exec (成功したかどうかに関係なく)、あるいはその他の呼び出しのインデックスです。

たとえば初期プロセスの実験名が test.1.er の場合、3 回目の fork の呼び出しで作成された子プロセスの実験は test.1.er/_f3.er となります。この子プロセスが新しいイメージを実行した場合、対応する実験名は test.1.er/_f3_x1.er となります。この子プロセスが popen 呼び出しを使用して別のプロセスを作成した場合、実験名は test.1.er/_f3_x1_c1.er となります。

アナライザと er_print ユーティリティーは、親実験が読み込まれると、自動的に派生プロセスの実験を読み込み、データ画面に派生を表示します。

表示するデータをコマンド行から選択するには、er_print か analyzer にパス名を明示的に指定します。指定するパスには、親の実験名と、親ディレクトリ内の派生実験名を含める必要があります。

たとえば、test.1.er 実験の 3 回目の fork のデータを表示する場合は、次のように指定します。

```
er_print test.1.er/_f3.er

analyzer test.1.er/_f3.er
```

もう一つの方法として、関心のある派生の実験の明示的な名前を入れた実験グループファイルを用意する方法もあります。

アナライザで派生プロセスを調べるには、親の実験を読み込んで、「表示」メニューから「データをフィルタ」を選択します。実験のリストは、親の実験のみが選択されて表示されます。これを選択解除し、対象とする派生実験を選択します。

注 – 派生プロセスが追跡されている間に親プロセスが終了した場合、まだ実行している派生のデータ収集は継続されます。それに従って親の実験ディレクトリは拡大を続けます。

また、スクリプトのデータを収集して、スクリプトの派生プロセスを追跡することもできます。詳細は、88 ページの「スクリプトからのデータの収集」を参照してください。

`-j` *option*

ターゲットプログラムが JVM の場合に Java プロファイルを有効にします。*option* に使用できる値は次のとおりです。

- `on` - Java HotSpot 仮想マシンによってコンパイルされたメソッドを認識し、Java 呼び出しスタックの記録を試みます。
- `off` - Java HotSpot 仮想マシンによってコンパイルされたメソッドの認識を試みません。
- *path* - 指定された *path* にインストールされている JVM についてのプロファイルデータを記録します。

`-j` オプションは、`.class` ファイルまたは `.jar` ファイルについてのデータを収集する場合は必要ありません。ただし、`java` 実行可能ファイルへのパスが `JDK_HOME` 環境変数または `JAVA_PATH` 環境変数に入っている必要があります。それから `collect` コマンド行でターゲットの *program* を `.class` ファイルまたは `.jar` ファイルとして指定します。拡張子は付けても付けなくてもかまいません。

`JDK_HOME` または `JAVA_PATH` 環境変数で `java` 実行可能ファイルのパスを定義できない場合や、Java HotSpot 仮想マシンによってコンパイルされたメソッドの認識を無効にしたい場合は、`-j` オプションを使用できます。このオプションを使用する場合、`collect` コマンド行で指定する *program* は Java 仮想マシンで、JDK 6、Update 18 またはそれ以降の必要があります。`collect` コマンドは、*program* が JVM で、かつ ELF 実行可能ファイルであることを検証し、そうでない場合は `collect` コマンドがエラーメッセージを出力します。

64 ビット JVM を使用してデータを収集する場合、32 ビット JVM 用の java コマンドに -d64 オプションを使用しないでください。これを使用すると、データは収集されません。その代わりに、collect コマンドの *program* 引数、あるいは JDK_HOME または JAVA_PATH 環境変数に、64 ビット JVM へのパスを指定する必要があります。

-J *java_argument*

プロファイルで使用するために JVM へ渡される追加引数を指定します。-J オプションを指定し、Java プロファイルを指定しない場合、エラーが生成され、実験は実行されません。*java_argument* に複数の引数が含まれる場合、全体を引用符でくくる必要があります。この引数は、空白またはタブで区切られた一連のトークンで構成される必要があります。各トークンは、別々の引数として JVM へ渡されます。JVM へのほとんどの引数は、「-」文字で始まる必要があります。

-l *signal*

signal という名前のシグナルがプロセスへ送信されたときに、標本パケットを記録します。

シグナルは、完全なシグナル名、先頭文字 SIG を省いたシグナル名、またはシグナル番号のいずれの形式でも指定できます。ただし、プログラムが使用するシグナル、または実行を終了するシグナルは指定しないでください。推奨するシグナルは SIGUSR1 および SIGUSR2 です。 SIGPROF は、時間プロファイリングが指定されている場合でも使用できます。シグナルは、kill コマンドを使用してプロセスに送信できます。

-l および -y の両方のオプションを使用する場合は、それぞれのオプションに異なるシグナルを使用する必要があります。

プログラムに独自のシグナルハンドラがあるときにこのオプションを使用する場合は、-l で指定するシグナルが、阻止されたり無視されたりすることなく、確実にコレクタのシグナルハンドラに渡されるようにする必要があります。

シグナルについては、signal(3HEAD) のマニュアルページを参照してください。

-t *duration*

データ収集の時間範囲を指定します。

duration は単独の数値として指定でき、オプションとして m または s 接尾辞を付け、実験終了までの時間が分単位か秒単位かを示すこともできます。 デフォルトでは、所要時間は秒です。所要時間はハイフンで区切られた 2 つの数で指定することもできます。これは、1 つ目の時間が経過するまでデータ収集を停止し、そして、データ収集を始める時間を示しています。2 つ目の時間が経過すると、データ収集が終了されます。2 つ目の時間がゼロの場合、初めてプログラムが停止したあと、そのプログラムの実行の終わりまで、データの収集が実行されます。実験が終了しても、ターゲットプロセスは最後まで実行できます。

-x

デバッガがプロセスに接続できるよう、exec システムコールの終了時にターゲットプロセスを停止したままにします。dbx をプロセスに接続した場合は、dbx コマンドの ignore PROF と ignore EMT を使用して、収集シグナルが確実に collect コマンドに渡されるようにします。

-y *signal*[**,** **r**]

signal という名前のシグナルで、データの記録を制御します。このシグナルがプロセスに送信されると、一時停止状態 (データは記録されない) と記録状態 (データは記録される) が切り替わります。ただし、このスイッチの状態に関係なく、標本ポイントは常に記録されます。

シグナルは、完全なシグナル名、先頭文字 SIG を省いたシグナル名、またはシグナル番号のいずれの形式でも指定できます。ただし、プログラムが使用するシグナル、または実行を終了するシグナルは指定しないでください。推奨するシグナルは SIGUSR1 および SIGUSR2 です。 SIGPROF は、時間プロファイリングが指定されている場合でも使用できます。シグナルは、kill コマンドを使用してプロセスに送信できます。

-l および -y の両方のオプションを使用する場合は、それぞれのオプションに異なるシグナルを使用する必要があります。

-y オプションに r 引数 (省略可能) を指定した場合、コレクタは記録状態で起動します。-y オプションが使用されていない場合、コレクタは記録状態で起動します。

プログラムに独自のシグナルハンドラがあるときにこのオプションを使用する場合は、-y で指定するシグナルが、阻止されたり、無視されたりすることなく、確実にコレクタのシグナルハンドラに渡されるようにする必要があります。

シグナルについては、signal(3HEAD) のマニュアルページを参照してください。

出力オプション

これらのオプションは、コレクタによって生成される実験を制御します。

-o *experiment_name*

記録する実験の名前として *experiment_name* を使用します。*experiment_name* 文字列は文字列「.er」で終わる必要があり、そうでない場合、collect ユーティリティーはエラーメッセージを出力して終了します。

-o オプションを指定しない場合、実験名を「*stem*.*n*.er」(*stem* は文字列、*n* は数値) の形式で指定します。-g オプションでグループ名を指定した場合、*stem* を「.erg」接尾辞なしのグループ名に設定します。グループ名を指定していない場合、*stem* を文字列「test」に設定します。

collect コマンドを、MPI ジョブの実行に使用されるコマンドの 1 つ、たとえば mpirun から起動し、-M *MPI-version* オプションおよび -o オプションを指定しない場合、そのプロセスの MPI ランクを定義するために使用された環境変数から、名前に使用されている *n* の値を使用します。それ以外の場合、現在使用されている最も大きい整数値に 1 を加えた値を *n* に設定します。

名前が「*stem.n.er*」の形式で指定されておらず、与えられた名前がすでに使用されている場合、エラーメッセージが表示され、実験は実行されません。名前が「*stem.n .er*」の形式で、与えられた名前がすでに使用されている場合、現在使用されている最も大きい数値より 1 大きい *n* の値に対応する名前で、実験が記録されます。名前が変更された場合、警告が表示されます。

-d *directory-name*

directory-name というディレクトリに実験を格納します。このオプションは個別の実験にのみ適用され、実験グループには適用されません。ディレクトリが存在しない場合、collect ユーティリティーはエラーメッセージを出力して終了します。-g オプションでグループが指定されている場合、グループファイルも *directory-name* へ書き込まれます。

データ収集をできるだけ軽くするには、レコードデータをローカルファイルに記録し、-d オプションを使用してデータ格納ディレクトリを指定するのが最適です。ただし、クラスタ上の MPI 実験では、すべてのプロセスから、記録されたすべてのデータが親の実験に書き込まれるため、親の実験がどのプロセスからでも同じパスに存在する必要があります。

待ち時間の長いファイルシステムに書き込まれる実験は特に問題が発生しやすく、標本データが収集される場合 (デフォルトの -S on オプション) には特に、非常に低速になることがあります。待ち時間の長い接続を経由して記録を行う必要がある場合には、標本データを無効にしてください。

-g *group-name*

実験を *group-name* という実験グループに含めます。*group-name* の末尾が .erg でない場合、collect ユーティリティーはエラーメッセージを出力して終了します。グループが存在する場合は、そのグループに実験が追加されます。*group-name* が絶対パスでない場合、-d でディレクトリを指定されていれば、実験グループがディレクトリ *directory-name* に、それ以外の場合は現在のディレクトリに格納されます。

-A *option*

ターゲットプロセスで使用されるロードオブジェクトを、記録済み実験に保管またはコピーするかどうかを管理します。option に使用できる値は次のとおりです。

- off – ロードオブジェクトを実験に保管しません。
- on – ロードオブジェクトを実験に保管します。

- copy – ロードオブジェクト (ターゲット、およびターゲットが使用するすべての共有オブジェクト) を実験にコピーして保管します。

実験データが記録されたマシンとは異なるマシンに実験データをコピーするか、異なるマシンから実験データを読み取る場合は、 `-A copy` を指定します。このオプションを使用しても、ソースファイルまたはオブジェクトファイル (`.o`) は実験にコピーされません。これらのファイルが、実験の検査に使用するマシンからアクセス可能で、変更されていないことを確認してください。

-L *size*

記録するプロファイルデータの量を *size*M バイトに制限します。この制限は、時間ベースのプロファイルデータ、ハードウェアカウンタオーバーフローのプロファイルデータ、および同期待ちのトレースデータの合計に適用されますが、標本ポイントには適用されません。この制限値は概数にすぎないので、この値を超えることは可能です。

制限に達すると、それ以上のプロファイルデータは記録されませんが、ターゲットプロセスが終了するまで実験はオープン状態となります。定期的な標本収集が有効である場合、標本ポイントの書き込みが継続されます。

約 2G バイトの制限を付けるには、たとえば、`-L 2000` を指定します。指定するサイズは、ゼロより大きい必要があります。

デフォルトでは、記録されるデータサイズに制限はありません。

-O *file*

`collect` 自体の全出力を指定された *file* に付加しますが、生成されたターゲットからの出力はリダイレクトしません。ファイルが `/dev/null` に設定されている場合は、エラーメッセージを含む `collect` の全出力が抑制されます。

その他のオプション

これらの `collect` コマンドオプションはさまざまな目的で使用されます。

-P *process_id*

`dbx` が指定された *process_id* のプロセスに接続し、データを収集してから、スクリプト上で `dbx` を起動するためのスクリプトを作成します。プロファイルデータのみを指定でき、トレースデータは指定できません。時間指定の実行 (`-t` オプション) はサポートされません。

-C コメント

実験の `notes` ファイルにコメントを追加します。最大 10 個の `-C` オプションを指定できます。notes ファイルの内容は、実験のヘッダーの先頭に付加されます。

-n

ターゲットを実行しませんが、ターゲットが実行されれば生成されたはずの実験の詳細を出力します。このオプションは「ドライラン」オプションです。

-R

パフォーマンスアナライザの Readme のテキストバージョンを端末ウィンドウに表示します。readme が見つからない場合は、警告が出力されます。これ以降に指定した引数は検査されず、これ以外の処理は行われません。

-V

collect コマンドの現在のバージョンを表示します。これ以降に指定した引数は検査されず、これ以外の処理は行われません。

-v

collect コマンドの現在のバージョンと、実行中の実験に関する詳細情報を表示します。

collect ユーティリティーによる動作中のプロセスからのデータの収集

Solaris OS の場合のみ、collect ユーティリティーで -P *pid* オプションを使用して、指定された PID のプロセスに接続し、そのプロセスのデータを収集できます。collect コマンドのそのほかのオプションは dbx 用のスクリプトに変換され、そのスクリプトを起動してデータが収集されます。時間ベースのプロファイルデータ (-p オプション) とハードウェアカウンタオーバーフローのプロファイルデータ (-h オプション) だけを収集できます。トレースデータはサポートされていません。

-p オプションを明示的に指定せずに -h オプションを使用すると、時間ベースのプロファイルが無効となります。ハードウェアカウンタデータと時間ベースデータの両方を収集するには、-h オプションと -p オプションの両方を指定する必要があります。

▼ collect ユーティリティーを使用して動作中のプロセスからデータを収集する

1 プログラムのプロセス ID (PID) の判定

コマンド行からプログラムを起動していて、バックグラウンドで実行している場合は、シェルによってその PID が標準出力に出力されます。それ以外の場合、次のコマンドを入力してプログラムの PID を判定できます。

```
% ps -ef | grep program-name
```

2 **collect** コマンドを使用してプロセスのデータの収集を有効にし、オプションのパラメータを適宜設定します。

```
% collect -P pid collect-options
```

コレクタのオプションについては、62 ページの「データ収集オプション」を参照してください。時間ベースのプロファイルについては、63 ページの「-p *option*」を参照してください。ハードウェアカウンタオーバーフローのプロファイルについては、-h optionを参照してください。

dbx collector サブコマンドによるデータの収集

この節では、dbx からコレクタを実行する方法、また dbx 内の collector コマンドで使用できる各サブコマンドについて説明します。

▼ dbx からコレクタを実行する

1 次のコマンドを使用して、**dbx** にプログラムを読み込みます。

```
% dbx program
```

2 **collector** コマンドを使用してデータの収集を有効にし、データの種類を選択し、オプションのパラメータを適宜設定します。

```
(dbx) collector subcommand
```

利用可能な collector サブコマンドの一覧を表示するには、次のコマンドを使用します。

```
(dbx) help collector
```

サブコマンドごとに collector コマンドを 1 つ使用する必要があります。

3 使用する **dbx** のオプションを設定し、プログラムを実行します。

指定したサブコマンドに誤りがある場合は、警告メッセージが出力され、サブコマンドは無視されます。このあとに、collector の全サブコマンドをまとめます。

データ収集のサブコマンド

次のサブコマンドを、dbx 内で collector コマンドとともに使用して、コレクタにより収集されるデータの種類を制御できます。実験がアクティブな場合は、警告メッセージが出力され、サブコマンドは無視されます。

profile *option*

時間ベースのプロファイルデータの収集を制御します。*option* に使用できる値は次のとおりです。

- on - デフォルトのプロファイル間隔 (10 ミリ秒) で時間ベースのプロファイルを有効にします。
- off - 時間ベースのプロファイルを無効にします。
- timer *interval* - プロファイル間隔を設定します。*interval* に使用できる値は次のとおりです。
  - on - デフォルトのプロファイル間隔 (約 10 ミリ秒) を使用します。
  - lo[w] - 低分解能のプロファイル間隔 (約 100 ミリ秒) を使用します。
  - hi[gh] - 高分解能のプロファイル間隔 (約 1 ミリ秒) を使用します。高分解能のプロファイルについては、54 ページの「時間ベースのプロファイルに関する制限事項」を参照してください。
  - *value* - プロファイル間隔を *value* に設定します。*value* のデフォルトの単位はミリ秒です。*value* は、整数または浮動小数点数として指定できます。オプションとして、数値の後ろに接尾辞 m を付けてミリ秒単位を選択するか、u を付けてマイクロ秒単位を選択することができます。プロファイル間隔は、時間の分解能の倍数である必要があります。時間の分解能値よりも大きな値であっても倍数でない場合は、端数が切り捨てられます。時間の分解能値よりも小さな値の場合は、時間の分解能に設定されます。どちらの場合にも、警告メッセージが表示されます。

    デフォルトの設定は約 10 ミリ秒です。

    デフォルトでは、hwprofile サブコマンドを使用してハードウェアカウンタオーバーフローのプロファイルデータ収集が有効になっていないかぎり、コレクタは時間ベースのプロファイルデータを収集します。

hwprofile *option*

ハードウェアカウンタオーバーフローのプロファイルデータの収集を制御します。ハードウェアカウンタオーバーフローのプロファイルをサポートしていないシステム上でこの機能を有効にしようとすると、dbx から警告メッセージが返され、コマンドは無視されます。*option* に使用できる値は次のとおりです。

- on - ハードウェアカウンタオーバーフローのプロファイルを有効にします。デフォルトでは、通常のオーバーフロー値で cycles カウンタのデータが収集されます。

- `off` - ハードウェアカウンタオーバーフローのプロファイルを無効にします。
- `list` - 使用可能なカウンタの一覧を返します。一覧の説明については、28 ページの「ハードウェアカウンタのリスト」を参照してください。ハードウェアカウンタオーバーフローのプロファイル機能がシステムでサポートされていない場合は、`dbx` から警告メッセージが返されます。
- `counter` *counter_definition*... [, *counter_definition*] – カウンタの定義は次の形式です。

  [+]*counter_name* [~ *attribute_1*=*value_1*]...[~ *attribute_n* =*value_n*] [/ *register_number*] [,*interval*]

  ハードウェアカウンタの *name* を選択し、そのオーバーフロー値を *interval* に設定します。オプションとして、追加のハードウェアカウンタ名を選択し、それらのオーバーフロー値を指定された間隔に設定します。オーバーフロー値は次のいずれかです。

  - `on` または NULL 文字列 - デフォルトのオーバーフロー値で、`collect` を引数なしで入力することによって判別できます。
  - `hi[gh]` - 選択したカウンタの高分解能値で、デフォルトのオーバーフロー値の約 1/10 です。旧バージョンのソフトウェアとの互換を図るため、`h` の省略形もサポートされています。
  - `lo[w]` - 選択したカウンタの低分解能値で、デフォルトのオーバーフロー値の約 10 倍です。
  - *interval* - 特定のオーバーフロー値で、10 進数または 16 進数形式の正の整数です。

    複数のカウンタを指定する場合、それらのカウンタは異なるレジスタを使用する必要があります。使用するレジスタが同じである場合は警告メッセージが出力され、コマンドは無視されます。

    ハードウェアカウンタがメモリーアクセスに関連するイベントをカウントする場合、カウンタ名の前に + 記号を付けて、カウンタのオーバーフローを発生させた命令の実際の PC の検索をオンにすることができます。検索が成功すると、PC と参照された有効アドレスがイベントデータパケットに格納されます。

    デフォルトでは、コレクタは、ハードウェアカウンタのオーバーフロープロファイルデータを収集しません。ハードウェアカウンタオーバーフローのプロファイルが有効になっていて `profile` コマンドが指定されていない場合、時間ベースのプロファイルは無効となります。

    55 ページの「ハードウェアカウンタオーバーフローのプロファイルに関する制限事項」も参照してください。

`synctrace` *option*

同期待ちトレースデータの収集を制御します。*option* に使用できる値は次のとおりです。

- on - デフォルトのしきい値で同期待ちトレースを有効にします。
- off - 同期待ちトレースを無効にします。
- threshold *value* - 同期遅延の最小しきい値を設定します。*value* に使用できる値は次のとおりです。
  - all - しきい値 0 を使用します。このオプションは、すべての同期イベントの記録を強制的に有効にします。
  - calibrate - 実行時に測定を行ってしきい値を設定します。on と等価です。
  - off - 同期待ちのトレースを無効にします。
  - on - デフォルトのしきい値 (実行時の測定により値を決定) を設定します。calibrate と等価です。
  - *number* - しきい値を *number* に設定します。この数値は正の整数で、単位はマイクロ秒です。value が 0 の場合、すべてのイベントをトレースします。

    デフォルトでは、コレクタは同期待ちのトレースデータを収集しません。

heaptrace *option*

ヒープトレースデータの収集を制御します。*option* に使用できる値は次のとおりです。

- on - ヒープトレースを有効にします。
- off - ヒープトレースを無効にします。

デフォルトでは、コレクタはヒープのトレースデータを収集しません。

tha *option*

スレッドアナライザ用に、データ競合検出またはデッドロック検出のデータを収集します。次のいずれかの値を指定できます。

- off – スレッドアナライザのデータ収集を無効にします。
- all – すべてのスレッドアナライザデータを収集します。
- race - データ競合検出データを収集します。
- deadlock - デッドロックと潜在的デッドロックのデータを収集します。

スレッドアナライザについての詳細は、『Oracle Solaris Studio 12.2: スレッドアナライザユーザーズガイド』および tha.1 のマニュアルページを参照してください。

sample *option*

標本収集モードを制御します。*option* に使用できる値は次のとおりです。

- periodic - 定期的な標本収集を有効にします。
- manual - 定期的な標本収集を無効にします。手動の標本収集は有効のままです。
- period *value* - 標本収集の間隔を *value* (秒単位) に設定します。

デフォルトでは、標本収集間隔 *value* が 1 秒での定期的な標本収集が有効となります。

dbxsample{on|off}

dbx がターゲットプロセスを停止したときに、標本を記録するかどうかを制御します。キーワードの意味は、次のとおりです。

- on - dbx がターゲットプロセスを停止するたびに標本が記録されます。
- off - dbx がターゲットプロセスを停止したときに標本は記録されません。

デフォルトでは、dbx がターゲットプロセスを停止したとき、標本が記録されます。

実験制御のサブコマンド

次のサブコマンドを、dbx 内で collector コマンドとともに使用して、コレクタによる実験データの収集を制御できます。実験がアクティブな場合は、警告メッセージが出力され、サブコマンドは無視されます。

disable

データ収集を無効にします。プロセスが実行中でデータが収集されている場合、その実験が終了し、データ収集は無効になります。プロセスが動作中でデータ収集が無効になっている場合、警告が出され、このサブコマンドは無視されます。プロセスが動作していない場合は、以降の実行のデータ収集が無効になります。

enable

データ収集を有効にします。プロセスが動作していてデータ収集が無効であった場合、データ収集が有効になって新しい実験が開始されます。プロセスが動作中でデータ収集が有効になっている場合、警告が出され、このサブコマンドは無視されます。プロセスが動作していない場合は、以降の実行のデータ収集が有効になります。

プロセスの動作中、データ収集は何回でも有効にしたり、無効にしたりできます。データ収集を有効にするたびに、新しい実験が作成されます。

pause

実験を開いたまま、データの収集を一時停止します。コレクタが一時停止している間、標本ポイントは記録されません。標本は一時停止の前に生成され、再開直後に別の標本が生成されます。データの収集がすでに一時停止されている場合、このサブコマンドは無視されます。

resume

pause が実行されたあとに、データ収集を再開します。データ収集中は、このサブコマンドは無視されます。

sample record *name*

name のラベルが付いた標本パケットを記録します。ラベルは、パフォーマンスアナライザの「イベント」タブで表示されます。

出力のサブコマンド

次のサブコマンドを、dbx 内で collector コマンドとともに使用して、実験の格納オプションを定義できます。実験がアクティブな場合は、警告メッセージが出力され、サブコマンドは無視されます。

archive *mode*

実験を保管するためのモードを設定します。 *mode* に使用できる値は次のとおりです。

- on - ロードオブジェクトを通常の方法で保管します。
- off - ロードオブジェクトを保管しません。
- copy - 通常の保管に加えて、ロードオブジェクトを実験にコピーします。

異なるマシンに実験を移動するか、別のマシンから実験を読み取る場合は、ロードオブジェクトのコピーを有効にする必要があります。実験がアクティブな場合、警告が出され、このコマンドは無視されます。このコマンドを使用しても、ソースファイルまたはオブジェクトファイルは実験にコピーされません。

limit *value*

記録するプロファイルデータの量を *value*M バイトに制限します。この制限は、時間ベースのプロファイルデータ、ハードウェアカウンタオーバーフローのプロファイルデータ、および同期待ちのトレースデータの合計に適用されますが、標本ポイントには適用されません。この制限値は概数にすぎないので、この値を超えることは可能です。

制限に達すると、それ以上のプロファイルデータは記録されませんが、実験はオープンのままで標本ポイントの記録は継続します。

デフォルトでは、記録されるデータサイズに制限はありません。

store *option*

実験の格納先を指定します。実験がアクティブな場合、警告が出力され、このコマンドは無視されます。*option* に使用できる値は次のとおりです。

- `directory` *directory-name* - 実験と実験グループの格納先のディレクトリを設定します。指定したディレクトリが存在しない場合、警告が出され、このサブコマンドは無視されます。
- `experiment` *experiment-name* - 実験の名前を設定します。指定した実験名の末尾が `.er` でない場合、警告が出され、このサブコマンドは無視されます。実験名とコレクタにおける実験名の取り扱いについての詳細は、58 ページの「収集データの格納場所」を参照してください。
- `group` *group-name* - 実験グループの名前を設定します。指定したグループ名の末尾が `.erg` でない場合、警告が出され、このサブコマンドは無視されます。グループがすでに存在する場合は、実験がグループに追加されます。ディレクトリ名が `store directory` サブコマンドで設定され、グループ名が絶対パスでない場合、グループ名の前にディレクトリ名が付きます。

情報のサブコマンド

次のサブコマンドを、`dbx` 内で `collector` コマンドとともに使用して、コレクタ設定と実験状態についてのレポートを取得できます。

`show`

すべてのコレクタ制御の、現在の設定を表示します。

`status`

開かれている実験の状態を報告します。

Solaris プラットフォームで dbx を使用して実行中のプロセスからデータを収集する方法

Solaris プラットフォームでは、コレクタを使用して、実行中のプロセスからデータを収集できます。プロセスがすでに `dbx` の制御下にある場合は、プロセスを一時停止し、これまでに説明した方法を使用してデータ収集を有効にすることができます。実行中のプロセスからデータ収集を開始する操作は、Linux プラットフォームではサポートされていません。

プロセスが `dbx` の制御下にない場合は、`collect –P` *pid* コマンドを使用して、実行中のプロセスからデータを収集できます。詳細は、75 ページの「`collect` ユーティリティーによる動作中のプロセスからのデータの収集」を参照してください。プロセスに `dbx` を接続し、パフォーマンスデータを収集し、プロセスから切り離して、プロセスの実行を継続することもできます。選択した派生プロセスのパフォーマンスデータを収集するには、各プロセスに `dbx` を接続する必要があります。

▼ dbx の制御下にない実行中のプロセスからデータを収集する方法

1. プログラムのプロセス ID (PID) の判定

   コマンド行からプログラムを起動していて、バックグラウンドで実行している場合は、シェルによってその PID が標準出力に出力されます。それ以外の場合、次のコマンドを入力してプログラムの PID を判定できます。

   ```
   % ps -ef | grep program-name
   ```

2. プロセスに接続します。

   dbx から、次のように入力します。

   ```
   (dbx) attach program-name pid
   ```

   dbx をまだ実行していない場合は、次のように入力します。

   ```
   % dbx program-name pid
   ```

   実行中のプロセスに接続すると、そのプロセスが一時停止します。

   プロセスへの接続についての詳細は、マニュアル『Oracle Solaris Studio 12.2: dbx コマンドによるデバッグ』を参照してください。

3. データの収集を開始します。

   dbx から、`collector` コマンドを使用してデータ収集パラメータを設定し、`cont` コマンドを使用してプロセスを再開します。

4. プロセスから切り離します。

   データの収集を完了したら、プログラムを一時停止し、dbx からプロセスを切り離します。

   dbx から、次のように入力します。

   ```
   (dbx) detach
   ```

動作中のプロセスからのトレースデータの収集

いずれかの種類のトレースデータを収集するには、プログラムを実行する前にコレクタライブラリ `libcollector.so` をプリロードする必要があります。ヒープトレースデータまたは同期待ちトレースデータを収集するには、`er_heap.so` および `er_sync.so` もそれぞれプリロードする必要があります。これらのライブラリは、データ収集を実行するための実際の関数へのラッパーを提供します。また、コレクタは、ほかのシステムライブラリの呼び出しにもラッパー関数を追加し、それによって完全なパフォーマンスデータを確保できます。ライブラリをプリロードし

ないと、これらのラッパー関数を挿入できません。コレクタがシステムライブラリ関数に割り込む方法の詳細については、47 ページの「システムライブラリの使用」を参照してください。

libcollector.so をプリロードするには、次の表に示すように、環境変数を使用してライブラリ名とライブラリへのパスを設定する必要があります。LD_PRELOAD 環境変数を使用して、ライブラリ名を設定します。ライブラリへのパスを設定するには、LD_LIBRARY_PATH、LD_LIBRARY_PATH_32、または LD_LIBRARY_PATH_64 環境変数を使用します。LD_LIBRARY_PATH は、_32 および _64 バリアントが定義されていない場合に使用されます。これらの環境変数をすでに定義している場合は、新しい値を追加してください。

表 3–2 libcollector.so、er_sync.so、および er_heap.so をプリロードするための環境変数設定

環境変数	値
LD_PRELOAD	libcollector.so
LD_PRELOAD	er_heap.so
LD_PRELOAD	er_sync.so
LD_LIBRARY_PATH	/opt/solstudio12.2/prod/lib/dbxruntime
LD_LIBRARY_PATH_32	/opt/solstudio12.2/prod/lib/dbxruntime
LD_LIBRARY_PATH_64	/opt/solstudio12.2/prod/lib/v9/dbxruntime
LD_LIBRARY_PATH_64	/opt/solstudio12.2/prod/lib/amd64/dbxruntime

Oracle Solaris Studio ソフトウェアが /opt/solstudio12.2 にインストールされていない場合は、システム管理者に正しいパスを確認してください。LD_PRELOADにフルパスを設定することもできますが、そのようにすると、SPARC V9 の 64 ビットアーキテクチャーを使用するときに問題が発生する可能性があります。

注 – 実行が終了したら、LD_PRELOAD と LD_LIBRARY_PATH の設定を削除し、同じシェルから起動されるほかのプログラムが設定の影響を受けないようにしてください。

MPI プログラムからのデータの収集

コレクタは、Message Passing Interface (MPI) を使用するマルチプロセスプログラムからパフォーマンスデータを収集できます。

コレクタは、Oracle Message Passing Toolkit 8 (以前の Sun HPC ClusterTools 8) とその更新をサポートしています。コレクタは、その他のバージョンの MPI も認識できます。collect を引数なしで実行すると、有効な MPI バージョンが一覧表示されます。

Oracle Message Passing Toolkit MPI ソフトウェアは http://www.oracle.com/us/products/tools/message-passing-toolkit-070499.html から入手できます。

MPI と MPI 規格については、MPI の Web サイト http://www.mcs.anl.gov/mpi/ を参照してください。Open MPI については、Web サイト http://www.open-mpi.org/ を参照してください。

MPI ジョブからデータを収集するには、collect コマンドを使用する必要があります。dbx collector サブコマンドは、MPI データ収集を開始するために使用できません。詳細は、85 ページの「MPI 用の collect コマンドの実行」を参照してください。

MPI 用の collect コマンドの実行

collect コマンドを使用して、MPI アプリケーションのトレースとプロファイルを実行できます。

データを収集するには、次の構文を使用します。

collect [*collect-arguments*] **mpirun** [*mpirun-arguments*] **--** *program-name* [*program-arguments*]

たとえば、次のコマンドは 16 の MPI プロセスそれぞれについて MPI トレースおよびプロファイルを実行し、データを単一の MPI 実験に格納します。

```
collect -M OMPT mpirun -np 16 -- a.out 3 5
```

-M OMPT オプションは、MPI プロファイルが実行されること、および Oracle Message Passing Toolkit が MPI バージョンであることを示します。

最初の収集プロセスは mpirun コマンドの形式を変更し、各 MPI プロセスについて適切な引数で collect コマンドを実行することを指定します。

program_name の直前にある -- 引数は、MPI プロファイルに必要となります。-- 引数を含めないと、collect コマンドによりエラーメッセージが表示され、実験は収集されません。

注 – mpirun コマンドを使用して MPI プロセス上で明示的に collect コマンドを生成する方法は、MPI トレースデータの収集ではサポートされなくなりました。ほかの種類のデータを収集する場合は、この方法を使用できます。

MPI 実験の格納

マルチプロセス環境は複雑になることがあるので、MPI プログラムからパフォーマンスデータを収集するときは、MPI 実験の格納に関するいくつかの問題に留意する

必要があります。これら問題はデータ収集とデータ保存の効率、実験の命名に関わります。MPI実験を含む命名実験については、58 ページの「収集データの格納場所」を参照。

パフォーマンスデータを収集する各 MPI プロセスは、独自のサブ実験を作成します。MPI プロセスは、実験を作成している間、実験ディレクトリをロックします。このため、ほかのすべての MPI プロセスは、ロックが解放されるまで待ってからそのディレクトリを使用する必要があります。実験は、すべての MPI プロセスからアクセス可能なファイルシステム上に格納します。

実験名を指定しなかった場合、デフォルトの実験名が使用されます。コレクタは、各 MPI ランクについて 1 つのサブ実験を実験の中に作成します。コレクタは MPI ランクを使用して、`M_rm .er` の形式でサブ実験名を構築します (*m* は MPI ランク)。

実験の完了後に、実験を別の場所へ移動する予定がある場合、`collect` コマンドで `-A copy` オプションを指定します。実験のコピーや移動には、UNIX® の `cp` または `mv` コマンドを使用しないで、第 8 章「実験の操作」で説明されている `er_cp` または `er_mv` コマンドを使用してください。

MPI トレースでは、各ノード上に一時ファイル `/tmp/a.*.z` が作成されます。これらのファイルは、`MPI_finalize()` 関数呼び出しによって削除されます。ファイルシステムに、実験のため十分な空き容量があることを確認します。実行時間の長い MPI アプリケーションのデータを収集する前に、短時間のテストを実行してファイルサイズを確認します。必要な空き容量を推定する方法について、60 ページの「必要なディスク容量の概算」も参照してください。

MPI プロファイルは、オープンソースの VampirTrace 5.5.3 リリースがベースになっています。サポートされているいくつかの VampirTrace 環境変数に加えて、`VT_STACKS` を新たに認識します。この環境変数は、呼び出しスタックをデータに記録するかどうかを制御します。これらの変数の意味については、VampirTrace 5.5.3 のドキュメントを参照してください。

環境変数 `VT_BUFFER_SIZE` のデフォルト値は、MPI API トレースコレクタの内部バッファーを 64 M バイトに制限します。特定の MPI プロセスについてこの制限に到達した後、`VT_MAX_FLUSHES` 制限にまだ到達していなければ、バッファーはディスクにフラッシュされます。デフォルトでは、`VT_MAX_FLUSHES` は 0 です。この設定を行うと、MPI API トレースコレクタは、バッファーがいっぱいになるたびにバッファーをディスクにフラッシュします。`VT_MAX_FLUSHES` を正数に設定すると、フラッシュ回数が制限されます。バッファーがいっぱいになり、フラッシュできない場合、そのプロセスについては、それ以降イベントがトレースファイルに書き込まれなくなります。結果として実験が不完全なものとなり、場合によっては実験が読み取り不能になることがあります。

バッファーのサイズを変更するには、`VT_BUFFER_SIZE` 環境変数を使用します。この変数の最適値は、トレース対象のアプリケーションによって異なります。小さい値を設定すると、アプリケーションで利用できるメモリーが増えますが、MPI API ト

レースコレクタによってバッファーのフラッシュが頻繁に引き起こされます。このようなバッファーのフラッシュによって、アプリケーションの動作が大幅に変化する可能性があります。一方、2G バイトなどの大きい値を設定すると、MPI API トレースコレクタによるバッファーフラッシュは最小限に抑えられますが、アプリケーションで利用できるメモリーが減少します。バッファーとアプリケーションデータを保持するための十分なメモリーが利用できない場合、この設定によりアプリケーションの一部がディスクにスワップされ、アプリケーションの動作が大きく変化する可能性があります。

もう 1 つの重要な変数は `VT_VERBOSE` で、各種のエラーメッセージおよび状態メッセージを有効にします。問題が発生した場合、この変数を 2 以上に設定してください。

通常、MPI トレース出力データは `mpirun` ターゲットの終了時に後処理されます。処理されたデータファイルは実験に書き込まれ、後処理時刻情報が実験ヘッダーに書き込まれます。MPI トレースが `-m off` により明示的に無効になっている場合、MPI 後処理は行われません。後処理に失敗した場合、エラーが報告され、MPI タブと MPI トレースメトリックスは使用できなくなります。

`mpirun` ターゲットが実際に MPI を起動しない場合、実験は記録されますが、MPI トレースデータは生成されません。実験により、MPI 後処理エラーが報告され、MPI タブと MPI トレースメトリックスは使用できなくなります。

環境変数 `VT_UNIFY` が `0` に設定された場合、`collect` により後処理ルーチンは実行されません。後処理ルーチンは、実験で `er_print` または `analyzer` が初めて起動したときに実行されます。

注 – コンピュータまたはノード間で実験のコピーあるいは移動を行うと、ソースファイル、または同じタイムスタンプを持つコピーにアクセスできないかぎり、注釈付きソースコードまたは注釈付き逆アセンブリコードのソース行を表示できません。注釈付きソースを表示するため、元のソースファイルへのシンボリックリンクを現在のディレクトリに置くことができます。また、「データ表示方法の設定」ダイアログボックスの設定を使用することもできます。「検索パス」タブ (114 ページの「「検索パス」タブ」を参照) では、ソースファイルの検索に使用するディレクトリのリストを管理できます。「パスマップ」タブ (114 ページの「「パスマップ」タブ」を参照) では、ファイルパスの先頭部分を、ある場所から別の場所へマッピングできます。

スクリプトからのデータの収集

デフォルトでは、collect は ELF 実行可能ファイルをターゲットとする必要があり、このことを確認するためにターゲットをチェックします。ただし、このチェック機能を無効にして、ターゲットとして指定するスクリプト上で collect を実行することができます。

注 – スクリプトプロファイリングは、試験段階の機能です。この実装は、以降のリリースで変更される可能性があります。

スクリプトをプロファイルするには、まず、環境変数 SP_COLLECTOR_SKIP_CHECKEXEC を設定して、ELF 実行可能ファイルのチェックを無効にします。

デフォルトでは、データは、スクリプトを実行するために起動したプログラムと、すべての派生プロセスで収集されます。特定のプロセスのみデータを収集するには、-F フラグを使用して、追跡する実行機能ファイルの名前を指定します。たとえば、スクリプト foo.sh をプロファイルするが、実行可能ファイル bar から主にデータを収集する場合は、次のコマンドを使用します。

csh の場合:

```
% setenv SP_COLLECTOR_SKIP_CHECKEXEC
% collect -F =bar foo.sh
```

sh の場合:

```
$ export SP_COLLECTOR_SKIP_CHECKEXEC
$ collect -F =bar foo.sh
```

スクリプトを実行するために起動した初期プロセスと、そのスクリプトから派生したすべての bar プロセスからデータを収集し、その他のプロセスからはデータを収集しません。

collect と ppgsz を組み合わせた使用法

ppgsz コマンド上で collect を実行し、-F on または -F all フラグを使用すると、collect と ppgsz(1) を組み合わせて使用できます。親の実験は ppgsz 実行可能ファイルにあり、注目対象外です。パスに 32 ビットバージョンの ppgsz が存在し、実験が 64 ビットプロセスをサポートするシステムで実行されている場合、最初に 64 ビットバージョンを exec して _x1.er を作成します。この実行可能ファイルを fork し、_x1_f1.er を作成します。

子プロセスは、パスの最初のディレクトリに存在する名前付きターゲットの exec を試み、exec の試みが成功するまで、順に次のディレクトリについて同様な操作を行います。たとえば、3 番目の試みが成功した場合、最初の 2 つの派生実験には

_x1_f1_x1.er および _x1_f1_x2.er という名前が付けられますが、これらは両方とも完全に空白です。ターゲット上の実験は、成功した 3 回目の exec によるもので、_x1_f1_x3.er という名前が付けられ、親の実験の下に格納されます。この実験は、test.1.er/_x1_f1_x3.er に対してアナライザまたは er_print ユーティリティーを起動することで直接処理可能です。

64 ビットの ppgsz が初期プロセスの場合、または 32 ビットの ppgsz が 32 ビットカーネル上で起動された場合、パスのプロパティーが前述の例と同じであると仮定して、実際のターゲットを exec する fork の子データは _f1.er にあり、実際のターゲットの実験は _f1_x3.er に存在します。

◆ ◆ ◆ 第 4 章

パフォーマンスアナライザツール

パフォーマンスアナライザは、コレクタが収集するパフォーマンスデータを解析するグラフィカルデータ解析ツールです。コレクタは、パフォーマンスアナライザのメニューオプションから開始することも、`dbx` 内で `collect` コマンドまたは `collector` コマンドを使用して開始することもできます。第 3 章「パフォーマンスデータの収集」で説明されているように、プロセスの実行中に、コレクタはパフォーマンス情報を収集して実験を作成します。パフォーマンスアナライザはこれらの実験を読み取り、そのデータを解析して表や図に表示します。また、コマンド行ツールである `er_print` ユーティリティーでは、ASCII テキスト形式で実験データを表示できます。詳細は、第 5 章「`er_print` コマンド行パフォーマンス解析ツール」を参照してください。

この章では、次の内容について説明します。

パフォーマンスアナライザの起動

パフォーマンスアナライザを起動するには、コマンド行で次のように入力します。

```
% analyzer [control-options] [experiment-list]
```

experiment-list 引数は、実験の名前または実験グループの名前、あるいはその両方を空白文字で区切ったリストです。実験リストを指定しない場合は、アナライザが起動すると、「実験を開く」ダイアログボックスが自動的に開くので、ここで実験を選択して開くことができます。

複数の実験または実験グループをコマンドで指定できます。内部に派生実験を持つ実験を指定すると、すべての派生実験が自動的に読み込まれます。初期プロセスで取得されたデータとすべての派生データが集計されます。個々の派生実験を読み込むには、明示的に各実験を指定するか、実験グループを作成する必要があります。また、`.er.rc` ファイルに `en_desc` 指令を記述することもできます (151 ページの「`en_desc { on | off | =`*regexp*` }`」を参照)。

実験グループを作成するには、`collect` ユーティリティーの `-g` 引数を使用できます。実験グループを手動で作成するには、最初の行が次のようなプレーンテキストファイルを作成します。

```
#analyzer experiment group
```

このあとの行に実験の名前を追加します。ファイルの拡張子は、`erg` である必要があります。

また、「アナライザ」ウィンドウの「ファイル」メニューを使って、実験や実験グループを追加することもできます。ファイル選択用ダイアログではディレクトリとして実験を開くことはできないため、派生プロセスについて記録された実験を開くには、「実験を開く」ダイアログボックスまたは「実験を追加」ダイアログボックスにファイル名を入力する必要があります。

アナライザで複数の実験が表示される場合、デフォルトでは、すべての実験から取得されたデータが集計されます。データはまとめられ、1 つの実験から取得されたものであるかのように表示されます。ただし、データを集計するのではなく、実験を比較することもできます。120 ページの「実験の比較」を参照してください。

「実験を開く」ダイアログボックスまたは「実験を追加」ダイアログボックスのいずれかで名前をシングルクリックすることで、読み込む実験や実験グループをプレビューできます。

また、次のようにコマンド行から、実験を記録するためにパフォーマンスアナライザを起動することもできます。

```
% analyzer [Java-options] [control-options] target [target-arguments]
```

アナライザによって「収集」ウィンドウが開かれ、指定したターゲットとその引数、および実験を収集するための設定が表示されます。詳細は、118 ページの「アナライザからの実験の記録」を参照してください。

アナライザのコマンドオプション

アナライザの動作を制御するこれらのオプションは、3 つのグループに分かれます。

- Java オプション
- 制御オプション
- 情報オプション

Java オプション

これらのオプションは、アナライザを実行する JVM の設定を指定します。

-j | --jdkhome *jvm-path*

アナライザを実行するための JVM ソフトウェアへのパスを指定します。-j オプションを指定しなかった場合、JVM へのパスを示す環境変数、JDK_HOME と JAVA_PATH をこの順序で調べることによってデフォルトのパスが最初に取得されます。どちらの環境変数も設定されていない場合は、現在の PATH にある JVM が使用されます。デフォルトのパスをすべて無効にするには、-j オプションを使用します。

-J *jvm-options*

JVM オプションを指定します。複数のオプションを指定できます。次に例を示します。

- 64 ビットのアナライザを実行するには、次のように入力します。

  ```
  analyzer -J-d64
  ```
- 最大 2G バイトの JVM メモリーを使用してアナライザを実行するには、次のように入力します。

  ```
  analyzer -J-Xmx2G
  ```
- 最大 8G バイトの JVM メモリーを使用して 64 ビットのアナライザを実行するには、次のように入力します。

  ```
  analyzer -J-d64 -J-Xmx8G
  ```

制御オプション

これらのオプションは、GUI のフォントサイズを制御し、アナライザを起動する前に、バージョン情報とランタイム情報を表示します。

-f | --fontsize *size*

アナライザ GUI で使用するフォントサイズを指定します。

-v | --verbose

起動する前にバージョン情報と Java 実行時引数を出力します。

情報オプション

これらのオプションではパフォーマンスアナライザ GUI は起動されず、analyzer についての情報が標準出力に出力されます。次に示すオプションはそれぞれスタンドアロンのオプションで、ほかの analyzer オプションと組み合わせたり、target または experiment-list 引数と組み合わせることはできません。

-V | --version

バージョン情報を出力して終了します。

-? | --h | --help

使用方法に関する情報を出力して終了します。

アナライザのデフォルト設定

アナライザは、.er.rc というリソースファイルを使用して、起動時の各種設定のデフォルト値を決定します。システム全体のデフォルトファイル er.rc が最初に読み取られ、次にユーザーのホームディレクトリ内の er.rc ファイル (存在する場合)、そして現在のディレクトリ内の er.rc ファイル (存在する場合) が読み取られます。ホームディレクトリの .er.rc ファイル内のデフォルト値はシステムのデフォルト値よりも優先され、現在のディレクトリの .er.rc ファイル内のデフォルト値は、ユーザーのホームおよびシステムのデフォルト値よりも優先されます。.er.rc ファイルは、アナライザと er_print ユーティリティーによって使用されます。ソースおよび逆アセンブリのコンパイラのコメントに適用する .er.rc 内の設定は、er_src ユーティリティーによっても使用されます。

.er.rc ファイルについては、119 ページの「アナライザのデフォルト設定」の節を参照してください。er_print コマンドを使用したデフォルト設定については、149 ページの「デフォルト値を設定するコマンド」と151 ページの「パフォーマンスアナライザにのみデフォルト値を設定するコマンド」を参照してください。

パフォーマンスアナライザ GUI

「アナライザ」ウィンドウは、メニューバー、ツールバー、および各種データ表示のためのタブを含む分割区画で構成されます。

メニューバー

メニューバーには、「ファイル」メニュー、「表示」メニュー、「タイムライン」メニュー、および「ヘルプ」メニューがあります。

「ファイル」メニューは、実験や実験グループを開く、追加する、ドロップする際に使用します。「ファイル」メニューでは、パフォーマンスアナライザ GUI を使って実験のデータを収集できます。パフォーマンスアナライザを使ってデータを収集する方法についての詳細は、118 ページの「アナライザからの実験の記録」を参照してください。「ファイル」メニューから、新しいアナライザウィンドウを作成したり、アナライザに現在表示されているデータをファイルやプリンタに出力したりすることもできます。

「表示」メニューでは、実験データの表示方法を設定できます。

「タイムライン」メニューは、その名前が示すとおり、タイムライン表示の操作を支援します。これについては、95 ページの「アナライザデータ表示」で説明します。

「ヘルプ」メニューは、パフォーマンスアナライザのオンラインヘルプを提供します。これには、新機能の概要を示し、クイックリファレンスとショートカットの説明、およびトラブルシューティングの説明が含まれます。

ツールバー

ツールバーは、ショートカットとして一連のアイコンを提供します。ツールバーには各タブ内のテキストまたは強調表示された行を見つける際に役立つ検索機能が含まれています。検索機能の詳細については、115 ページの「テキストとデータの検索」を参照してください。

アナライザデータ表示

パフォーマンスアナライザは、分割ウィンドウを使って、表示されるデータを 2 つの区画に分割します。各区画にはタブが付けられており、同じ実験や実験グループに対して異なるデータ表示を選択できます。

データ表示、左の区画

左の区画には、主要なアナライザ表示のタブが次の順序で表示されます。

- 「MPI タイムライン」タブ
- 「MPI グラフ」タブ
- 「競合」タブ
- 「デッドロック」タブ
- 「関数」タブ
- 「呼び出し元 - 呼び出し先」タブ
- 「呼び出しツリー」タブ
- 「デュアルソース」タブ
- 「ソース/逆アセンブリ」タブ

- 「ソース」タブ
- 「行」タブ
- 「逆アセンブリ」タブ
- 「PC」タブ
- 「OpenMP 並列領域」タブ
- 「OpenMP タスク」タブ
- 「タイムライン」タブ
- 「リーク一覧」タブ
- 「データオブジェクト」タブ
- 「データレイアウト」タブ
- 「命令頻度」タブ
- 「統計」タブ
- 「実験」タブ
- 各種のメモリーオブジェクトのタブ
- 各種のインデックスオブジェクトのタブ

ターゲットを指定せずにアナライザを起動すると、開く実験を入力するプロンプトが表示されます。

デフォルトでは、表示可能な最初のタブが選択されます。表示されるタブは、読み込まれた実験内のデータに適用できるタブだけです。

実験を開いたときに「アナライザ」ウィンドウの左の区画にタブが表示されるかどうかは、アナライザを起動したときに読み込まれた `.er.rc` ファイル内の tabs 指令と、そのタブを実験内のデータに適用できるかどうかによって決定されます。「データ表示方法の設定」ダイアログボックスの「タブ」タブを (114 ページの「「タブ」タブ」を参照) 使用すると、実験について表示するタブを選択できます。

「MPI タイムライン」タブ

「MPI タイムライン」タブには、MPI 実験の各プロセスに 1 つずつ水平のバーが表示され、それらをつなぐ対角線がメッセージを示します。各バーには、MPI 機能によって色分けされた領域や、プロセスが MPI に含まれていない (アプリケーションコード内の別の場所にある) ことを示す領域があります。

バーのいずれかの領域またはメッセージ行を選択すると、「MPI タイムラインコントロール」タブでの選択に関する詳細情報が表示されます。

マウスをドラッグすると、ドラッグの主な移動方向に応じて、「MPI タイムライン」タブが水平 (時間) 軸方向または垂直 (プロセス) 軸方向にズームインします。

MPI タイムラインの画像をプリンタまたは `.jpg` ファイルに出力できます。「ファイル」→「印刷」を選択し、「印刷」または「ファイル」を選択して、プリンタかファイル名を指定します。

「MPI グラフ」タブ

「MPI グラフ」タブには、「MPI タイムライン」タブに表示される MPI トレースデータのグラフが表示されます。このタブには、MPI の実行に関するデータのプロットが表示されます。「MPI グラフ」タブ内のコントロールを変更して、「変更を表示更新する」をクリックすると、新しいグラフが表示されます。グラフからいずれかの要素を選択すると、その要素の詳細情報が「MPI グラフコントロール」タブに表示されます。

マウスをドラッグすると、ドラッグによって定義された矩形領域上で、「MPI グラフ」タブがズームインします。

MPI グラフ画像をプリンタまたは `.jpg` ファイルに出力できます。「ファイル」→「印刷」を選択し、「印刷」または「ファイル」を選択して、プリンタかファイル名を指定します。

「競合」タブ

「競合」タブには、データ競合実験で検出されたすべてのデータ競合のリストが表示されます。詳細は、『Oracle Solaris Studio 12.2: スレッドアナライザユーザーズガイド』を参照してください。

「デッドロック」タブ

「デッドロック」タブには、デッドロック実験で検出されたデッドロックのリストが表示されます。詳細は、『Oracle Solaris Studio 12.2: スレッドアナライザユーザーズガイド』を参照してください。

「関数」タブ

「関数」タブには、関数とそのメトリックスが一覧表示されます。メトリックスは、実験で収集されたデータから得られます。メトリックスは、排他的メトリックスまたは包括的メトリックスのいずれかになります。排他的メトリックスは、その関数自体のみによる使用を表します。包括的メトリックスは、関数とその関数が呼び出したすべての関数による使用を表します。

収集されたそれぞれの種類のデータで使用できるメトリックスのリストは、`collect`(1) のマニュアルページを参照してください。表示されるのは、メトリックスがゼロ以外の関数だけです。

時間メトリックスは秒で表示され、ミリ秒の精度で提供されます。百分率は 0.01% の精度で表示されます。メトリック値が正確にゼロである場合、その時間と百分率は「0」として表示されます。メトリック値が正確にゼロではなく、精度よりも小さい場合、その値は「0.000」として表示され、その百分率は「0.00」として表示されます。丸めのため、百分率は合計が正確に 100 % にならない場合もあります。カウントメトリックスは整数カウントとして示されます。

初めに表示されるメトリックスは、収集されたデータおよび各種 .er.rc ファイルから読み取られたデフォルト設定に基づいています。パフォーマンスアナライザを最初にインストールしたときのデフォルトは、次のようになります。

- 時間ベースのプロファイリングでは、デフォルトセットは包括的および排他的ユーザー CPU 時間から構成されます。
- 同期遅延トレースでは、デフォルトセットは包括的同期待ちカウントと包括的同期待ち時間から構成されます。
- ハードウェアカウンタオーバーフローのプロファイリングでは、デフォルトセットは、包括的および排他的時間 (循環カウントのカウンタの場合) またはイベントカウント (ほかのカウンタの場合) から構成されます。
- ヒープトレースでは、デフォルトセットはヒープリークとリークしたバイト数から構成されます。

複数の種類のデータが収集された場合、各種類のデフォルトのメトリックスが表示されます。

表示されるメトリックスは、変更または再構成できます。詳細はオンラインヘルプを参照してください。

関数を検索するには、Find ツールを使用します。「検索」ツールの詳細については、115 ページの「テキストとデータの検索」を参照してください。

単一の関数を選択するには、その関数をクリックします。

タブ内に隣接して表示された複数の関数を選択するには、グループの最初の関数を選択したあと、Shift キーを押したままグループの最後の関数をクリックします。

タブ内に表示された隣接していない複数の関数を選択するには、グループの最初の関数を選択したあと、Ctrl キーを押したまま、追加する関数を個々にクリックして選択します。

ツールバーの「フィルタ句を構成」ボタンをクリックすると、「データをフィルタ」ダイアログボックスが開き、その中で「詳細」タブが選択され、「フィルタ句」テキストボックスに「関数」タブで選択された 1 つまたは複数の項目を反映したフィルタ句が読み込まれます。

「関数」タブを右クリックして、コンテキストメニューを開き、選択した関数に対して定義済みフィルタを設定できます。コンテキストメニューには次の項目があります。

フィルタの設定: スタック内の関数
フィルタの設定: リーフ関数
フィルタの設定: スタック内の名前

いずれかのオプションを選択すると、「データをフィルタ」ダイアログボックスが開き、その中で「詳細」タブが選択され、「フィルタ句」テキストボックスに選択

したフィルタを実装するフィルタ式が表示されます。「設定」をクリックし、「適用」をクリックすると、データがフィルタされます。詳細は、116 ページの「データのフィルタリング」を参照してください。

「呼び出し元 - 呼び出し先」タブ

「呼び出し元 - 呼び出し先」タブには、コード内の関数間の呼び出し関係がパフォーマンスメトリックスとともに表示されます。「呼び出し元 - 呼び出し先」タブでは、1 度に呼び出し 1 つずつ、呼び出しスタックフラグメントが構築されるので、コード分岐のメトリックスを詳細に確認できます。

このタブには、一番上に「呼び出し元」パネル、中央に「スタックフラグメント」パネル、一番下に「呼び出し先」パネルの 3 つのパネルがあります。「呼び出し元 - 呼び出し先」タブを初めて開くと、「スタックフラグメント」パネル上の関数は、「関数」タブや「ソース」タブなど、ほかのアナライザタブで選択された関数になります。「呼び出し元」パネルには、「スタックフラグメント」パネル上の関数を呼び出す関数が一覧表示され、「呼び出し先」パネルには、「スタックフラグメント」パネル上の関数によって呼び出される関数が一覧表示されます。

1 度に呼び出し 1 つずつについて、呼び出し元と呼び出し先を呼び出しスタックに追加することにより、1 つの関数を中心にして呼び出しスタックフラグメントを構築できます。

「呼び出し元」パネルまたは「呼び出し先」パネル上の関数をダブルクリックするか、関数を 1 つ選択し、「追加」ボタンをクリックすることにより、スタックフラグメントに呼び出しを追加できます。同様に、呼び出しスタックフラグメントの一番上または一番下の関数をダブルクリックするか、一番上か一番下の関数を選択し、「削除」をクリックすることにより、関数呼び出しを削除できます。この追加または削除は、コンテキストメニューからでも実行できます。関数を右クリックし、該当するコマンドを選択します。

関数を選択し、「先頭を設定」、「中心を設定」、または「末尾を設定」を選択することにより、関数を呼び出しスタックフラグメントの先頭 (一番上)、中央、または末尾 (一番下) として設定できます。この新しい順序により、呼び出しスタックフラグメント内に現在あるほかの関数が「呼び出し元」領域と「呼び出し先」領域のいずれか、スタックフラグメント内の選択された関数の新しい場所との関係において適切な場所に移動します。

「スタックフラグメント」パネルの上にある「戻る」ボタンと「次へ」ボタンを使用すると、呼び出しスタックフラグメントへの変更履歴間を移動することができます。

スタックフラグメント内の関数を追加したり削除したりすると、メトリックスがフラグメント全体に対して計算され、そのフラグメント内の最後の関数の隣に表示されます。

「呼び出し元 - 呼び出し先」タブ内を右クリックして、コンテキストメニューを開き、選択された関数に対して定義済みのフィルタを設定できます。コンテキストメニューには次の項目があります。

フィルタの設定: スタック内のスタックフラグメント
フィルタの設定: スタック内の関数
フィルタの設定: リーフ関数
フィルタの設定: スタック内の名前

いずれかのオプションを選択すると、「データをフィルタ」ダイアログボックスが開き、「詳細」タブが選択され、「フィルタ句」テキストボックスに、「フィルタの設定」で選択したフィルタを実装するフィルタ式が表示されます。「設定」をクリックし、「適用」をクリックすると、データがフィルタされます。詳細は、116 ページの「データのフィルタリング」を参照してください。

「呼び出し元 - 呼び出し先」タブには、属性メトリックスが表示されます。

- 「スタックフラグメント」パネル内の呼び出しスタックフラグメントについて、属性メトリックスは、その呼び出しスタックフラグメントの排他的メトリックスを表します。
- 呼び出し先の場合、属性メトリックは、呼び出し先のメトリックのうち、呼び出しスタックフラグメントからの呼び出しに起因する部分を表します。呼び出し先と呼び出しスタックフラグメントの属性メトリックスの合計は、呼び出しスタックフラグメントのメトリックスにもなるはずです。
- 呼び出し元の場合、属性メトリックスは、呼び出しスタックフラグメントのメトリックスのうち、呼び出し元からの呼び出しに起因する部分を示します。すべての呼び出し元の属性メトリックスの合計が、呼び出しスタックフラグメントのメトリックにもなるはずです。

メトリックスについての詳細は、38 ページの「関数レベルのメトリックス: 排他的、包括的、属性」を参照してください。

「呼び出しツリー」タブ

「呼び出しツリー」タブでは、プログラムの動的な呼び出しグラフがツリー表示されます。各関数はノードとして表示され、展開したり折りたたんだりできます。展開された関数ノードには、その関数によって行われる関数呼び出しがすべて表示され、加えて、それらの関数呼び出しのパフォーマンスメトリックスも表示されます。ノードを選択すると、右側の「概要」タブに、関数呼び出しのメトリックスとその呼び出し先が表示されます。属性メトリックスに対して提供される割合は、プログラムメトリックス全体の割合になります。ツリーのデフォルトルートは `<Total>` で、関数ではありませんが、プログラムの全関数のパフォーマンスメトリックスの 100% を表します。

「呼び出しツリー」タブでは、特定の呼び出しトレースまで掘り下げて、どのトレースがパフォーマンスに一番大きく影響しているかを探ることができます。高いメトリック値を探しながら、プログラムの構造内を移動できます。

ヒント – 多くの時間を消費している分岐を簡単に見つけるには、任意のノードを右クリックし、「一番ホットな分岐を展開」を選択します。

「呼び出しツリー」タブを右クリックして、コンテキストメニューを開き、選択した関数に対して定義済みフィルタを設定できます。コンテキストメニューには次の項目があります。

フィルタの設定: スタック内の呼び出しパス
フィルタの設定: スタック内の関数
フィルタの設定: リーフ関数
フィルタの設定: スタック内の名前

いずれかのオプションを選択すると、「データをフィルタ」ダイアログボックスが開き、「詳細」タブが選択され、「フィルタ句」テキストボックスに、「フィルタの設定」で選択したフィルタを実装するフィルタ式が表示されます。「設定」をクリックし、「適用」をクリックすると、データがフィルタされます。詳細は、116 ページの「データのフィルタリング」を参照してください。

「デュアルソース」タブ

「デュアルソース」タブには、選択したデータ競合またはデッドロックに関わる 2 つのソースコンテキストが表示されます。このタブは、データ競合検出またはデッドロック実験が読み込まれている場合にのみ表示されます。

「ソース/逆アセンブリ」タブ

「ソース/逆アセンブリ」タブでは、上の区画に注釈付きソースが、下の区画には注釈付き逆アセンブリが表示されます。このタブはデフォルトでは表示されません。「ソース/逆アセンブリ」タブを追加するには、「表示」メニューの「データ表示方法の設定」オプションを使用します。

「ソース」タブ

「ソース」タブには、選択された関数のソースコードが利用可能である場合に、そのソースコードを含むファイルが、各ソース行の左側の列にパフォーマンスメトリックスの注釈付きで表示されます。高いメトリックスはオレンジで表示され、リソース利用率のホット領域であるソース行が示されます。また、オレンジのナビゲーションマーカーが、各ホットソース行の右側のスクロールバーの隣のマージン内に表示されます。ホットしきい値を下回るゼロ以外のメトリックスは強調表示されませんが、黄色のナビゲーションマーカーが表示されます。メトリックス付きのソース行にすばやく移動するには、右側マージン内のオレンジと黄色のマーカーをクリックすると、メトリックス付きの行にジャンプできます。また、メトリックス自体を右クリックし、「次のホットライン」や「次のゼロ以外のメトリックライン」などのオプションを選択して、メトリックス付きの次の行にジャンプすることができます。

強調表示メトリックスのしきい値は、「データ表示方法の設定」ダイアログボックスの「ソース/逆アセンブリ」タブで設定できます。デフォルトのしきい値は、.er.rc デフォルト値ファイルで設定できます。.er.rc ファイルについての詳細は、119 ページの「アナライザのデフォルト設定」を参照してください。

「ソース」タブには、ソースファイルおよび対応するオブジェクトファイルへの完全パスが表示され、ソースコードの列見出しにはロードオブジェクトの名前が表示されます。まれに、1 つのソースファイルを使って複数のオブジェクトファイルがコンパイルされている場合は、選択された関数を含むオブジェクトファイルのパフォーマンスデータが「ソース」タブに表示されます。

アナライザは、実行可能ファイルに記録されているような絶対パス名で、選択された関数を含むファイルを探します。そのようなファイルが存在しない場合、アナライザは、現在の作業ディレクトリから同じベース名のファイルを探そうと試みます。ソースを移動した場合、また別のファイルシステムに実験が記録された場合は、注釈付きソースを表示するために、現在のディレクトリにソースのコピーまたは元のパス名へのシンボリックリンクを設定できます。また、「データ表示方法の設定」ダイアログボックスの設定を使用することもできます。「検索パス」タブ (114 ページの「「検索パス」タブ」を参照) では、ソースファイルの検索に使用するディレクトリのリストを管理できます。「パスマップ」タブ (114 ページの「「パスマップ」タブ」を参照) では、ファイルパスの先頭部分を、ある場所から別の場所へマッピングできます。

「関数」タブで関数を選択し、「ソース」タブを開いたとき、表示されるソースファイルは、その関数のデフォルトソースコンテキストです。関数のデフォルトソースコンテキストは、その関数の最初の命令 (C コードの場合は、開く中括弧) を含むファイルです。注釈付きソースファイルでは、最初の命令の直後にその関数のインデックス行が追加されます。ソースウィンドウには、インデックス行が赤いイタリック体のテキストとして山括弧内に次の形式で表示されます。

<Function: f_name>

関数は、代替のソースコンテキストを持つ場合もあります。代替のソースコンテキストは、その関数に帰属する命令が入った別のファイルです。そのような命令は、インクルードファイルに入っているか、選択された関数内にインライン化された別の関数に入っている命令です。代替のソースコンテキストが存在する場合、デフォルトのソースコンテキストの先頭に、代替ソースコンテキストが置かれている場所を示す拡張インデックス行のリストが組み込まれます。

<Function: f, instructions from source file src.h>

別のソースコンテキストを参照するインデックス行をダブルクリックすると、そのソースコンテキストが入っているファイルが、インデックスで示された関数に関連した位置から開かれます。

ナビゲーションに役立つよう、代替ソースコンテキストも、そのデフォルトソースコンテキストおよびほかの代替ソースコンテキスト内で定義された関数を逆に参照するインデックス行のリストで始まります。

ソースコードは、表示用に選択されているコンパイラのコメントとともにインターリーブされます。表示するコメントのクラスは、「データ表示方法の設定」ダイアログボックスで設定できます。デフォルトのクラスは、`.er.rc` デフォルト値ファイルで設定できます。

「ソース」タブに表示されるメトリックスは、変更または再構成できます。詳細はオンラインヘルプを参照してください。

「行」タブ

「行」タブには、ソース行とそのメトリックスのリストが表示されます。ソース行は、そのラベルとして、呼び出し元の関数、行番号、およびソースファイル名で示されます。関数の行番号情報が得られない場合、または関数のソースファイルが不明の場合は、その関数の全プログラムカウンタ (Program Counter、PC) が、行表示でその関数の 1 つのエントリにまとめて表されます。関数が表示されないロードオブジェクトにある関数の PC は、ロードオブジェクト別に「行」表示で 1 つのエントリにまとめて表示されます。「行」タブで行を選択すると、その行の全メトリックスが「概要」タブに表示されます。「行」タブで行を選択してから「ソース」タブか「逆アセンブリ」タブを選択すると、適切な行に表示が位置付けられます。

「逆アセンブリ」タブ

「逆アセンブリ」タブには、選択した関数が含まれているオブジェクトファイルの逆アセンブリリストが表示され、各命令のパフォーマンスメトリックスが注釈として付きます。

ソースコード情報が得られる場合、逆アセンブリリスト内にそのソースコードがインタリーブされて、任意のコンパイラのコメントが表示用に選択されます。「逆アセンブリ」タブでソースファイルを見つけるためのアルゴリズムは、「ソース」タブで使用されるアルゴリズムと同じです。

「ソース」タブと同様に、「逆アセンブリ」タブにはインデックス行が表示されます。しかし、「ソース」タブとは異なり、代替ソースコンテキストのインデックス行をナビゲーションの目的で直接使用することはできません。また、代替ソースコンテキストのインデックス行は、「逆アセンブリ」表示の先頭に単に一覧表示されるのではなく、`#include` またはインライン化されたコードの挿入位置の先頭に表示されます。`#include` またはほかのファイルからインライン化されたコードは、raw の逆アセンブリ命令として、ソースコードをインタリーブせずに示されます。ただし、これらの命令の 1 つにカーソルを置いて「ソース」タブを選択すると、`#include` またはインライン化されたコードを含むファイルが開きます。このファイルを表示した状態で「逆アセンブリ」タブを選択すると、新しいコンテキストで「逆アセンブリ」表示が開き、インタリーブされたソースコードと一緒に逆アセンブリコードが表示されます。

表示するコメントのクラスは、「データ表示方法の設定」ダイアログボックスで設定できます。デフォルトのクラスは、.er.rc デフォルト値ファイルで設定できます。

メトリックスがメトリック固有のしきい値と等しいか、それを超える場合、アナライザによって行が強調表示されるため、重要な行を容易に識別できます。しきい値は、「データ表示方法の設定」ダイアログボックスで設定できます。デフォルトのしきい値は、.er.rc デフォルト値ファイルで設定できます。「ソース」タブの場合と同様に、逆アセンブリコード内のしきい値を超えている行の位置に対応して、スクロールバーの横にティックマークが表示されます。

「PC」タブ

「PC」タブには、プログラムカウンタ (Program Counter、PC) とそのメトリックスのリストが表示されます。PC には、呼び出し元の関数、およびその関数内のオフセットがラベルとして表示されます。関数が表示されないロードオブジェクトにある関数の PC は、ロードオブジェクト別に「PC」表示に 1 つのエントリにまとめて表示されます。「PC」タブで行を選択すると、その PC の全メトリックスが「概要」タブに表示されます。「PC」タブで行を選択してから「ソース」タブか「逆アセンブリ」タブを選択すると、適切な行に表示が位置付けられます。

PC の詳細については、171 ページの「呼び出しスタックとプログラムの実行」の節を参照してください。

「OpenMP 並列領域」タブ

「OpenMP 並列領域」タブは、OpenMP 3.0 のコレクタで記録された実験に対してのみ適用できます。このタブには、プログラムの実行中に発生したすべての並列領域が、同じプロファイルデータから計算されたメトリック値とともに一覧表示されます。現在の並列領域に対しては、排他的メトリックスが計算されます。包括的メトリックスには、入れ子並列性が反映されます。これらは、現在の並列領域、およびこれが作成された親並列領域、そしてさらに再帰的に、プログラムの直列実行を表す (すべての並列領域の外側の) 最上位の暗黙 OpenMP 並列領域にまで属します。プログラムに入れ子の並列領域がない場合、排他的メトリックスと包括的メトリックスは同じ値を持ちます。

並列領域を含む関数が複数回呼び出された場合、並列領域のすべてのインスタンスが集計され、対応するタブに 1 項目として表示されます。

このタブはナビゲーションに便利です。OpenMP 待ち時間が最大値の並列領域など、興味のある項目を選択して、そのソースを解析したり、これに基づいてフィルタを作成および適用して、「関数リスト」、「タイムライン」、「スレッド」などのほかのタブを使用して、ほかのプログラムオブジェクトによってどのように表されるかを解析したりできます。

「OpenMP タスク」タブ

「OpenMP タスク」タブには、OpenMP タスクとそれぞれのメトリックスの一覧が表示されます。このタブは、OpenMP 3.0 のコレクタで記録された実験に対してのみ適用できます。

「OpenMP タスク」タブは、Oracle Solaris Studio でコンパイルした OpenMP タスクを使用するプログラムについて、OpenMP 3.0 コレクタで記録されている実験にのみ適用されます。

このタブには、プログラムの実行中に発生したタスクが、プロファイルデータから計算されたメトリック値とともに一覧表示されます。排他的メトリックスは、現在のタスクのみに適用されます。包括的メトリックスは、OpenMP タスクのメトリックス、および、タスク生成時に構築された親子関係を持つ、子タスクのメトリックスを含みます。暗黙並列領域の OpenMP タスクは、プログラムの直列実行を表します。

タスクを含む関数が複数回呼び出される場合、並列領域のすべてのインスタンスが集計され、対応するタブに 1 項目として表示されます。

このタブはナビゲーションに便利です。OpenMP 待ち時間が最大値のタスクなど、興味のある項目を選択して、そのソースを解析したり、これに基づいてフィルタを作成および適用して、「関数リスト」、「タイムライン」、「スレッド」などのほかのタブを使用して、ほかのプログラムオブジェクトによってどのように表されるかを解析したりできます。

「タイムライン」タブ

「タイムライン」タブには、コレクタによって記録されたイベントおよび標本ポイントのグラフが、時間の関数として表示されます。データは、水平バーに表示されます。それぞれの実験について、標本データに対応するバーと、各 LWP に対応する一連のバーが表示されます。LWP のセットは、記録される各データ型ごとに 1 つのバーで構成されます。時間ベースのプロファイリング、ハードウェアカウンタオーバーフローのプロファイリング、同期トレース、ヒープトレース、および MPI トレースなどがあります。

標本データを含むバーは、各標本の各マイクロステートで費やされた時間の色分け表現です。標本ポイントのデータは、そのポイントと前のポイントの間で費やされた時間を表すため、標本は時間として表示されます。標本をクリックすると、その標本のデータが「イベント」タブに表示されます。

プロファイルデータまたはトレースデータのバーには、記録される各イベントのイベントマーカーが表示されます。イベントマーカーは、イベントとともに記録された呼び出しスタックの色分けされた表現 (色付きの長方形が積み重ねられたもの) からなります。イベントマーカー内の色付き長方形をクリックすると、対応する関数と PC が選択され、そのイベントと関数のデータが「イベント」タブに表示されま

す。選択された項目は「イベント」タブと「凡例」タブの両方で強調表示され、「ソース」タブまたは「逆アセンブリ」タブを選択すると、呼び出しスタック内のそのフレームに対応する行にタブ表示が位置付けられます。

ある種のデータでは、イベントが重なって見えない場合があります。まったく同じ位置に複数のイベントが表示される場合は、常に 1 つだけが描画されます。1 - 2 ピクセル以内に複数のイベントがある場合、すべてが描画されますが、見た目には判別できない可能性があります。いずれの場合も、描画されたイベントの下に小さな灰色のティックマークが表示され、重なっていることが示されます。

「データ表示方法の設定」ダイアログボックスの「タイムライン」タブでは、表示するイベント固有データの種類を変更したり、スレッド、LWP、または CPU に関するイベント固有データの表示を選択したり、ルートまたはリーフでの呼び出しスタックの表示の配置を選択したり、表示する呼び出しスタックのレベル数を選択したりできます。

「タイムライン」タブに表示するイベント固有データの種類や、選択された関数にマップする色も変更できます。「タイムライン」タブの使い方の詳細は、オンラインヘルプを参照してください。

「リーク一覧」タブ

「リーク一覧」タブには 2 つの行が表示され、上の行はリークを表し、下の行は割り当てを示します。それぞれには、「タイムライン」タブで表示されているものと同じような呼び出しスタックが中央に表示され、その上にはリークまたは割り当てられたバイト数に比例するバーが、その下にはリークまたは割り当ての数に比例するバーが表示されます。

リークまたは割り当てを選択すると、選択されたリークや割り当てのデータが「リーク」タブに表示され、「タイムライン」タブの場合と同様に呼び出しスタックのフレームが選択されます。

「リーク一覧」タブを表示するには、「データ表示方法の設定」ダイアログボックスの「タブ」タブ (114 ページの「「タブ」タブ」を参照) で、そのタブを選択します。「リーク一覧」タブを表示可能にできるのは、1 つ以上の読み込まれた実験の中に、ヒープトレースデータが含まれている場合だけです。

「データオブジェクト」タブ

「データオブジェクト」タブには、データオブジェクトおよびそのメトリックスが一覧表示されます。このタブは、ハードウェアカウンタオーバーフロープロファイリングの拡張版であるデータ空間プロファイリングを含む実験にのみ使用されます。詳細は、170 ページの「データ空間プロファイリング」を参照してください。

タブを表示するには、「データ表示方法の設定」ダイアログボックスの「タブ」タブ (114 ページの「「タブ」タブ」を参照) で、そのタブを選択します。「データオブ

ジェクト」タブを表示可能にできるのは、1 つ以上の読み込まれた実験に、データ空間プロファイルが含まれている場合だけです。

このタブには、プログラムのさまざまなデータ構造体と変数に対するハードウェアカウンタのメモリー演算のメトリックスが示されます。

単一のデータオブジェクトを選択するには、そのオブジェクトをクリックします。

タブ内に隣接して表示された複数のオブジェクトを選択するには、最初のオブジェクトを選択したあと、Shift キーを押したまま最後のオブジェクトを選択します。

タブ内に表示された隣接していない複数のオブジェクトを選択するには、最初のオブジェクトを選択したあと、Ctrl キーを押したまま、追加するオブジェクトを個々にクリックして選択します。

ツールバーの「フィルタ句を構成」ボタンをクリックすると、「フィルタ」ダイアログボックスが開き、その中で「詳細」タブが選択され、「フィルタ句」テキストボックスに「データオブジェクト」タブで選択された項目を反映したフィルタ句が読み込まれます。

「データレイアウト」タブ

「データレイアウト」タブには、すべてのプログラムデータオブジェクトの注釈付きデータオブジェクトレイアウトが、データ派生メトリックデータと一緒に表示されます。このタブは、ハードウェアカウンタオーバーフロープロファイリングの拡張版であるデータ空間プロファイリングを含む実験にのみ使用されます。詳細は、170 ページの「データ空間プロファイリング」を参照してください。

各レイアウトは、構造全体のデータソートメトリックス値によってソートされた状態で、タブ内に表示されます。このタブには、集合体データオブジェクトごとに、そのオブジェクトに帰属する合計メトリックスが表示され、そのあとに、そのデータオブジェクトのすべての要素がオフセット順に表示されます。各要素には、そのメトリックスと、32 バイトブロックでそのサイズと位置を示す指示子があります。

「データレイアウト」タブを表示するには、「データ表示方法の設定」ダイアログボックスの「タブ」タブ (114 ページの「「タブ」タブ」を参照) で、そのタブを選択します。「データオブジェクト」タブの場合と同様に、「データレイアウト」タブを表示可能にできるのは、1 つ以上の読み込まれた実験に、データ空間プロファイルが含まれている場合だけです。

単一のデータオブジェクトを選択するには、そのオブジェクトをクリックします。

タブ内に隣接して表示された複数のオブジェクトを選択するには、最初のオブジェクトを選択したあと、Shift キーを押したまま最後のオブジェクトを選択します。

タブ内に表示された隣接していない複数のオブジェクトを選択するには、最初のオブジェクトを選択したあと、Ctrl キーを押したまま、追加するオブジェクトを個々にクリックして選択します。

ツールバーの「フィルタ句を構成」ボタンをクリックすると、「フィルタ」ダイアログボックスが開き、その中で「詳細」タブが選択され、「フィルタ句」テキストボックスに「データレイアウト」タブで選択された項目を反映したフィルタ句が読み込まれます。

「命令頻度」タブ

「命令頻度」タブには、カウントデータ実験で各種類の命令が実行された頻度の概要が表示されます。また、ロード、ストア、浮動小数点の各命令の実行頻度に関するデータも示します。無効になった命令や、分岐遅延スロット内の命令に関する情報も表示されます。

「統計」タブ

「統計」タブには、選択した実験と標本について集計されたさまざまなシステム統計の合計値が表示されます。合計値のあとには、それぞれの実験について選択した標本の統計値が表示されます。表示される統計値については、`getrusage`(3C) と `proc`(4) のマニュアルページを参照してください。

「実験」タブ

「実験」タブは 2 つのパネルに分割されます。上のパネルにはツリーが入っており、このツリーには、読み込まれたすべての実験に含まれるロードオブジェクトのノード、およびそれぞれの実験読み込みのノードが含まれています。「ロードオブジェクト」ノードを展開すると、すべてのロードオブジェクトのリストが、それらの処理に関するさまざまなメッセージと一緒に表示されます。実験のノードを展開すると、「注記」と「情報」という 2 つの領域が表示されます。

「注記」領域には、実験内の notes ファイルの内容が表示されます。「注記」領域に直接入力することにより、注記を編集できます。「注記」領域には独自のツールバーが組み込まれており、注記の保存や破棄のほか、前回の保存以降の編集内容の取り消しや再実行を行うためのボタンがあります。

「情報」領域には、収集した実験と収集ターゲットがアクセスしたロードオブジェクトに関する情報が入っており、それには、実験またはロードオブジェクトの処理中に出力されたエラーメッセージや警告メッセージも含まれます。

下のパネルには、アナライザセッションから出力されたエラーメッセージと警告メッセージが表示されます。

インデックスタブ

各インデックスタブには、スレッド、CPU、秒など、さまざまなインデックスオブジェクトに帰属するデータのメトリック値が表示されます。インデックスオブジェクトは階層構造ではないので、包括的メトリックスと排他的メトリックスは表示されません。各種類の単一のメトリックだけが表示されます。

事前定義されているいくつかのインデックスタブは、スレッド、CPU、標本、および秒です。カスタムインデックスオブジェクトを定義するには、「データ表示方法の設定」ダイアログボックスで「「カスタムインデックス」タブを追加」ボタンをクリックし、「インデックスオブジェクトを追加」ダイアログボックスを開きます。インデックスオブジェクトは、`.er.rc` ファイルで `indxobj_define` 指令を使用して定義することもできます (139 ページの「`indxobj_define` *indxobj_type index_exp*」 を参照)。

各インデックスタブの上部のラジオボタンを使用して、テキスト表示とグラフィカル表示を切り替えることができます。テキスト表示は「データオブジェクト」タブ内の表示に似ており、同じメトリックの設定を使用します。グラフィカル表示では、各インデックスオブジェクトの相対値が、データソートメトリックによってソートされた各メトリックごとの個別のヒストグラムを使用してグラフィカル表示されます。

ツールバーの「データをフィルタ」ボタンをクリックすると、「データをフィルタ」ダイアログボックスが開きます。「詳細」タブをクリックすると、「インデックスオブジェクト」タブで選択された項目を反映したフィルタ句が「フィルタ句」に読み込まれます。

メモリーオブジェクトのタブ

各メモリーオブジェクトのタブには、ページなど、さまざまなメモリーオブジェクトに帰属するデータ空間メトリックスのメトリック値が表示されます。1 つ以上の読み込まれた実験にデータ空間プロファイルが含まれている場合は、「データ表示方法の設定」ダイアログボックスの「タブ」タブで、タブを表示するメモリーオブジェクトを選択できます。メモリーオブジェクトタブは、いくつでも表示できます。

さまざまなメモリーオブジェクトタブが事前定義されています。カスタムメモリーオブジェクトを定義するには、「データ表示方法の設定」ダイアログボックスで「カスタムオブジェクトを追加」ボタンをクリックし、「メモリーオブジェクトを追加」ダイアログボックスを開きます。メモリーオブジェクトは、`.er.rc` ファイルで `mobj_define` 指令を使用して定義することもできます (137 ページの「`mobj_define` *mobj_type index_exp*」 を参照)。

各メモリーオブジェクトタブのラジオボタンを使用して、テキスト表示とグラフィカル表示を切り替えることができます。テキスト表示は「データオブジェクト」タブ内の表示に似ており、同じメトリックの設定を使用します。グラフィカル

表示では、各メモリーオブジェクトの相対値が、データソートメトリックによってソートされた各メトリックごとの個別のヒストグラムを使用してグラフィカル表示されます。

ツールバーの「フィルタ句を構成」ボタンをクリックすると、「フィルタ」ダイアログボックスが開き、その中で「詳細」タブが選択され、「フィルタ句」テキストボックスにメモリーオブジェクトタブで選択された項目を反映したフィルタ句が読み込まれます。

データ表示、右の区画

右の区画には、「MPI タイムラインコントロール」タブ、「MPI グラフコントロール」タブ、「概要」タブ、「イベント」タブ、「競合の詳細」タブ、「デッドロックの詳細」タブ、および「リーク」タブがあります。デフォルトでは、「概要」タブが表示されます。

「MPI タイムラインコントロール」タブ

「MPI タイムラインコントロール」タブは、「MPI タイムライン」タブのズーム、パン、イベントステップ、およびフィルタリングをサポートします。このタブには、「MPI タイムライン」タブに表示される MPI メッセージの割合を調整するコントロールがあります。

フィルタリングでは、ビューの現在のフィールドの外部にあるデータが、「MPI タイムライン」タブおよび「MPI グラフ」タブに表示されるデータセットから削除されます。フィルタを適用するには、「フィルタ」ボタンをクリックします。フィルタの戻るボタンは最後に適用したフィルタを取り消す場合に使用し、フィルタの進むボタンはフィルタを再適用する場合に使用します。フィルタは「MPI タイムライン」タブと「MPI グラフ」タブの間で共有されますが、現在のところそのほかのタブには適用されません。

メッセージスライダを調整して、表示されるメッセージの割合を制御できます。100% 未満に指定すると、最もコストの高いメッセージが優先されます。コストは、メッセージの送受信に費やされる時間で定義されます。

また、「MPI タイムラインコントロール」タブを使用して、「MPI タイムライン」タブでの機能やメッセージの選択に関する詳細を表示することもできます。

「MPI グラフコントロール」タブ

「MPI グラフコントロール」タブには、グラフの種類、X 軸と Y 軸のパラメータ、およびデータの集計に使用されるメトリックや演算子を制御するための、一連のドロップダウンリストがあります。「変更を表示更新する」をクリックすると、新しいグラフが描画されます。

フィルタリングでは、ビューの現在のフィールドの外部にあるデータが、「MPI タイムライン」タブおよび「MPI グラフ」タブに表示されるデータセットから削除されます。フィルタを適用するには、「フィルタ」ボタンをクリックします。フィルタの戻るボタンは最後に適用したフィルタを取り消す場合に使用し、フィルタの進むボタンはフィルタを再適用する場合に使用します。

また、「MPI グラフコントロール」タブを使用して、「MPI グラフ」タブでの選択に関する詳細を表示することもできます。

「概要」タブ

「概要」タブには、選択した関数やロードオブジェクトについて記録されたすべてのメトリックス (値と百分率)、および選択した関数やロードオブジェクトについての情報が表示されます。「概要」タブは、任意のタブで新しく関数やロードオブジェクトを選択するたびに更新されます。

「イベント」タブ

「イベント」タブには、イベントタイプ、リーフ関数、LWP ID、スレッド ID、および CPU ID など、「タイムライン」タブで選択されたイベントの詳細なデータが表示されます。データパネルの下に、スタック内の関数ごとに色分けされた呼び出しスタックが表示されます。呼び出しスタック内の関数をクリックすると、その関数が選択されます。

「タイムライン」タブで標本を選択すると、その標本番号、標本の開始時間と終了時間、および各マイクロステートで費やされた時間を示す色別のマイクロステートが「イベント」タブに表示されます。

「リーク」タブ

「リーク」タブには、「リーク一覧」タブ内で選択したリークまたは割り当ての詳細データが表示されます。「リーク」タブのデータパネルの下には、選択したリークまたは割り当てが検出されたときの呼び出しスタックが表示されます。呼び出しスタック内の関数をクリックすると、その関数が選択されます。

「競合の詳細」タブ

「競合の詳細」タブには、「競合」タブ内で選択したデータ競合の詳細データが表示されます。詳細は、『Oracle Solaris Studio 12.2: スレッドアナライザユーザーズガイド』を参照してください。

「デッドロックの詳細」タブ

「デッドロックの詳細」タブには、「デッドロック」タブ内で選択したデッドロックの詳細データが表示されます。詳細は、『Oracle Solaris Studio 12.2: スレッドアナライザユーザーズガイド』を参照してください。

データ表示オプションの設定

データの表示は、「データ表示方法の設定」ダイアログボックスで制御できます。このダイアログボックスは、ツールバーの「データ表示方法の設定」ボタンをクリックするか、「表示」→「データ表示方法の設定」を選択することで、開くことができます。

「データ表示方法の設定」ダイアログボックスには、次のタブを持つタブ区画が含まれています。

- メトリックス
- ソート
- ソース/逆アセンブリ
- 書式
- タイムライン
- 検索パス
- パスマップ
- タブ

このダイアログボックスの「OK」ボタンをクリックすると、現在のセッションに対して行った変更が適用され、ダイアログボックスが閉じます。「適用」ボタンをクリックすると、現在のセッションに対して、変更が適用されますが、ダイアログボックスは開いたままなので、変更を続行できます。

「保存」ボタンをクリックすると、カスタム定義されたメモリーオブジェクトを含む現在の設定が、ホームディレクトリまたは現在の作業ディレクトリ内の `.er.rc` ファイルに保存されます。設定を保存することにより、変更は、現在のセッションに加えて、将来のアナライザのセッションにも適用されます。

注 – `.er.rc` ファイルは、アナライザ、`er_print` ユーティリティー、および `er_src` ユーティリティーのデフォルト設定を指定します。「データの設定」ダイアログボックスで変更を保存すると、`.er.rc` ファイルが更新され、3 つすべてのユーティリティーの出力に影響します。`.er.rc` ファイルについての詳細は、119 ページの「アナライザのデフォルト設定」を参照してください。

「メトリックス」タブ

「メトリックス」タブには、使用できるすべてのメトリックスが表示されます。各メトリックの 1 つ以上の列には、メトリックの種類に応じて「時間」、「値」、および「%」というラベルの付いたチェックボックスが表示されます。別の方法として、個々のメトリックスを設定する代わりに、ダイアログボックスの下部の行にある複数のチェックボックスをオンまたはオフにしてから「すべてのメトリックスに適用」ボタンをクリックすることにより、すべてのメトリックスを一度に設定できます。

「ソート」タブ

「ソート」タブには、メトリックスが表示される順序と、ソート基準のメトリックスが表示されます。

「ソース/逆アセンブリ」タブ

「ソース/逆アセンブリ」タブには、次のような表示情報を選択するためのチェックボックスのリストが表示されます。

- ソースリストと逆アセンブリリストに含めるコンパイラのコメント
- ソースリストと逆アセンブリリストで重要な行を強調表示するためのしきい値
- 逆アセンブリリストにおけるソースコードのインタリーブ
- 逆アセンブリリストにおけるソース行のメトリックス
- 逆アセンブリリストにおける 16 進での命令の表示

「書式」タブ

「書式」タブでは、C++ 関数名と Java メソッド名に、長い形式、短い形式、または符号化された形式のいずれかを使用することを選択できます。「関数名に `SO` 名を付加」チェックボックスをオンにすると、その関数またはメソッドを含んでいる共有オブジェクトの名前が関数名またはメソッド名に付加されます。

また、「書式」タブでは、ユーザー、上級、マシンのいずれかの表示モードも選択できます。「表示モード」の設定は、Java 実験と OpenMP 実験の処理を制御します。

Java 実験の場合:

- 「ユーザー」モードでは、Java スレッドの Java 呼び出しスタックが表示されますが、ハウスキーピングスレッドは表示されません。
- 「上級」モードでは、ユーザーの Java コードの実行中には Java スレッドの Java 呼び出しスタックが表示され、JVM コードの実行中または JVM ソフトウェアが Java 呼び出しスタックを報告しないときにはネイティブな呼び出しスタックが表示されます。このモードでは、ハウスキーピングスレッドのネイティブな呼び出しスタックが表示されます。
- 「マシン」モードでは、すべてのスレッドのネイティブな呼び出しスタックが表示されます。

OpenMP 実験の場合:

- 「ユーザー」モードおよび「上級」モードでは、マスタースレッド呼び出しスタックとスレーブスレッド呼び出しスタックが調整されて表示されます。また、OpenMP ランタイムが特定の操作を実行しているときは、`<OMP-*>` という形式の名前を持つ特殊関数が追加されます。
- 「マシン」モードでは、すべてのスレッドのネイティブな呼び出しスタックと、コンパイラによって生成されたアウトライン関数が表示されます。

それ以外のすべての実験では、3 つのモードのすべてで同じデータが表示されます。

「タイムライン」タブ

「タイムライン」タブでは、表示するイベント固有のデータの種類を選択したり、スレッド、LWP、または CPU に関するイベント固有のデータを表示したり、ルートまたはリーフでの呼び出しスタックの表示の配置を選択したり、表示する呼び出しスタックのレベル数を選択することができます。

「検索パス」タブ

「パスを検索」タブでは、ソースおよびオブジェクトファイルを検索するためのディレクトリリストを管理できます。特別な名前「`$expts`」は読み込まれている実験を参照しており、ほかのすべての名前はファイルシステム内のパスを示しているはずです。

「パスマップ」タブ

「パスマップ」タブでは、ファイルパスの先頭部分を別の位置にマップできます。接頭辞の組のセット、つまり元の接頭辞と新しい接頭辞を設定できます。そのあと、パスは元の接頭辞から新しい接頭辞にマップされます。パスマップは複数指定でき、ファイルの検索時にはそれぞれが順番に試されます。

「タブ」タブ

「データ表示方法の設定」ダイアログボックスの「タブ」タブを使用すると、「アナライザ」ウィンドウに表示するタブを選択できます。

「タブ」タブには、現在の実験に適用できるタブが一覧表示されます。標準のタブは左の列に表示されます。インデックスタブが中央の列に表示され、定義されたメモリータブが右の列に表示されます。

左の列で各チェックボックスをクリックし、標準のタブを表示用に選択または選択解除します。

中央の列で各チェックボックスをクリックし、インデックスタブを表示用に選択または選択解除します。事前定義されているインデックスタブは、スレッド、CPU、標本、および秒です。別のインデックスオブジェクト用のタブを追加するには、「「カスタムインデックス」タブを追加」ボタンをクリックして「インデックスオブジェクトを追加」ダイアログボックスを開きます。「オブジェクト名」テキストボックスに、新しいオブジェクトの名前を入力します。「式」テキストボックスに、記録された物理アドレスまたは仮想アドレスをオブジェクトインデックスへマップするために使用するインデックス式を入力します。インデックス式の規則については、139 ページの「`indxobj_define` *indxobj_type index_exp*」 を参照してください。

右の列で各チェックボックスをクリックし、メモリーオブジェクトタブを表示用に選択または選択解除します。カスタムオブジェクトを追加するには、「カスタムオブジェクトを追加」ボタンをクリックして「メモリーオブジェクトを追加」ダイアログボックスを開きます。「オブジェクト名」テキストボックスに、新しいカスタムメモリーオブジェクトの名前を入力します。「式」テキストボックスに、記録された物理アドレスまたは仮想アドレスをオブジェクトインデックスへマップするために使用するインデックス式を入力します。インデックス式の規則については、137 ページの「`mobj_define` *mobj_type index_exp*」 を参照してください。

カスタムインデックスオブジェクトまたはメモリーオブジェクトを追加したときに、そのオブジェクトのチェックボックスが「タブ」タブに追加され、デフォルトで選択されます。

テキストとデータの検索

アナライザのツールバーには「検索」ツールが組み込まれており、検索ターゲット用にドロップダウンリストで指定する 2 つのオプションがあります。「関数」タブや「呼び出し元 - 呼び出し先」タブの「名前」列のテキストや、「ソース」タブや「逆アセンブリ」タブの「コード」列のテキストを検索できます。また、「ソース」タブや「逆アセンブリ」タブの高メトリック項目を検索できます。高メトリック項目を含む行のメトリック値は、緑色に強調表示されます。「検索」フィールドの隣りの矢印ボタンを使用すると、上または下に検索できます。

関数の表示と非表示

デフォルトでは、「関数」タブと「呼び出し元 - 呼び出し先」タブに各ロードオブジェクトのすべての関数が表示されます。「関数の表示/非表示/API のみ」ダイアログボックスを使用して、ロードオブジェクト内のすべての関数を非表示にしたり、ロードオブジェクトに対する API を表す関数のみを表示することができます。詳細はオンラインヘルプを参照してください。

ロードオブジェクト内の関数が非表示の場合、「関数」タブと「呼び出し元 - 呼び出し先」タブには、ロードオブジェクトのすべての関数の集合体を表す 1 つのエントリが表示されます。同様に、「行」タブと「PC」タブには、ロードオブジェクトのすべての関数のすべての PC の集合体を表す 1 つのエントリが表示されます。

ロードオブジェクト内の API 関数のみが表示される場合、ライブラリへの呼び出しを表す関数のみが表示されます。それらの関数の下のすべての呼び出しは、そのロードオブジェクト内の呼び出しであるか、ほかのロードオブジェクトに対する呼び出しであるかにかかわらず、コールバックを含めて表示されません。「呼び出し元 - 呼び出し先」タブには、このような関数の呼び出し先は表示されません。

ロードオブジェクトの設定は、`.er.rc` ファイル内でコマンドを使用して事前に設定できます。

フィルタリングとは対照的に、非表示となっている関数に対応するメトリックスは、すべての表示で何らかの形で示されます。

データのフィルタリング

デフォルトでは、各タブのデータは、すべての実験、すべての標本、すべてのスレッド、すべての LWP、およびすべての CPU について表示されます。「データをフィルタ」ダイアログボックスを使用して、データのサブセットを選択できます。

「データをフィルタ」ダイアログボックスには、「基本」タブと「詳細」タブがあります。

「データをフィルタ」ダイアログボックスの使用法については、オンラインヘルプを参照してください。

ここで説明されているフィルタは、110 ページの「「MPI タイムラインコントロール」タブ」および110 ページの「「MPI グラフコントロール」タブ」で説明されている MPI フィルタリングとは関係ありません。これらのフィルタは、「MPI グラフ」タブに影響しません。

「基本」タブ

「基本」タブでは、フィルタするデータの実験を選択できます。その後、メトリックスを表示する標本、スレッド、LWP、および CPU を指定できます。実験をクリックするか、「すべて選択」ボタン、「すべて消去」ボタン、または「反転」ボタンを使用して、実験の一覧から 1 つ以上の実験を選択できます。その後テキストボックスを使用して、それらの実験に対して表示されるデータを変更できます。3 つのフィルタをすべて同時に適用することはできますが、複数の CPU、スレッド、および LWP によってフィルタリングされたデータを解析するときは注意してください。「すべて有効」ボタン、「選択的に有効」ボタン、「すべて無効」ボタン、および「選択的に無効」ボタンを使用して、実験のデータ表示を有効または無効にします。

実験の選択

アナライザでは、複数の実験が読み込まれているときに、実験でフィルタリングすることができます。実験は個々に読み込むことも、実験グループを名前で指定することもできます。

標本の選択

標本には 1 - *N* の番号が付けられ、任意の標本セットを選択できます。選択する標本は、標本番号をコンマで区切ったリストで指定するか、または 1–5 のように範囲を指定します。

スレッドの選択

スレッドには 1 - *N* の番号が付けられ、任意のスレッドセットを選択できます。選択するスレッドは、スレッド番号をコンマで区切ったリストで指定するか、または範囲を指定します。スレッドのプロファイルデータは、LWP 上でスレッドが実際にスケジュールされている実行部分のみをカバーします。

LWP の選択

LWP には 1 - *N* の番号が付けられ、任意の LWP セットを選択できます。選択する LWP は、LWP 番号をコンマで区切ったリストで指定するか、または範囲を指定します。同期データが記録されている場合は、報告される LWP は、同期イベントの入口の LWP になり、これは同期イベントの出口の LWP とは異なる場合があります。

Linux システムでは、スレッドと LWP は同義です。

CPU の選択

CPU 情報が記録されている場合 (Solaris OS) は、任意の CPU セットを選択できます。選択する CPU は、CPU 番号をコンマで区切ったリストで指定するか、または範囲で指定します。

「詳細」タブ

「詳細」タブではフィルタ式を指定でき、そのフィルタ式が真と評価されたデータレコードは、表示に組み込まれます。フィルタ式で使用する文法については、154 ページの「式の文法」を参照してください。

「詳細」タブを表示するには、ツールバーの右端のボタンをクリックするか、「フィルタ」ダイアログボックスが開いている場合は、その「詳細」タブをクリックします。

「詳細」タブは、ヘッダーとフィルタ指定のテキストボックスで構成されます。ヘッダーには、フィルタ句を入力するための読み取り専用のテキストフィールドと、AND で追加するボタン、OR で追加するボタン、およびその句にフィルタを設定するボタンがあります。このフィールドの内容が読み込まれて、「関数」タブ、「データオブジェクト」タブ、「データレイアウト」タブ、またはいずれかのメモリーオブジェクトタブでの単一選択または複数選択が反映されます。いずれかのボタンをクリックすると、選択内容が句に変換されて、その句がフィルタの指定に追加されるか、その句でフィルタの指定が置き換えられます。

フィルタを合成した場合は、フィルタ指定フィールドにテキストを入力するか、句を追加し、「OK」または「適用」をクリックしてフィルタを設定します。

正しくないフィルタが指定された場合は、エラーが表示され、古いフィルタ設定のままになります。

アナライザからの実験の記録

ターゲット名とターゲット引数を指定してアナライザを起動すると、アナライザによって Oracle Solaris Studio の「パフォーマンスコレクタ」ウィンドウが開かれます。このウィンドウでは、指定したターゲットに実験を記録できます。引数を指定せずに、または実験リストを指定してアナライザを起動した場合、新規の実験を記録するには、「ファイル」→「実験を収集」を選択して「パフォーマンスコレクタ」ウィンドウを開きます。

「収集」ウィンドウの「実験を収集」タブには、ターゲットとその引数、および実験の実行に使用する各種パラメータを指定できるパネルがあります。パネル内のオプションは、第 3 章「パフォーマンスデータの収集」で説明されている `collect` コマンドで使用できるオプションに対応します。

このパネルのすぐ下には、「プレビューコマンド」ボタンとテキストフィールドがあります。このボタンをクリックすると、テキストフィールドに、「実行」ボタンをクリックしたときに使用される `collect` コマンドが取り込まれます。

「収集するデータ」タブでは、収集するデータの種類を選択できます。

「入力/出力」タブには 2 つのパネルがあります。コレクタ自体からの出力を受け取るパネルと、プロセスからの出力を受け取るパネルです。

一連のボタンを使って、次の操作を実行できます。

- 実験を実行する
- 実行を終了する
- 実行中に「一時停止」、「再開」、および「標本」シグナルをプロセスに送信する (対応するシグナルが指定されている場合に有効)
- ウィンドウを閉じる

実験の進行中にウィンドウを閉じても、実験は続行されます。ウィンドウを再度開くと、実行中にパネルが開いたままであったかのように、実行中の実験が表示されます。実験の実行中にアナライザを終了しようとすると、実行を終了するか継続するかを確認するダイアログボックスが表示されます。

アナライザのデフォルト設定

アナライザのデフォルト設定は、.er.rc デフォルト値ファイルで制御されます。アナライザは、複数のこれらのファイルからの指示を次の順序で処理します。

- インストールされている Oracle Solaris Studio ソフトウェアの lib ディレクトリ内の er.rc ファイル。たとえば、デフォルト Solaris インストールの場合、このファイルは /opt/solstudio12.2/lib/er.rc にあります。
- ホームディレクトリ内の .er.rc ファイル (ある場合)
- 現在のディレクトリ内の .er.rc ファイル (ある場合)

最後に処理された設定が優先されます。システム全体の .er.rc 設定よりホームディレクトリ内の .er.rc 設定の方が優先され、この設定より現在のディレクトリ内の .er.rc 設定の方が優先されます。

パフォーマンスアナライザで .er.rc ファイルを作成および更新するには、「表示」メニューから開くことができる「データ表示方法の設定」ダイアログボックスで「保存」ボタンをクリックします。「データ表示方法の設定」ダイアログボックスから .er.rc ファイルを保存すると、それ以後のアナライザの呼び出しに影響が出るだけでなく、er_print ユーティリティーと er_src ユーティリティーにも影響が及びます。

.er.rc ファイルの設定

.er.rc ファイルには、次の設定を含めることができます。

- アナライザに実験を読み込んだときに表示可能とするタブの指定。アナライザのタブ名は、「実験」タブと「タイムライン」タブを除く、対応するレポートの er_print コマンドの結果と一致します。
- カスタム MemoryObjects と IndexObjects の定義。
- メトリックス、ソート、およびコンパイラ注釈オプションの指定用のデフォルト設定値。
- ソースと逆アセンブリ出力でのメトリックスの強調表示用のしきい値。
- 「タイムライン」タブ、名前の書式指定、および表示モードの設定のデフォルト設定値。
- ソースファイルとオブジェクトファイルの検索パスまたはパスマップの指定。
- ロードオブジェクトの関数の表示と非表示。
- 初期実験が読み込まれるときに派生実験を読み込むかどうかの指定。en_desc の設定は、on、off、または *=regexp* のいずれかです。on を指定した場合、すべての派生の読み取りと読み込みを行い、off を指定した場合は、派生の読み取りと読み込みを行いません。=regexp を指定した場合は、系統または実行可能ファイルの

名前が、指定した正規表現と一致する派生の読み取りと読み込みを行います。デフォルトでは、en_desc は on なので、すべての派生が読み込まれます。

.er.rc ファイル内で使用できるコマンドについての詳細は、149 ページの「デフォルト値を設定するコマンド」と151 ページの「パフォーマンスアナライザにのみデフォルト値を設定するコマンド」を参照してください。

実験の比較

アナライザに複数の実験または実験グループを読み込むことができます。デフォルトでは、同じ実行可能ファイル上での複数の実験が読み込まれると、データは集計され、1 つの実験であるかのように表示されます。このデータを別々に表示して、実験データを比較することもできます。

アナライザで 2 つの実験を比較するには、最初の実験を通常どおりに開いてから「ファイル」>「実験を追加」を選択して、2 つ目の実験を読み込みます。これらを比較するには、比較をサポートするタブを右クリックし、「実験を比較」を選択します。

実験の比較をサポートするタブは、「関数」、「呼び出し元 - 呼び出し先」、「ソース」、「逆アセンブリ」、「行」、および「PC」です。比較モードでは、これらのタブ上において、実験または実験グループのデータは隣接する列に表示されます。列は、実験または実験グループの読み込み順に表示され、追加のヘッダー行に実験名または実験グループ名が示されます。

比較モードの有効化

.er.rc ファイルで compare on と設定すると、デフォルトで比較モードが有効になります。または、アナライザの「データ表示方法の設定」ダイアログボックスで、「書式」タブの「実験を比較」オプションを選択することにより、比較モードを有効にできます。

er_print compare コマンドを使って、実験を比較することもできます。詳細は、148 ページの「compare { on | off }」を参照してください。

◆ ◆ ◆ 第 5 章

er_print コマンド行パフォーマンス解析ツール

この章では、er_print ユーティリティーを使用してパフォーマンス解析を行う方法を説明します。er_print ユーティリティーは、パフォーマンスアナライザがサポートする各種の表示内容を ASCII 形式で出力します。これらの情報は、ファイルにリダイレクトしないかぎり、標準出力に書き込まれます。er_print ユーティリティーには、引数として、コレクタが生成した実験名または実験グループ名を指定する必要があります。

er_print ユーティリティーを使用して、関数や呼び出し元と呼び出し先のパフォーマンスメトリックス、ソースコードと逆アセンブリコードのリスト、標本収集情報、データ空間データ、スレッド解析データ、および実行統計情報を表示することができます。

er_print を、複数の実験または実験グループで起動した場合、デフォルトでは、実験データが集計されますが、実験を比較することもできます。詳細は、148 ページの「compare { on | off }」を参照してください。

この章では、次の内容について説明します。

コレクタが収集するデータについては、第 2 章「パフォーマンスデータ」を参照してください。

パフォーマンスアナライザを使用して情報をグラフィカル形式で表示する方法については、第 4 章「パフォーマンスアナライザツール」およびオンラインヘルプを参照してください。

er_print の構文

er_print ユーティリティーのコマンド行構文は、次のとおりです。

```
er_print [ -script script | -command | - | -V ] experiment-list
```

er_print ユーティリティーのオプションは、次のとおりです。

-	キーボードから入力された er_print コマンドを読み取ります。
-script *script*	*script* ファイルからコマンドを読み取ります。script ファイルは、一連の er_print コマンドを含むファイルで、1 行に 1 つの er_print コマンドがあります。-script オプションを指定しない場合、er_print は、端末またはコマンド行からコマンドを読み取ります。
-*command* [*argument*]	指定されたコマンドを処理します。
-V	バージョン情報を表示して終了します。

er_print のコマンド行には、複数のオプションを指定できます。指定したオプションは、指定した順に処理されます。スクリプト、ハイフン、明示的なコマンドを任意の順序で組み合わせることができます。コマンドまたはスクリプトを何も指定しなかった場合、er_print はデフォルトで対話モードになり、キーボードからコマンドを入力します。対話モードを終了するには、quit と入力するか、Ctrl-D を押します。

それぞれのコマンドが処理されたあと、その処理から発生したエラーメッセージまたは警告メッセージがある場合は、出力されます。処理に関する統計情報の概要を出力するには、procstats コマンドを使用します。

er_print ユーティリティーで使用できるコマンドについては、以降の節で示します。

すべてのコマンドは、そのコマンドであることが明確なかぎり、短縮することができます。コマンドを複数の行に分割するには、行の末尾にバックスラッシュ \ を付けます。\ で終わる行は、その行の構文解析が行われる前に \ 文字が削除され、次の行の内容が追加されます。1 つのコマンドに使用できる行数は、利用可能なメモリー以外に制限はありません。

空白文字を含んでいる引数は、二重引用符で囲む必要があります。引用符の内部では、行をまたいでテキストを分割してもかまいません。

メトリックリスト

多くの er_print コマンドでは、メトリックキーワードのリストを使用します。リストの構文は次のとおりです。

metric-keyword-1[:*metric-keyword2*...]

測定されたデータに基づく動的メトリックスの場合、メトリックのキーワードは、メトリックフレーバー文字列、メトリック表示形式文字列、およびメトリック名文字列の 3 つの部分から構成されます。これらは、空白を入力せずに次のように続けて指定します。

flavorvisibilityname

実験内のロードオブジェクトの静的プロパティー (名前、アドレス、およびサイズ) に基づく静的メトリックスの場合、メトリックのキーワードは、メトリック名とその前に付加されるメトリック表示形式文字列 (省略可能) を空白なしに結合して構成されます。

[*visibility*]*name*

メトリックの *flavor* 文字列と *visibility* 文字列は、フレーバー文字と表示形式文字を使用して指定します。

指定可能なメトリックフレーバー文字を表 5–1 に示します。複数のフレーバー文字を含むメトリックキーワードが展開されて、メトリックキーワードリストになります。たとえば、ie.user は、展開されて i.user:e.user になります。

表 5–1 メトリックフレーバー文字

文字	内容の説明
e	排他的メトリック値を表示します。
i	包括的メトリック値を表示します。
a	属性メトリック値を表示します (呼び出し元 - 呼び出し先メトリックの場合のみ)。
d	データ空間メトリック値を表示します (データ派生メトリックスの場合のみ)。

指定可能なメトリック表示形式文字を表 5–2 に示します。指定可能なメトリック表示形式文字をTable 5–2 に示します。表示形式文字列の文字の順序は重要ではありません。対応するメトリックスの表示順序には影響しません。たとえば、`i%.user` と `i.%user` はともに `i.user:i%user` と解釈されます。

表示形式だけが異なるメトリックスは、常に標準の順序で一緒に表示されます。表示形式だけが異なる 2 つのメトリックキーワードがほかのキーワードで区切られている場合は、標準の順序で 2 つのメトリックスの 1 つ目の位置にメトリックスが表示されます。

表 5–2 メトリック表示形式文字

文字	内容の説明
.	メトリックを時間で表示します。この指定は、サイクルカウントを計測する時間メトリックスとハードウェアカウンタメトリックスに適用されます。そのほかのメトリックスは「+」と解釈されます。
%	プログラム全体のメトリックを百分率で表示します。呼び出し元 - 呼び出し先リストの属性メトリックスの場合は、選択した関数の包括的メトリックにの割合が表示されます。
+	メトリックを絶対値で表示します。ハードウェアカウンタの場合、この値はイベントの回数です。時間メトリックスの場合は「.」と解釈されます。
!	メトリック値を表示しません。ほかの表示形式文字と組み合わせることはできません。

フレーバー文字列と可視文字列のそれぞれが複数の文字から構成されている場合は、フレーバー文字列が先に展開されます。すなわち、`ie.%user` は展開されて `i.%user:e.%user` になり、`i.user:i%user:e.user:e%user` と解釈されます。

静的メトリックスの場合、ソート順序の定義という観点からは、表示形式文字のピリオド (.)、正符号 (+)、パーセント記号 (%) は同等と見なされます。つまり、`sort i%user`、`sort i.user`、`sort i+user` はすべて、「どのような形式で表示するにせ

よ、包括的ユーザー CPU 時間を基準にソートする」ことを意味します。また、sort i!user は、「表示するかどうかに関係なく、包括的ユーザー CPU 時間を基準にソートする」という意味になります。

可視文字の感嘆符 (!) を使用すると、各フレーバーのメトリックに組み込みのデフォルト値を置き換えられます。

メトリックリスト内で同じメトリックを複数回表示した場合は、最初に出現したものだけが処理され、それ以後のものは無視されます。名前付きメトリックがリストにない場合は、そのメトリックがリストに付加されます。

表 5–3 に、時間メトリックス、同期遅延メトリックス、メモリー割り当てメトリックス、MPI トレースメトリックス、および 2 つの一般的なハードウェアカウンタメトリックスに指定可能な er_print メトリック名文字列を示します。ほかのハードウェアカウンタメトリックスの場合、メトリック名文字列はカウンタ名と同じです。読み込まれた実験に適用できるすべてのメトリック名文字列のリストは、metric_list コマンドで取得できます。カウンタ名は、collect コマンドを引数なしで使用することによって一覧表示できます。ハードウェアカウンタについての詳細は、27 ページの「ハードウェアカウンタオーバーフローのプロファイルデータ」を参照してください。

表 5–3　メトリック名文字列

カテゴリ	文字列	内容の説明
時間メトリックス	user	ユーザー CPU 時間
	wall	実経過時間
	total	LWP 合計時間
	system	システム CPU 時間
	wait	CPU 待ち時間
	ulock	ユーザーロック時間
	text	テキストページフォルト時間
	data	データページフォルト時間
	owait	ほかの待ち時間
時間ベースのプロファイルメトリックス	mpiwork	MPI ランタイムで作業する (処理の要求やメッセージなど) 時間
	mpiwait	MPI ランタイムで、イベント、バッファー、またはメッセージを待っている時間
	ompwork	作業を直列または並列で実行する時間

表 5–3 メトリック名文字列 (続き)

カテゴリ	文字列	内容の説明
	ompwait	OpenMP ランタイムが同期を待っている時間
同期遅延メトリックス	sync	同期待ち時間
	syncn	同期待ち回数
MPI トレースメトリックス	mpitime	MPI 呼び出しに費やされた時間
	mpisend	開始された MPI ポイントツーポイント送信数
	mpibytessent	MPI で送信されるバイト数
	mpireceive	完了した MPI ポイントツーポイント受信数
	mpibytesrecv	MPI で受信されるバイト数
	mpiother	その他の MPI 関数の呼び出し数
メモリー割り当てメトリックス	alloc	割り当て数
	balloc	割り当てバイト数
	leak	リーク数
	bleak	リークバイト数
ハードウェアカウンタオーバーフローのメトリックス	cycles	CPU サイクル
	insts	発行された命令
スレッドアナライザのメトリックス	raccesses	データ競合のアクセス
	deadlocks	デッドロック

表 5–3 に示した名前文字列のほかに、2 つの名前文字列をデフォルトのメトリックスリスト内でのみ使用できます。この 2 つの文字列は、任意のハードウェアカウンタ名に一致する hwc と、任意のメトリック名文字列に一致する any です。また、cycles と insts は、SPARC プラットフォームと x86 プラットフォームに共通のものですが、それ以外のアーキテクチャー固有のフレーバーも存在します。使用可能なすべてのカウンタを一覧表示するには、引数を指定せずに collect コマンドを使用します。

読み込んだ実験から使用可能なメトリックスを確認するには、metric_list コマンドを使用します。

関数リストを制御するコマンド

ここでは、関数情報の表示を制御するコマンドを説明します。

functions

現在選択されているメトリックスとともに関数リストを書き込みます。関数リストには、関数を表示するために選択されたロードオブジェクトに含まれているすべての関数、および object_select コマンドで非表示にされた関数を持つロードオブジェクトが含まれます。

書き込む行数は、limit コマンドを使用して制限できます (146 ページの「出力を制御するコマンド」を参照)。

出力されるデフォルトのメトリックスは、排他的および包括的ユーザー CPU 時間で、秒数および全プログラムのメトリックの割合 (百分率) で示されます。表示する現行メトリックスを変更するには、metrics コマンドを使用します。これは、functions コマンドを発行する前に行う必要があります。また、.er.rc ファイル内の dmetrics コマンドを使用して、デフォルト値を変更することもできます。

Java プログラミング言語で書かれたアプリケーションの場合、表示される関数情報は表示モードがユーザー、上級、マシンのどれに設定されているかによって異なります。

- 「ユーザー」モードでは、各メソッドが名前によって示され、インタプリタされたメソッドと HotSpot でコンパイルされたメソッドのデータがまとめて集計されます。また、非ユーザー Java スレッドのデータは抑止されます。
- 「上級」モードでは、HotSpot でコンパイルされたメソッドがインタプリタされたメソッドから分離され、非ユーザー Java スレッドは抑止されません。
- 「マシン」モードでは、Java 仮想マシン (JVM) ソフトウェアと突き合わせてインタプリタされた Java メソッドのデータがインタプリタの進行と同時に表示される一方、指定されたメソッドについて、Java HotSpot 仮想マシンでコンパイルされたメソッドのデータが報告されます。すべてのスレッドが表示されます。

3 つのモードすべてにおいて、データは、Java ターゲットによって呼び出された C、C++、または Fortran コードの通常の方法で報告されます。

metrics *metric_spec*

関数リストに表示するメトリックスを指定します。*metric_spec* には、キーワードの default (デフォルトのメトリック選択を復元します) またはコロンで区切ったメトリックキーワードのリストを指定できます。次に、メトリックリストの指定例を示します。

```
% metrics i.user:i%user:e.user:e%user
```

このコマンドを入力すると、`er_print` ユーティリティーは次のメトリックスを表示します。

- 包括的ユーザー CPU 時間 (秒単位)
- 包括的ユーザー CPU 時間 (百分率)
- 排他的ユーザー CPU 時間 (秒単位)
- 排他的ユーザー CPU 時間 (百分率)

デフォルトでは、149 ページの「デフォルト値を設定するコマンド」の説明のように、`.er.rc` ファイルから処理された `dmetrics` コマンドに基づいたメトリック設定が使用されます。`metrics` コマンドで明示的に *metric_spec* を `default` に設定した場合は、記録するデータに適したデフォルト設定が復元されます。

メトリックスがリセットされると、新しいリスト内でデフォルトのソートメトリックが設定されます。

metric_spec を省略した場合は、現在のメトリックスの設定が表示されます。

`metrics` コマンドは、関数リスト用のメトリックスを設定するほか、呼び出し元 - 呼び出し先のメトリックスと、データ派生出力のメトリックスを同じ設定値に設定します。

`metrics` コマンドが処理されると、現在有効なメトリックを示すメッセージが表示されます。前述の例では、メッセージは次のようになります。

```
current: i.user:i%user:e.user:e%user:name
```

メトリックリストの構文については、123 ページの「メトリックリスト」を参照してください。指定可能なメトリックスを一覧表示するには、`metric_list` コマンドを使用します。

`metrics` コマンドに誤りがあった場合、そのコマンドは警告とともに無視され、以前の設定が引き続き有効になります。

`sort` *metric_spec*

関数リストを *metric_spec* でソートします。メトリック名の中の *visibility* は、ソート順序に影響を及ぼしません。複数のメトリックが *metric_spec* の中で指定されている場合は、表示可能な最初のものが使用されます。指定されたメトリックスに表示可能なものがない場合は、コマンドが無視されます。*metric_spec* の前に負符号 (-) を付加することにより、逆順のソートを指定できます。

デフォルトでは、149 ページの「デフォルト値を設定するコマンド」の説明のように、`.er.rc` ファイルから処理された `dsort` コマンドに基づいたメトリックソート設定が使用されます。`sort` コマンドで明示的に *metric_spec* を `default` に設定した場合は、デフォルトの設定が使用されます。

文字列 *metric_spec* は、123 ページの「メトリックリスト」に示すメトリックキーワードのいずれか 1 つです。

```
% sort i.user
```

このコマンドは、`er_print` ユーティリティーに、関数リストを包括的ユーザー CPU 時間によってソートするよう指示します。指定したメトリックが読み込まれた実験に含まれていない場合は、警告メッセージが表示され、コマンドは無視されます。コマンドが終了すると、ソート基準メトリックが表示されます。

`fsummary`

関数リスト内の各関数について、概要パネルを出力します。出力するパネル数は、`limit` コマンドを使用して制限できます (146 ページの「出力を制御するコマンド」を参照)。

概要メトリックスパネルには、関数またはロードオブジェクトの名前、アドレス、およびサイズのほか、関数についてはソースファイル、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、ならびに選択された関数やロードオブジェクトについて記録された排他的メトリックスと包括的メトリックスの値と百分率が表示されます。

`fsingle` *function_name* [*N*]

指定された関数の概要パネルを出力します。同じ名前を持つ関数が複数存在する場合には、省略可能なパラメータ *N* が必要です。指定の関数名を持つ *N* 番目の関数について、概要メトリックスパネルが出力されます。コマンド行でコマンドを入力する場合、*N* を必ず指定する必要があります。不要な場合は無視されます。*N* が必要であるときに *N* を使用しないでコマンドを対話的に入力すると、対応する *N* 値を持つ関数のリストが出力されます。

関数の概要メトリックスについては、`fsummary` コマンドの解説を参照してください。

呼び出し元 - 呼び出し先リストを管理するコマンド

ここでは、呼び出し元と呼び出し先の情報の表示を制御するコマンドを説明します。

`callers-callees`

それぞれの関数の呼び出し元 - 呼び出し先パネルを、関数ソートメトリック (`sort`) で指定された順序で出力します。

各呼び出し元 - 呼び出し先レポート内では、呼び出し元 - 呼び出し先のソートメトリックス (csort) に従って呼び出し元と呼び出し先がソートされます。出力するパネル数は、limit コマンドを使用して制限できます (146 ページの「出力を制御するコマンド」を参照)。選択されている関数 (中央の関数) は、次のようにアスタリスクで示されます。

```
Attr.     Excl.     Incl.      Name
User CPU  User CPU  User CPU
 sec.      sec.       sec.
4.440     0.        42.910     commandline
0.        0.         4.440    *gpf
4.080     0.         4.080     gpf_b
0.360     0.         0.360     gpf_a
```

この例では、関数 gpf が選択されています。この関数は commandline によって呼び出され、gpf_a と gpf_b を呼び出します。

csingle *function_name* **[***N***]**

指定された関数の呼び出し元 - 呼び出し先パネルを出力します。同じ名前を持つ関数が複数存在する場合には、省略可能なパラメータ *N* が必要です。指定の関数名を持つ *N* 番目の関数について、呼び出し元 - 呼び出し先パネルが出力されます。コマンド行でコマンドを入力する場合、*N* を必ず指定する必要があります。不要な場合は無視されます。*N* が必要であるときに *N* を使用しないでコマンドを対話的に入力すると、対応する *N* 値を持つ関数のリストが出力されます。

cprepend *function-name [N | ADDR]*

呼び出しスタックを構築する際、現在の呼び出しスタックフラグメントの先頭に名前付き関数を追加します。関数名があいまいな場合、オプションパラメータが必要になります。パラメータの指定方法についての詳細は、132 ページの「source|src { *filename* | *function_name* } [*N*]」を参照してください。

cappend *function-name [N | ADDR]*

呼び出しスタックを構築する際、現在の呼び出しスタックフラグメントの末尾に名前付き関数を追加します。関数名があいまいな場合、オプションパラメータが必要になります。パラメータの指定方法についての詳細は、132 ページの「source|src { *filename* | *function_name* } [*N*]」を参照してください。

crmfirst

呼び出しスタックを構築する際、呼び出しスタックセグメントから最初のフレームを削除します。

crmlast

呼び出しスタックを構築する際、呼び出しスタックセグメントから最後のフレームを削除します。

呼び出しツリーリストを制御するコマンド

ここでは、呼び出しツリー用のコマンドについて説明します。

calltree

実験から動的コールグラフを表示します。各レベルに履歴メトリックスが表示されます。

リークリストと割り当てリストを管理するコマンド

ここでは、メモリーの割り当てと割り当て解除に関するコマンドについて説明します。

leaks

共通呼び出しスタックによって集計されたメモリーリークのリストを表示します。各エントリは、リーク総数、および指定の呼び出しスタックでリークした総バイト数を示します。このリストは、リークしたバイト数を基準としてソートされます。

allocs

共通呼び出しスタックによって集計されたメモリー割り当てのリストを表示します。各エントリは、割り当ての数、および指定の呼び出しスタックに割り当てられた総バイト数を示します。このリストは、割り当てられたバイト数を基準としてソートされます。

ソースリストと逆アセンブリリストを管理するコマンド

ここでは、注釈付きソースおよび逆アセンブリコードの表示を制御するコマンドを説明します。

pcs

現在のソートメトリックで整列されたプログラムカウンタ (Program Counter、PC) と、そのメトリックスのリストを出力します。このリストには、`object_select` コマンドで関数を非表示にした各ロードオブジェクトのメトリックスを集計した行が含まれています。

psummary

PC リスト内の各 PC の概要メトリックスパネルを現在のソートメトリックスで指定された順序で出力します。

lines

現在のソートメトリックで整列されたソース行と、そのメトリックスのリストを出力します。このリストには、行番号情報を持っていない各関数またはソースファイルが未知である各関数のメトリックスを集計した行と、`object_select` コマンドで関数を非表示している各ロードオブジェクトのメトリックスを集計した行が含まれています。

lsummary

行リスト内の各行の概要メトリックスパネルを現在のソートメトリックで指定された順序で出力します。

source|src { *filename* | *function_name* } [*N*]

指定したファイル、または指定した関数を含むファイルの注釈付きソースコードを出力します。いずれの場合も、指定したファイルはパスの通っているディレクトリに存在する必要があります。ソースが GNU Fortran コンパイラでコンパイルされている場合は、ソースに表示されるように、関数名のあとにアンダースコアを 2 つ追加する必要があります。

省略可能なパラメータ *N* (正の整数) は、ファイル名または関数名があいまいでない場合にだけ使用します。このパラメータを指定した場合は、*N* 番目の候補が使用されます。番号指定のないあいまいな名前が指定された場合、*er_print* ユーティリ

ティーはオブジェクトファイル名の候補のリストを表示します。指定された名前が関数の場合は、その関数名がオブジェクトファイル名に追加され、そのオブジェクトファイルの N の値を表す番号も出力されます。

関数名は *function"file"* としても指定できます。この場合、*file* は、関数の代替ソースコンテキストを指定するために使用されます。最初の命令の直後にその関数のインデックス行が追加されます。インデックス行は、次の書式で山括弧内にテキストとして表示されます。

```
<Function: f_name>
```

関数のデフォルトソースコンテキストは、その関数の最初の命令が帰するソースファイルとして定義されます。これは通常、関数を含むオブジェクトモジュールを生成するためにコンパイルされたソースファイルです。代替ソースコンテキストは、関数に属する命令を含むほかのファイルから構成されます。このようなコンテキストには、インクルードファイルの命令と、指定の関数にインライン化された関数の命令が含まれます。代替のソースコンテキストが存在する場合、デフォルトのソースコンテキストの冒頭に、代替ソースコンテキストが置かれている場所を示す拡張インデックス行のリストを次の形式で組み込みます。

```
<Function: f, instructions from source file src.h>
```

注 – コマンド行から er_print ユーティリティーを起動するときに -source 引数を使用する場合は、file の引用符の前にバックスラッシュのエスケープ文字を付加する必要があります。つまり、関数名の形式は、function\"file\" となります。er_print ユーティリティーが対話モードにあるときは、バックスラッシュは不要です。使用しないでください。

通常、デフォルトのソースコンテキストが使用された場合は、そのファイルに入っているすべての関数についてメトリックスが表示されます。ファイルを明示的に指定した場合は、指定した関数についてのみ、メトリックスが表示されます。

disasm|dis { *filename* **|** *function_name* **} [** *N***]**

指定したファイル、または指定した関数を含むファイルの注釈付き逆アセンブリコードを出力します。指定したファイルは、パスの通っているディレクトリに存在する必要があります。

省略可能なパラメータ *N* の意味は、source コマンドと同じです。

scc *com_spec*

注釈付きソースのリストに含めるコンパイラのコメントクラスを指定します。クラスリストはコロンで区切ったクラスのリストであり、次のメッセージクラスがゼロ個以上含まれています。

表 5–4 コンパイルコメントメッセージクラス

クラス	意味
b[asic]	基本的なレベルのメッセージを表示します。
v[ersion]	ソースファイル名、最終修正日付、コンパイラコンポーネントのバージョン、コンパイル日付とオプションなどのバージョンメッセージを表示します。
pa[rallel]	並列化に関するメッセージを表示します。
q[uery]	最適化に影響するコードに関する問い合わせメッセージを表示します。
l[oop]	ループの最適化と変換に関するメッセージを表示します。
pi[pe]	ループのパイプライン化に関するメッセージを表示します。
i[nline]	関数のインライン化に関するメッセージを表示します。
m[emops]	ロード、ストア、プリフェッチなどのメモリー操作に関するメッセージを表示します。
f[e]	フロントエンドのメッセージを表示します。
co[degen]	コードジェネレータのメッセージを表示します。
cf	ソースの下部にコンパイラのフラグを表示します。
all	すべてのメッセージを表示します。
none	メッセージを表示しません。

`all` および `none` クラスは常に単独で指定します。

`scc` コマンドを省略した場合は、`basic` がデフォルトのクラスになります。*class-list* が空の状態で `scc` コマンドを入力した場合、コンパイラのコメントは出力されません。通常、`scc` コマンドは、`.er.rc` ファイルでのみ使用します。

sthresh *value*

注釈付きソースコードでのメトリックスの強調表示に使用するしきい値の百分率を指定します。ファイル内のソース行で、メトリック値が、そのメトリックの最大値の *value* % 以上である場合、そのメトリックが発生する行の先頭に ## が挿入されます。

dcc *com_spec*

注釈付き逆アセンブリリストに含めるコンパイラのコメントクラスを指定します。クラスリストは、コロンで区切られたクラスのリストです。利用可能なクラスのリストは、表 5–4に示す注釈付きソースコードリストのクラスリストと同じです。クラスリストには、次のオプションを追加できます。

表 5–5 dcc コマンドの追加オプション

オプション	意味
h[ex]	命令の 16 進値を示します。
noh[ex]	命令の 16 進値を示しません。
s[rc]	ソースリストと注釈付き逆アセンブリリストをインタリーブします。
nos[rc]	ソースリストと注釈付き逆アセンブリリストをインタリーブしません。
as[rc]	注釈付きソースコードと注釈付き逆アセンブリリストをインタリーブします。

dthresh *value*

注釈付き逆アセンブリコードでのメトリックスの強調表示に使用するしきい値の百分率を指定します。ファイル内の命令行で、メトリック値が、そのメトリックの最大値の *value* % 以上である場合、そのメトリックが発生する行の先頭に ## が挿入されます。

cc *com_spec*

注釈付きのソースと逆アセンブリリストに含めるコンパイラのコメントクラスを指定します。クラスリストは、コロンで区切られたクラスのリストです。利用可能なクラスのリストは、表 5–4 に示す注釈付きソースコードリストのクラスリストと同じです。

setpath *path_list*

ソースファイルやオブジェクトファイルの検索に使用するパスを設定します。*path_list* は、コロンで区切られたディレクトリのリストです。ディレクトリ名にコロン文字がある場合は、バックスラッシュでコロンをエスケープします。特別なディレクトリ名 $expts は、現在の実験を読み込まれた順序で示します。これは、$ の 1 文字に短縮できます。

デフォルトの設定は、$expts:.. です。現在のパス設定の検索時にファイルが見つからなかった場合は、コンパイルされているフルパス名が使用されます。

引数のない `setpath` は、現在のパスを出力します。

addpath *path_list*

現在の `setpath` の設定に *path_list* を付加します。

pathmap *old-prefix new-prefix*

`addpath` または `setpath` で設定された *path_list* を使用してファイルが見つからなかった場合、`pathmap` コマンドを使用して、1 つまたは複数のパスの再マッピングを指定できます。ソースファイル、オブジェクトファイル、または共有オブジェクトのパス名が *old-prefix* で指定した接頭辞で始まる場合、古い接頭辞は *new-prefix* で指定した接頭辞に置き換えられます。結果のパスは、ファイルの検索に使用されます。複数の `pathmap` コマンドが提供されており、それぞれが、ファイルが見つかるまで試行されます。

ハードウェアカウンタデータ空間およびメモリーオブジェクトリストを制御するコマンド

ハードウェアカウンタプロファイリングおよびデータ空間プロファイリングにより収集される実験については、次のオブジェクトに関連するメトリックスを表示できます。

- データオブジェクト - ソースコードに記載されているプログラム定数、変数、配列、構造体や共用体などの集合体、個々の集合要素。
- メモリーオブジェクト - キャッシュ行、ページ、およびメモリーバンクなど、メモリーサブシステム内のコンポーネント。このオブジェクトは、記録された仮想アドレスまたは物理アドレスから計算されたインデックスから決定されます。メモリーオブジェクトは、仮想ページおよび物理ページについて 8K バイト、64K バイト、512K バイト、および 4M バイトのサイズで事前定義されています。ほかのオブジェクトも `mobj_define` コマンドにより定義できます。

このデータは、SPARC アーキテクチャーの `-xhwcprof` コンパイラオプションでコンパイルされた Solaris オブジェクトでのみ収集できます。

これらのデータの種類についての詳細は、27 ページの「ハードウェアカウンタオーバーフローのプロファイルデータ」を参照してください。ハードウェアカウンタオーバーフロープロファイリングの実行時に使用されるコマンド行については、63 ページの「`-h` *counter_definition_1*`...[,` *counter_definition_n*`]`」を参照してください。

`-xhwcprof` オプションについては、『Oracle Solaris Studio 12.2: Fortran ユーザーズガイド』、『Oracle Solaris Studio 12.2: C ユーザーガイド』、または『Oracle Solaris Studio 12.2: C++ ユーザーズガイド』を参照してください。

`data_objects`

データオブジェクトのリストを、それらのメトリックスとともに書き込みます。

`data_single` *name* [*N*]

指定されたデータオブジェクトの概要メトリックスパネルを書き込みます。オブジェクト名があいまいな場合には、省略可能なパラメータ N が必要です。指令がコマンド行にある場合には N は必要です。不要な場合は無視されます。

`data_layout`

データ派生メトリックデータを持つすべてのプログラミングデータオブジェクトについて、注釈付きのデータオブジェクトレイアウトを書き込みます。データは、各構造をひとまとめにして、現在のデータソートメトリック値によってソートされます。集合体データオブジェクトごとに、そのオブジェクトに加算される合計メトリックスが表示され、そのあとに、そのオブジェクトのすべての要素がオフセット順に表示されます。各要素には、そのメトリックスと、32 バイトブロックを基準にしたそのサイズと位置を示す情報が表示されます。

`memobj` *mobj_type*

所定のタイプのメモリーオブジェクトと現在のメトリックスのリストを出力します。メトリックスはデータ空間のリストとして、ソートされ表示されます。名前 *mobj_type* を直接、コマンドとして使用することもできます。

`mobj_list`

`memobj` コマンド内で *mobj_type* に使用する、既知のタイプのメモリーオブジェクトのリストを出力します。

`mobj_define` *mobj_type index_exp*

新しいタイプのメモリーオブジェクトを、*index_exp* で指定されたオブジェクトへの VA/PA のマッピングを使用して定義します。式の構文については、154 ページの「式の文法」に説明があります。

mobj_type は、定義済みであってはいけません。その名前は、すべて英数字または「_」文字で構成されている必要があり、1 文字目は英字である必要があります。

index_exp は、構文的に正しくなければいけません。構文的に正しくない場合はエラーが返され、定義は無視されます。

<Unknown> メモリーオブジェクトのインデックスは -1 です。また、新しいメモリーオブジェクトを定義するために使用する式は、<Unknown> の認識をサポートしている必要があります。たとえば、VADDR ベースのオブジェクトの場合、式は次の形式になっている必要があります。

```
VADDR>255?expression :-1
```

また、PADDR ベースのオブジェクトの場合、式は次の形式になっている必要があります。

```
PADDR>0?expression:-1
```

インデックスオブジェクトリストを制御するコマンド

インデックスオブジェクトのコマンドは、すべての実験に使用できます。インデックスオブジェクトリストは、記録されたデータからインデックスを計算できるオブジェクトのリストです。スレッド、CPU、標本、および秒のインデックスオブジェクトが事前に定義されています。その他のインデックスオブジェクトは、indxobj_define コマンドで定義できます。

次のコマンドは、インデックスオブジェクトのリストを制御します。

indxobj *indxobj_type*

所定のタイプのインデックスオブジェクトとそのメトリックスのリストを出力します。インデックスオブジェクトのメトリックスとソートは、排他的メトリックスだけが含まれる点を除き、関数リストと同じです。名前 *indxobj_type* を直接、コマンドとして使用することもできます。

indxobj_list

indxobj コマンド内で *indxobj_type* に使用する、既知のタイプのインデックスオブジェクトのリストを書き込みます。

indxobj_define *indxobj_type index_exp*

新しいタイプのインデックスオブジェクトを、*index_exp* で指定されたオブジェクトへのパケットのマッピングを使用して定義します。式の構文については、154 ページの「式の文法」に説明があります。

indxobj_type は、定義済みであってはいけません。その名前は、大文字と小文字が区別され、すべて英数字または「_」文字で構成されている必要があり、1 文字目は英字である必要があります。

index_exp の構文が正しくなかった場合は、エラーが返され、定義が無視されます。*index_exp* に空白文字が含まれる場合は、二重引用符 (") で囲む必要があります。

<Unknown> インデックスオブジェクトのインデックスは -1 です。また、新しいインデックスオブジェクトを定義するために使用する式は、<Unknown> の認識をサポートしている必要があります。

たとえば、仮想 PC また物理 PC に基づくインデックスオブジェクトの場合、式は次の形式になっている必要があります。

```
VIRTPC>0?VIRTPC:-1
```

OpenMP インデックスオブジェクトのコマンド

次のコマンドを使用して、OpenMP インデックスオブジェクトの情報を出力できます。

OMP_preg

実験で実行された OpenMP 並列領域とそれぞれのメトリックスのリストを出力します。このコマンドは、OpenMP 3.0 のパフォーマンスデータを使用した実験にのみ使用できます。

OMP_task

実験で実行された OpenMP タスクとそれぞれのメトリックスのリストを出力します。このコマンドは、OpenMP 3.0 のパフォーマンスデータを使用した実験にのみ使用できます。

スレッドアナライザ対応コマンド

次のコマンドはスレッドアナライザに対応しています。獲得されるデータや表示されるデータについての詳細は、『Oracle Solaris Studio 12.2: スレッドアナライザユーザーズガイド』を参照してください。

`races`

実験におけるすべてのデータ競合リストを書き出します。データ競合レポートは、データ競合検出データを使用した実験でのみ利用できます。

`rdetail` *race_id*

指定された *race_id* の詳細情報を書き出します。*race_id* を `all` に設定した場合は、すべてのデータ競合の詳細情報が表示されます。データ競合レポートは、データ競合検出データのある実験からのみ取得できます。

`deadlocks`

実験で検出されたすべての実際のデッドロックと潜在的なデッドロックのリストを書き出します。デッドロックレポートは、デッドロック検出データを使用した実験でのみ利用できます。

`ddetail` *deadlock_id*

指定された *deadlock_id* の詳細情報を書き出します。*deadlock_id* を `all` に設定した場合は、すべてのデッドロックの詳細情報が表示されます。デッドロックレポートは、デッドロック検出データを使用した実験でのみ利用できます。

実験、標本、スレッド、および **LWP** を一覧表示するコマンド

ここでは、実験、標本、スレッド、および LWP を一覧表示するコマンドについて説明します。

`experiment_list`

読み込まれている実験をそれぞれの ID 番号とともにすべて一覧表示します。各実験は、標本、スレッド、または LWP を選択する際に使用されるインデックス、および拡張フィルタリングに使用できる PID とともに一覧表示されます。

次に、実験リストの例を示します。

```
(er_print) experiment_list
ID Experiment
== ==========
1 test.1.er
2 test.6.er
```

sample_list

解析の対象として現在選択されている標本の一覧を表示します。

次に、標本リストの例を示します。

```
(er_print) sample_list
Exp Sel     Total
=== ======= =====
  1 1-6        31
  2 7-10,15    31
```

lwp_list

解析の対象として現在選択されている LWP の一覧を表示します。

thread_list

解析の対象として現在選択されているスレッドの一覧を表示します。

cpu_list

解析の対象として現在選択されている CPU の一覧を表示します。

実験データのフィルタリングを制御するコマンド

実験データのフィルタリングは、次の 2 つの方法で指定できます。

- フィルタ式を指定する。このフィルタ式は各データレコードごとに評価され、そのレコードを含めるかどうかを決定します。
- フィルタリング用の実験、標本、スレッド、CPU、および LWP を選択する。

フィルタ式の指定

フィルタ式は `filters` コマンドで指定できます。

filters *filter_exp*

filter_exp は式であり、この式が真と評価されたデータレコードは含まれ、偽と評価されたデータレコードは含まれません。式の文法については、154 ページの「式の文法」に説明があります。

フィルタ式のオペランドトークンの一覧表示

実験のフィルタ式で使用できるオペランドを一覧表示できます。

describe

フィルタ式の構築に使用できるトークンのリストを出力します。フィルタ式の一部のトークンと文法については、154 ページの「式の文法」を参照してください。

フィルタリング用の標本、スレッド、**LWP**、および **CPU** の選択

選択リスト

次に、選択の構文の例を示します。この構文はコマンドの説明で使用されます。

[*experiment-list*:]*selection-list*[+[
experiment-list:]*selection-list* ...]

各選択リストの前には、空白なしの 1 つのコロンで区切って実験リストを指定できます。選択リストを正符号 (+) で結合して、複数の選択を指定することもできます。

実験リストと選択リストの構文は同じで、次の例に示すように、キーワード `all`、または空白なしのコンマで区切った番号または番号の範囲 (*n*-*m*) のリストを指定できます。

```
2,4,9-11,23-32,38,40
```

実験番号は `experiment_list` コマンドを使用して特定できます。

次に選択の例をいくつか示します。

```
1:1-4+2:5,6
all:1,3-6
```

1 つ目の例では、実験 1 からオブジェクト 1 ～ 4、実験 2 からオブジェクト 5 ～ 6 を選択しています。2 つ目の例では、すべての実験からオブジェクト 1 と 3 ～ 6 を選択しています。オブジェクトは、LWP、スレッド、または標本のいずれかです。

選択用のコマンド

LWP、標本、CPU、およびスレッドを選択するためのコマンドは相互に依存しています。コマンドの実験リストの内容が、直前のコマンドのリストの内容と異なる場合は、最新のコマンドの実験リストの内容が、次のようにして 3 つのタイプの選択ターゲット (LWP、標本、スレッド) のすべてに適用されます。

- 最新の実験リストにない実験に対する既存の選択内容は無効になります。
- 最新の実験リストに含まれている実験に対する既存の選択内容は維持されます。
- 選択が行われていないターゲットに対しては `all` が適用されます。

`sample_select` *sample_spec*

情報を表示する標本を選択します。コマンドが終了すると、選択された標本が一覧表示されます。

`lwp_select` *lwp_spec*

情報を表示する LWP を選択します。コマンドが終了すると、選択された LWP が一覧表示されます。

`thread_select` *thread_spec*

情報を表示するスレッドを選択します。コマンドが終了すると、選択されたスレッドが一覧表示されます。

`cpu_select` *cpu_spec*

情報を表示する CPU を選択します。コマンドが終了すると、選択された CPU が一覧表示されます。

ロードオブジェクトの展開と短縮を制御するコマンド

これらのコマンドは、`er_print` ユーティティーによるロードオブジェクトの表示方法を決定します。

`object_list`

すべてのロードオブジェクトの状態と名前を示す 2 列のリストを表示します。最初の列には各ロードオブジェクトの表示/非表示/API の状態が示され、2 番目の列にはオブジェクトの名前が示されます。各ロードオブジェクトの名前の前には、そのオブジェクトの関数が関数リストに表示される (展開される) ことを示す `show`、そのオブジェクトの関数が関数リストに表示されない (短縮される) ことを示す `hide`、またはロードオブジェクトへのエントリポイントを表す関数のみが表示される場合は `API-only` が付きます。短縮されたロードオブジェクトのすべての関数は、そのロードオブジェクト全体を表す関数リスト内の単一の項目へマップされます。

ロードオブジェクトリストの表示例を次に示します。

```
(er_print) object_list
Sel  Load Object
==== ==================
hide <Unknown>
show <Freeway>
show <libCstd_isa.so.1>
show <libnsl.so.1>
show <libmp.so.2>
show <libc.so.1>
show <libICE.so.6>
show <libSM.so.6>
show <libm.so.1>
show <libCstd.so.1>
show <libX11.so.4>
show <libXext.so.0>
show <libCrun.so.1>
show <libXt.so.4>
show <libXm.so.4>
show <libsocket.so.1>
show <libgen.so.1>
show <libcollector.so>
show <libc_psr.so.1>
show <ld.so.1>
show <liblayout.so.1>
```

object_show *object1,object2,...*

すべての名前付きロードオブジェクトを、それらのすべての関数が表示されるように設定します。オブジェクトの名前は、フルパス名またはベース名で指定できます。名前にコンマ文字が含まれている場合は、その名前を二重引用符で囲む必要があります。ロードオブジェクトの名前に「all」という文字列が使用されている場合は、すべてのロードオブジェクトの関数が表示されます。

object_hide *object1,object2,...*

すべての名前付きロードオブジェクトを、それらのすべての関数が表示されないように設定します。オブジェクトの名前は、フルパス名またはベース名で指定できます。名前にコンマ文字が含まれている場合は、その名前を二重引用符で囲む必要があります。ロードオブジェクトの名前に「all」という文字列が使用されている場合は、すべてのロードオブジェクトの関数が表示されます。

object_api *object1,object2,...*

すべての名前付きロードオブジェクトを、ライブラリへのエントリポイントを表す関数のみがすべて表示されるように設定します。オブジェクトの名前は、フルパス

名またはベース名で指定できます。名前にコンマ文字が含まれている場合は、その名前を二重引用符で囲む必要があります。ロードオブジェクトの名前に「all」という文字列が使用されている場合は、すべてのロードオブジェクトの関数が表示されます。

objects_default

すべてのロードオブジェクトを、.er.rc に設定されている初期デフォルト値にしたがいます。

object_select *object1,object2,...*

関数の情報を表示するロードオブジェクトを選択します。すべての名前付きロードオブジェクトの関数が表示され、その他すべてのロードオブジェクトの関数は表示されません。*object-list* は、空白なしのコンマで区切ったロードオブジェクトのリストです。ロードオブジェクトの関数が表示される場合、ゼロではないメトリックスを保持するすべての関数が関数リストに表示されます。ロードオブジェクトの関数が表示されない場合、そのオブジェクトの関数は短縮され、そのロードオブジェクト全体のメトリックスが入った単一の行が表示され、個々の関数は表示されません。

ロードオブジェクト名は、フルパス名またはベース名で指定します。オブジェクト名そのものにコンマが含まれている場合は、名前を二重引用符で囲む必要があります。

メトリックスを一覧するコマンド

ここでは、現在選択されているメトリックスと使用可能なメトリックキーワードを一覧表示するコマンドを説明します。

metric_list

関数リストで現在選択されているメトリックスと、関数リスト内のさまざまな種類のメトリックスを参照する際にほかのコマンド (metrics、sort など) で使用可能なメトリックキーワードの一覧を表示します。

cmetric_list

現在選択されている呼び出し元 - 呼び出し先メトリックス、および呼び出し元 - 呼び出し先レポートのメトリックスとキーワード名のリストを表示します。metric_list 出力と同じ方法でリストを表示しますが、属性メトリックスも含みます。

data_metric_list

現在選択されているデータ派生メトリックス、およびすべてのデータ派生レポートのメトリックスとキーワード名のリストを表示します。metric_list コマンドの出力と同じ方法でリストを表示しますが、データ派生フレーバーを持つメトリックスと静的メトリックスだけを含めます。

indx_metric_list

現在選択されているインデックスオブジェクトメトリックス、およびすべてのインデックスオブジェクトレポートのメトリックスとキーワード名のリストを表示します。metric_list コマンドと同じ方法でリストを表示しますが、排他的フレーバーを持つメトリックスと静的メトリックスだけを含めます。

出力を制御するコマンド

ここでは、er_print の出力を制御するコマンドを説明します。

outfile { *filename* | - }

開いている出力ファイルを閉じ、以降の出力先として *filename* を開きます。*filename* を開くときに、既存のコンテンツを消去します。*filename* の代わりにハイフン (-) を指定した場合は、標準出力に出力されます。*filename* の代わりに 2 つのハイフン (--) を指定した場合は標準エラーへ出力されます。

appendfile *filename*

開いている出力ファイルを閉じ、*filename* を開きます。その際、既存のコンテンツがあれば残し、後続の出力がファイルの末尾に付加されるようにします。*filename* が存在しない場合、appendfile コマンドの機能は outfile コマンドと同じです。

limit *n*

出力をレポートの最初の *n* 個のエントリに制限します。*n* は符号なしの正の整数です。

name { long | short } [:{ *shared_object_name* **|** *no_shared_object_name* **}]**

長短どちらの形式の関数名を使用するかを指定します (C++ および Java のみ)。*shared_object_name* が指定された場合は、関数名に共有オブジェクト名を付加します。

viewmode { user| expert | machine }

モードを次のいずれかに設定します。

`user` (ユーザー)
: Java 実験の場合は、Java スレッドの Java 呼び出しスタックを表示し、ハウスキーピングスレッドを表示しません。関数リストには、Java 以外のスレッドからの集計時間を表す関数 `<JVM-System>` が含まれます。JVM ソフトウェアが Java 呼び出しスタックを報告しない場合、時間は関数 `<no Java callstack recorded>` に報告されます。

  OpenMP 実験の場合は、OpenMP を使用せずにプログラムがコンパイルされたときに取得されたものと同様の、再構築された呼び出しスタックが表示されます。また、OpenMP ランタイムが特定の操作を実行しているときは、`<OMP-*>` という形式の名前を持つ特殊関数を追加します。

`expert` (上級)
: Java 実験の場合は、ユーザーの Java コードの実行中には Java スレッドの Java 呼び出しスタックを表示し、JVM コードの実行中または JVM ソフトウェアが Java 呼び出しスタックを報告しないときにはマシン呼び出しスタックを表示します。ハウスキーピングスレッドについてはマシン呼び出しスタックを表示します。

  OpenMP 実験の場合は、コンパイラによって生成された関数を表示します。これらの関数は、ユーザー関数によってユーザーモードで集計される、並列化されたループやタスクなどを表します。また、OpenMP ランタイムが特定の操作を実行しているときは、`<OMP-*>` という形式の名前を持つ特殊関数を追加します。

`machine` (マシン)
: Java 実験と OpenMP 実験の場合は、すべてのスレッドのマシン呼び出しスタックを表示します。

Java 実験と OpenMP 実験を除くすべての実験の場合は、3 つのモードすべてに同じデータを表示します。

compare { on | off }

比較モードをオンまたはオフに設定します。デフォルトでは、値はオフなので、同じ実行可能ファイルの複数の実験が読み込まれると、そのデータは集計されます。.er.rc ファイルで compare on と設定して比較モードを有効にした状態で、同じ実行可能ファイルの複数の実験が読み込まれると、各実験データについて、メトリックスが別々の列に表示されます。er_print compare コマンドを使って、実験を比較することもできます。

比較モードでは、実験または実験グループのデータは、「関数」リスト、「呼び出し元 - 呼び出し先」リスト、「ソース」リスト、および「逆アセンブリ」リストで、隣接列に表示されます。列は、実験または実験グループの読み込み順に表示され、追加のヘッダー行に実験名または実験グループ名が示されます。

その他の情報を出力するコマンド

次の er_print サブコマンドは、実験についてのその他の情報を表示します。

header *exp_id*

指定した実験に関する説明情報を表示します。*exp_id* は、exp_list コマンドを使用して取得することができます。*exp_id* として all を指定するか、exp_id を省略した場合は、読み込まれた実験すべての情報が表示されます。

エラーや警告が発生した場合には、各ヘッダーのあとに表示されます。各実験のヘッダーは、ハイフン (-) で区切られます。

実験ディレクトリに notes という名前のファイルがある場合は、このファイルの内容がヘッダー情報の先頭に付加されます。notes ファイルは、collect コマンドに -C "*comment*" 引数を付けて、手動で追加、編集、または指定できます。

exp_id はコマンド行では必要ですが、スクリプトや対話モードでは不要です。

ifreq

測定されたカウントデータから命令頻度のリストを書き込みます。命令頻度のレポートは、カウントデータからのみ生成できます。このコマンドは、Solaris OS を実行している SPARC プロセッサだけが対象です。

objects

パフォーマンス解析の目的でロードオブジェクトを使用した結果として生じるエラーメッセージや警告メッセージのないロードオブジェクトを一覧表示します。表示されるロードオブジェクトの数は、limit コマンドを使用して制限できます (146 ページの「出力を制御するコマンド」を参照)。

overview *exp_id*

指定した実験の標本のうち、現在選択されている各標本の標本データを書き出します。*exp_id* は、exp_list コマンドを使用して取得することができます。*exp_id* として all を指定するか、または exp_id を省略した場合、すべての実験の標本データが表示されます。*exp_id* はコマンド行では必要ですが、スクリプトや対話モードでは不要です。

statistics *exp_id*

指定した実験の現在の標本セット全体にわたって集計された実行統計情報を書き出します。実行統計値の定義と意味については、getrusage(3C) と proc(4) のマニュアルページを参照してください。実行統計には、コレクタがデータをまったく収集しないシステムスレッドからの統計が含まれます。

exp_id は、experiment_list コマンドを使用して取得することができます。*exp_id* が指定されていない場合、各実験の標本セットを対象に集計された、すべての実験のデータの合計が表示されます。*exp_id* が all である場合、各実験の合計と個々の統計が表示されます。

デフォルト値を設定するコマンド

.er.rc ファイルで次のコマンドを使用して、er_print、er_src、およびパフォーマンスアナライザのデフォルト値を設定できます。これらのコマンドはデフォルト値を設定する目的でのみ使用できます。er_print ユーティリティーの入力として使用することはできません。これらのコマンドは、.er.rc という名前のデフォルト値ファイルにのみ組み込むことができます。パフォーマンスアナライザのデフォルト値にのみ適用できるコマンドについては、151 ページの「パフォーマンスアナライザにのみデフォルト値を設定するコマンド」に説明があります。アナライザによる .er.rc ファイルの使用方法については、119 ページの「アナライザのデフォルト設定」を参照してください。

.er.rc デフォルト値ファイルは、すべての実験のデフォルト値を設定するためにホームディレクトリに置くか、デフォルト値をローカルに設定するためにそれ以外のディレクトリに置くことができます。er_print ユーティリティー、 er_src ユーティリティー、パフォーマンスアナライザのいずれかを起動すると、現在のディレクトリとユーザーのホームディレクトリに .er.rc ファイルがあるかどうかが調べられ、存在する場合は、システムのデフォルト値ファイルとともに、そのファイルが読み取られます。ホームディレクトリの .er.rc ファイル内のデフォルト値はシステムのデフォルト値よりも優先され、現在のディレクトリの .er.rc ファイル内のデフォルト値は、ユーザーのホームおよびシステムのデフォルト値よりも優先されます。

注 – 実験が格納されているディレクトリからデフォルト値ファイルを読み取るには、そのディレクトリからパフォーマンスアナライザまたは er_print ユーティリティーを起動する必要があります。

デフォルト値ファイルには、scc、 sthresh、dcc、dthresh、cc、setpath、addpath、pathmap、name、mobj_define、object_show、object_hide、object_api、indxobj_define、tabs、rtabs、viewmode の各コマンドを含めることもできます。dmetrics、dsort、addpath、pathmap、mobj_define、indxobj_define の各コマンドは、1 つのデフォルト値ファイルに複数含めることができ、その場合、すべての .er.rc ファイル内のコマンドが連結されます。それ以外のすべてのコマンドの場合は、最初に出現するものが使用され、それ以後のものは無視されます。

dmetrics *metric_spec*

関数リストに表示または印刷するデフォルトのメトリックスを指定します。メトリックリストの構文と使用方法については、123 ページの「メトリックリスト」で説明しています。メトリックスが出力される順序とアナライザの「メトリック」ダイアログボックスに表示されるメトリックの順序は、このリスト内のメトリックキーワードの順序によって決まります。

呼び出し元 - 呼び出し先リストのデフォルトのメトリックスは、このリスト内の各メトリック名の最初の名前の前に対応する属性メトリックを追加することによって得られます。

dsort *metric_spec*

関数リストの内容をソートするときの基準として、デフォルトで使用するメトリックを指定します。ソート基準メトリックは、このリスト内で、読み込まれている実験内のメトリックと一致する最初のメトリックです。このとき、次の条件が適用されます。

- *metric_spec* のエントリに表示文字列の感嘆符「!」が含まれている場合、表示されているかどうかに関係なく、一致する名前を持つメトリックスの中で最初のメトリックが使用されます。
- *metric_spec* のエントリにほかの表示文字列が含まれている場合、一致する名前を持つメトリックスの中の最初の表示メトリックが使用されます。

メトリックリストの構文と使用方法については、123 ページの「メトリックリスト」で説明しています。

呼び出し元 - 呼び出し先リストのデフォルトソート基準メトリックは、関数リストのデフォルトソート基準メトリックに対応する属性メトリックです。

en_desc { on | off | =*regexp* **}**

派生実験を読み取るためのモードを `on` (すべての派生実験を有効にする) か `off` (すべての派生実験を無効にする) に設定します。*=regexp* を使用した場合、系統または実行可能ファイル名が正規表現と一致する実験のデータが有効になります。デフォルト設定は `on` で、すべての派生を追跡します。

パフォーマンスアナライザにのみデフォルト値を設定するコマンド

`.er.rc` ファイルで次のコマンドを使用して、パフォーマンスアナライザのデフォルト値を追加できます。

tabs *tab_spec*

アナライザで表示可能にするタブのデフォルトセットを設定します。各タブは、対応するレポートを生成する `er_print` コマンドによって指定されます。たとえば、メモリーオブジェクトのタブの場合は *mobj_type*、インデックスオブジェクトのタブの場合は *indxobj_type* になります。*mpi_timeline* は「MPI タイムライン」タブを、*mpi_chart* は「MPI グラフ」タブを、*timeline* は「タイムライン」タブを、*headers* は「実験」タブを指定します。

読み込まれた実験内のデータがサポートするタブだけが表示されます。

rtabs *tab_spec*

スレッドアナライザの実験を確認するために tha コマンドでアナライザを起動したときに表示可能にするタブのデフォルトセットを設定します。読み込まれた実験内のデータがサポートするタブだけが表示されます。

tlmode *tl_mode*

パフォーマンスアナライザの「タイムライン」タブの表示モードオプションを設定します。オプションのリストは、コロン区切りのリストです。使用できるオプションを次の表に示します。

表 5–6 タイムラインの表示モードのオプション

オプション	意味
`lw[p]`	LWP のイベントを表示する
`t[hread]`	スレッドのイベントを表示する
`c[pu]`	CPU のイベントを表示する
`r[oot]`	ルートで呼び出しスタックを配置する
`le[af]`	リーフで呼び出しスタックを配置する
`d[epth]` *nn*	表示できる呼び出しスタックの最大深さを設定する

`lwp`、`thread`、`cpu` の各オプションは相互排他的です。`root` と `leaf` も相互排他的です。相互排他オプションの複数のセットをリストに含めた場合、最後のオプションだけが使用されます。

tldata *tl_data*

パフォーマンスアナライザの「タイムライン」タブに表示されるデフォルトのデータの種類を選択します。種類リストの種類はコロンで区切られます。使用できるタイプを次の表に示します。

表 5–7 タイムラインに表示するデータの種類

種類	意味
sa[mple]	標本データを表示する
c[lock]	時間プロファイルデータを表示する
hw[c]	ハードウェアカウンタプロファイルデータを表示する
sy[nctrace]	スレッド同期トレースデータを表示する
mp[itrace]	MPI トレースデータを表示する
he[aptrace]	ヒープトレースデータを表示する

その他のコマンド

次のコマンドにより、er_print ユーティリティーでさまざまなタスクが実行されます。

procstats

処理データから蓄積された統計を出力します。

script *file*

file に指定したスクリプトファイル内の追加コマンドを処理します。

version

現在の er_print ユーティリティーのバージョン情報を出力します。

quit

現在のスクリプトの処理を打ち切るか、対話モードを終了します。

help

er_print コマンドの一覧を表示します。

式の文法

フィルタを定義する式と、メモリーオブジェクトインデックスを算出するために使用される式には、共通の文法が使用されます。

その文法は、式を演算子とオペランドの組み合わせとして指定します。フィルタの場合は、式が真と評価されるとパケットが包含され、式が偽と評価されるとパケットが除外されます。メモリーオブジェクトまたはインデックスオブジェクトの場合、式は、パケット内で参照される特定のメモリーオブジェクトまたはインデックスオブジェクトを定義するインデックスへと評価されます。

式のオペランドは定数か、データレコード内のフィールドになります。`describe` コマンドで一覧表示できます。オペランドには、`THRID`、`LWPID`、`CPUID`、`USTACK`、`XSTACK`、`MSTACK`、`LEAF`、`VIRTPC`、`PHYSPC`、`VADDR`、`PADDR`、`DOBJ` またはメモリーオブジェクトの名前が含まれます。オペランドの名前は大文字と小文字が区別されません。

`USTACK`、`XSTACK`、および `MSTACK` は、それぞれユーザー表示、上級表示、マシン表示の関数呼び出しスタックを表します。

`VIRTPC`、`PHYSPC`、`VADDR`、および `PADDR` は、ハードウェアカウンタプロファイリングまたは時間プロファイリングで「+」が指定された場合のみゼロ以外になります。さらに、`VADDR` は、実際の仮想アドレスが決定できなかった場合、256 未満になります。`VADDR` を決定できなかった場合、または、仮想アドレスを物理アドレスにマップできなかった場合、`PADDR` はゼロになります。同様に、バックトラッキングが失敗した場合、または要求されなかった場合、 `VIRTPC` はゼロになり、`VIRTPC` がゼロか、`VIRTPC` が物理アドレスにマップできなかった場合、`PHYSPC` はゼロになります。

演算子は、C の表記法と C の優先順位規則に従った通常の論理演算子と算術 (シフトを含む) 演算子、要素が集合に含まれるかどうかを決める演算子 (`IN`)、要素集合の一部または全部が 1 つの集合に含まれるかどうかを決める演算子 (それぞれ、`SOME IN` か `IN`) のいずれかです。追加の演算子 `ORDERED IN` は、左側のオペランドの全要素が、右側のオペランドに同じ順序で現れているかどうかを判断するためのものです。`IN` 演算子は、左側のオペランドの全要素が右側のオペランドに現れることを要求しますが、順序については強制しません。If-then-else 構造は、C のように `?` 演算子と `:` 演算子で指定されます。すべての式が正しく構文解析されるよう、小括弧を使用してください。`er_print` のコマンド行では、複数の行にまたがって式を分割することはできません。スクリプト内またはコマンド行では、式に空白文字が含まれる場合、その式を二重引用符で囲む必要があります。

フィルタ式はブール値がパケットを包含する場合は真、除外する場合は偽と評価します。スレッド、LWP、CPU、実験 id、プロセス id、および標本のフィルタリングは、適切なキーワードと 1 つの整数を結び付ける関係式、または `IN` 演算子とコンマで区切った整数リストを使用した関係式に基づいて行われます。

時間フィルタリングを使用するには、TSTAMP と時間を結び付ける 1 つ以上の関係式を指定し、時間は、現在パケットが処理されている実験の開始以降のナノ秒数 (整数) で指定します。標本の時間を取得するには、overview コマンドを使用します。overview コマンドでは時間が秒単位で与えられるため、時間フィルタリングに使用するには、ナノ秒に変換する必要があります。時間は、アナライザの「タイムライン」表示から取得することもできます。

関数フィルタリングは、リーフ関数に基づいて行うか、スタック内の任意の関数に基づいて行うことができます。リーフ関数によるフィルタリングを指定するには、LEAF キーワードと整数の関数 id を結び付ける関係式を使用するか、IN 演算子と構造 FNAME("*regexp*") を使用した関係式を使用します。ただし、*regexp* は regexp(5) のマニュアルページで指定されているような正規表現です。現在の *name* の設定によって指定されている関数全体の名前が一致する必要があります。

呼び出しスタック内の任意の関数に基づいたフィルタリングは、構造 FNAME(“*regexp*”) 内の任意の関数が、キーワード USTACK: (FNAME("myfunc") SOME IN USTACK) によって表された関数配列に含まれるかどうかを判定することによって指定されます。

データオブジェクトのフィルタリングは、スタック関数のフィルタリングに似ており、DOBJ キーワードと構造 DNAME("*regexp*") を小括弧で囲んで使用します。

メモリーオブジェクトのフィルタリングを指定するには、mobj_list コマンドに示すようなメモリーオブジェクトの名前とオブジェクトの整数インデックス、または一連のオブジェクトのインデックスを使用します。<Unknown> メモリーオブジェクトのインデックスは、-1 になります。

インデックスオブジェクトのフィルタリングを指定するには、indxobj_list コマンドに示すようなインデックスオブジェクトの名前とオブジェクトの整数インデックス、または一連のオブジェクトのインデックスを使用します。<Unknown> インデックスオブジェクトのインデックスは、-1 になります。

データオブジェクトのフィルタリングとメモリーオブジェクトのフィルタリングは、データ空間データを持つハードウェアカウンタパケットについてのみ意味があり、ほかのすべてのパケットは、そのようなフィルタリングでは除外されます。

仮想アドレスまたは物理アドレスの直接フィルタリングを指定するには、VADDR または PADDR とアドレスの間の関係式を使用します。

メモリーオブジェクトの定義 (137 ページの「mobj_define *mobj_type index_exp*」 を参照) では、1 つの整数インデックスへと評価される式が使用され、VADDR キーワードまたは PADDR キーワードが使用されます。それらの定義は、メモリーカウンタとデータ空間データについてのハードウェアカウンタパケットにのみ適用されます。式は整数を返すか、<Unknown> メモリーオブジェクトについては -1 を返す必要があります。

インデックスオブジェクトの定義 (139 ページの「indxobj_define *indxobj_type index_exp*」 を参照) では、1 つの整数インデックスへと評価される式が使用されます。式は整数を返すか、<Unknown> インデックスオブジェクトについては -1 を返す必要があります。

フィルタ式の例

ここでは、er_print -filters コマンドまたはアナライザのフィルタダイアログボックスで使用できるフィルタ式の例を示します。

er_print -filters コマンドでは、フィルタ式は、次のように単一引用符で囲まれます。

```
er_print -filters 'FNAME("myfunc") SOME IN USTACK' -functions test.1.er
```

例 5–1 名前とスタックによる関数のフィルタリング

ユーザー関数スタックから myfunc という名前の関数をフィルタリングするには、次のようにします。

```
FNAME("myfunc") SOME IN USTACK
```

例 5–2 スレッドと CPU によるイベントのフィルタリング

CPU 2 上でのみ実行したスレッド 1 からのイベントを表示するには、次のようにします。

```
THRID == 1 && CPUID == 2
```

例 5–3 インデックスオブジェクトによるイベントのフィルタリング

インデックスオブジェクト THRCPU が CPUID<<16|THRID として定義されている場合、次のフィルタは、上記の CPU 2 上で実行したスレッド 1 からのイベントを表示する場合のフィルタと等価です。

```
THRCPU == 0x10002
```

例 5–4 指定の時間内に発生したイベントのフィルタリング

5 秒と 9 秒の間に発生した実験 2 のイベントをフィルタリングするには、次のようにします。

```
EXPID==2 && TSTAMP >= 5000000000 && TSTAMP < 9000000000
```

例 5–5 特定の Java クラスのイベントのフィルタリング

スタック内の特定の Java クラスのメソッドを持つイベントをフィルタリングするには、次のようにします。(ユーザー表示モードの場合)

```
FNAME("myClass.*") SOME IN USTACK
```

例 5–6 内部関数 ID と呼び出し順序によるイベントのフィルタリング

関数 ID が既知である (アナライザに表示されている) 場合、マシン呼び出しスタック内の特定の呼び出し順序を含むイベントをフィルタリングするには、次のようにします。

例 5–6 内部関数 ID と呼び出し順序によるイベントのフィルタリング (続き)

```
(314,272) ORDERED IN MSTACK
```

例 5–7 状態または期間によるイベントのフィルタリング

describe コマンドにより、時間プロファイリング実験の次のプロパティが一覧表示された場合は、

```
MSTATE    UINT32  Thread state
NTICK     UINT32  Duration
```

次のフィルタを使用して、特定の状態のイベントを選択できます。

```
MSTATE == 1
```

または、次のフィルタを使用して、特定の状態にあり、期間が 1 クロック刻みより長いイベントを選択できます。

```
MSTATE == 1 && NTICK > 1
```

er_print コマンドの例

ここでは、er_print コマンドの使用例を示します。

例 5–8 関数にかかる時間の概要を表示

```
er_print -functions test.1.er
```

例 5–9 呼び出し元 - 呼び出し先関係を表示

```
er_print -callers-callees test.1.er
```

例 5–10 ホットなソース行を表示

ソース行情報は、コードのコンパイルとリンクで -g が指定されていることを前提にしています。Fortran の関数とルーチンの場合は、関数名の最後に下線を付けてください。関数名のあとの 1 は、myfunction の複数インスタンスを区別するためのものです。

```
er_print -source  myfunction 1 test.1.er
```

例 5–11 ユーザー関数スタックから myfunc という名前の関数をフィルタリング

```
er_print -filters 'FNAME("myfunc") SOME IN USTACK' -functions test.1.er
```

例 5–12 gprof に似た出力を生成

次の例は、実験から gprof 形式に似た一覧を生成します。出力は er_print.out というファイルで、先頭の 100 個の関数と、関数ごとの属性ユーザー時間でソートされた呼び出し元 - 呼び出し先データの一覧です。

```
er_print -outfile  er_print.out -metrics e.%user -sort e.user \
-limit 100 -func -callers-callees test.1.er
```

この例のコマンドを分解して、次の独立した 2 つのコマンドにすることもできます。ただし、大規模な実験またはアプリケーションでは、er_print の呼び出しのたびに、かなりの時間がかかることがありますので注意してください。

```
er_print -metrics  e.%user -limit 100  -functions test.1.er

er_print -metrics  e.%user -callers-callees test.1.er
```

例 5–13 コンパイラのコメントのみを表示

このコマンドを使用するためにプログラムを実行する必要はありません。

```
er_src -myfile.o
```

例 5–14 時計時間プロファイリングを使用して、関数および呼び出し元と呼び出し先を一覧表示

```
er_print -metrics  ei.%wall -functions test.1.er

er_print -metrics aei.%wall  -callers-callees test.1.er
```

例 5–15 er_print コマンドを含むスクリプトを実行

```
er_print -script myscriptfile test.1.er
```

myscriptfile スクリプトに er_print コマンドが含まれます。スクリプトファイルの例を次に示します。

```
## myscriptfile

## Send script output to standard output
outfile -

## Display descriptive information about the experiments
header

## Write out the sample data for all experiments
overview

## Write out execution statistics, aggregated over
## the current sample set for all experiments
statistics

## List functions
```

例 5–15 er_print コマンドを含むスクリプトを実行 (続き)

```
functions

## Display status and names of available load objects
object_list


## Write out annotated disassembly code for systime,
## to file disasm.out
outfile disasm.out
disasm systime


## Write out annotated source code for synprog.c
## to file source.out
outfile source.out
source synprog.c

## Terminate processing of the script
quit
```

◆ ◆ ◆ 第 6 章

パフォーマンスアナライザとそのデータについて

パフォーマンスアナライザは、コレクタが収集したイベントデータを読み取り、そのデータをパフォーマンスメトリックスに変換します。メトリックスは、ターゲットプログラムの構造内の、命令、ソース行、関数、ロードオブジェクトなどのさまざまな要素について計算されます。タイムスタンプ、スレッド ID、LWP ID、および CPU ID を含むヘッダーに加えて、各イベントについて次の 2 つの部分からなるデータが収集され、記録されます。

- メトリックスの計算に使用されるイベント固有のデータ
- プログラム構造へのメトリックスの関連付けに使用されるアプリケーションの呼び出しスタック

プログラム構造にメトリックスを関連付ける処理は、常に簡単にできるとはかぎりません。これは、コンパイラによって、コードの挿入や変換、最適化が行われるためです。この章では、この処理を説明するとともに、パフォーマンスアナライザの表示にそのことがどのように反映されるのかという問題を取り上げます。

この章では、次の内容について説明します。

データ収集の動作

データ収集の実行による出力は実験であり、さまざまな内部ファイルとサブディレクトリを持つディレクトリとしてファイルシステム内に格納されます。

実験の形式

すべての実験には、次の 3 つのファイルが含まれています。

- ログファイル (`log.xml`)。どのようなデータが収集されたか、各種コンポーネントのバージョン、ターゲットが存続している間の各種イベントのレコード、およびターゲットのワードサイズなどに関する情報が含まれている XML ファイル。
- マップファイル (`map.xml`)。どのようなロードオブジェクトがターゲットのアドレス空間に読み込まれたかに関する時間従属情報と、それらのロードオブジェクトが読み込まれたか読み込み解除された時間を記録した XML ファイル。
- オーバービューファイル。実験内のあらゆる標本点で記録された使用情報を含むバイナリファイル。

また、実験にはプロセスが存続している間のプロファイルイベントを表すバイナリデータファイルがあります。各データファイルには、165 ページの「パフォーマンスメトリックスの解釈」で説明しているように、一連のイベントがあります。データの種類ごとに個別のファイルを使用しますが、各ファイルはターゲット内のすべての LWP で共有されます。

時間ベースのプロファイルまたはハードウェアカウンタオーバーフローのプロファイルの場合、データはクロック刻みまたはカウンターオーバーフローによって呼び出されたシグナルハンドラに書き込まれます。同期トレース、ヒープトレース、MPI トレース、または OpenMP トレースの場合は、通常のユーザー呼び出しルーチンで `LD_PRELOAD` 環境変数により割り込み処理される `libcollector` ルーチンからデータが書き込まれます。そのような割り込み処理ルーチンは部分的にデータレコードを記入したあと、通常のユーザー呼び出しルーチンを呼び出し、ルーチンが復帰したときにデータレコードの残りの部分を記入し、データファイルにレコードを書き込みます。

すべてのデータファイルはメモリーマップされ、ブロック単位で書き込まれます。レコードは常に有効なレコード構造を持つように記入されるので、実験は書き込み中に読み取ることができます。LWP 間の競合とシリアル化を最小限にするために、バッファー管理戦略が設計されています。

オプションで、`notes` というファイル名の ASCII ファイルを実験に含めることができます。このファイルは、`collect` コマンドに `-C comment` 引数を使用すると、自動的に作成されます。実験の作成後、ファイルを手動で編集または作成できます。ファイルの内容は、実験のヘッダーの先頭に付加されます。

archives ディレクトリ

各実験には archives ディレクトリがあり、このディレクトリには、map.xml ファイル内で参照されている各ロードオブジェクトについて記述したバイナリファイルがあります。これらのファイルは、データ収集の終了時に実行される er_archive ユーティリティーによって作成されます。プロセスが異常終了すると、er_archive ユーティリティーが呼び出されない場合があります。その場合、アーカイブファイルは最初に実験に対して er_print ユーティリティーまたはアナライザを呼び出したときに書き込まれます。

派生プロセス

派生プロセスは、その実験データを親プロセスの実験ディレクトリのサブディレクトリに書き込みます。

これらの新しい実験には、次のようにそれぞれの系統を示す名前が付けられます。

- 作成者の実験名にアンダースコアが付加される。
- fork には f、exec には x、そのほかの派生実験には c のコード文字が追加される。
- コード文字のあとに、fork または exec のインデックスを示す数字が追加される。この数字は、プロセスが正常に開始されたかどうかに関係なく適用される。
- 実験接尾辞 .er を付加して実験名が完成する。

たとえば親プロセスの実験名が test.1.er の場合、3 回目の fork の呼び出しで作成された子プロセスの実験は test.1.er/_f3.er となります。この子プロセスが新しいイメージを実行した場合、対応する実験名は test.1.er/_f3_x1.er となります。派生実験は親の実験と同じファイルから構成されていますが、派生実験を持たず (すべての派生は親の実験内のサブディレクトリで表される)、アーカイブサブディレクトリを持っていません (すべてのアーカイブが親の実験内へ行われる)。

動的な関数

ターゲットが動的な関数を作成する実験には、map.xml ファイル内に動的な関数を記述する追加レコードと、動的な関数の実際の命令のコピーを含む追加ファイル dyntext があります。動的な関数の注釈付き逆アセンブリを生成するには、このコピーが必要です。

Java 実験

Java 実験の map.xml ファイル内には、その内部処理用に JVM ソフトウェアで作成された動的な関数用と、ターゲット Java メソッドの動的にコンパイルされた (HotSpot) バージョン用の追加レコードがあります。

さらに、Java 実験には、呼び出されたすべての Java ユーザークラスの情報を含む JAVA_CLASSES ファイルがあります。

Java トレースデータは、libcollector.so の一部である JVMTI エージェントを使用して記録されます。エージェントは、記録されたトレースイベントへマップされるイベントを受け取ります。このエージェントは、JAVA_CLASSES ファイルの書き込みに使用するクラスの読み込みと HotSpot のコンパイルのためのイベント、および map.xml ファイル内の Java でコンパイルされたメソッドレコードも受信します。

実験の記録

実験を記録する方法には、次の 3 つがあります。

- collect コマンド
- dbx によるプロセスの生成
- dbx による実行中のプロセスからの実験の作成

アナライザ GUI の「パフォーマンスコレクタ」ウィンドウでは、collect 実験が実行されます。

collect による実験

collect コマンドを使用して実験を記録する場合、collect ユーティリティーは実験ディレクトリを作成し、libcollector.so およびそのほかの libcollector モジュールがターゲットのアドレス空間にあらかじめ読み込まれるように LD_PRELOAD 環境変数を設定します。collect ユーティリティーは、その後、libcollector.so に実験名を知らせるための環境変数とデータ収集オプションを設定し、ターゲットをその上で実行します。

libcollector.so および関連モジュールが、すべての実験ファイルを書き込みます。

dbx でプロセスを作成する実験

データ収集を有効にした状態で dbx を使用してプロセスを起動すると、dbx は実験ディレクトリも作成し、libcollector.so が事前に読み込まれるようにします。dbx は最初の命令の前のブレークポイントでプロセスを停止し、次にデータ収集を開始するために libcollector.so 内の初期化ルーチンを呼び出します。

Java 実験データが dbx で収集できないのは、dbx がデバッグのために Java Virtual Machine Debug Interface (JVMDI) エージェントを使用し、そのエージェントがデータの収集に必要な Java Virtual Machine Tools Interface (JVMTI) エージェントと共存できないからです。

dbx による実行中のプロセスの実験

dbx を使用して実行中のプロセスで実験を開始すると、dbx は実験ディレクトリを作成しますが、LD_PRELOAD 環境変数を使用できません。dbx は対話式関数呼び出しをターゲット内に行なって libcollector.so を開き、次にプロセスを作成する場合と同様に libcollector.so の初期化ルーチンを呼び出します。データは、collect による実験の場合と同様に libcollector.so とそのモジュールによって書き込まれます。

プロセスが開始したときに libcollector.so はターゲットアドレス空間になかったので、ユーザー呼び出し可能関数 (同期トレース、ヒープトレース、MPI トレース) に対する割り込み処理に依存するデータ収集は機能しない場合があります。一般に、シンボルはすでに基礎的な関数に分解されているので、割り込み処理は行えません。さらに、派生プロセスの次も割り込み処理に依存し、実行中プロセスで dbx により作成された実験に対して適切に機能しません。

dbx でプロセスを開始する前、または dbx で実行中プロセスに接続する前に明示的に libcollector.so を事前読み込みした場合は、トレースデータを収集できます。

パフォーマンスメトリックスの解釈

各イベントのデータには、高分解能のタイムスタンプ、スレッド ID、LWP ID、プロセッサ ID が含まれます。パフォーマンスアナライザでは、この最初の 3 つのデータを使用して、時間、スレッド、LWP、または CPU によるメトリックスのフィルタ処理を実行できます。プロセッサ ID については、getcpuid(2) のマニュアルページを参照してください。getcpuid を利用できないシステムでのプロセッサ ID は -1 であり、Unknown にマップされます。

各イベントでは、共通データ以外に、以降の節で説明する固有の raw データが生成されます。これらの節ではまた、raw データから得られるメトリックスの精度と、データ収集がメトリックスに及ぼす影響についても説明しています。

時間ベースのプロファイリング

時間ベースのプロファイリングのイベント固有のデータは、プロファイル間隔カウント値の配列で構成されています。Solaris OS の場合は、間隔カウンタが提供されます。プロファイル間隔の最後で適切な間隔カウンタが 1 増分され、別のプロファイル信号がスケジューリングされます。この配列が記録され、リセットされるのは、Solaris LWP スレッドが CPU ユーザーモードに入った場合だけです。配列のリセット時には、ユーザー CPU 状態の配列要素が 1 に設定され、ほかの全状態の配列要素が 0 に設定されます。配列データが記録されるのは、配列がリセットされる前にユーザーモードに入るときです。したがって、配列には、Solaris LWP ごとにカーネルが保持する 10 個のマイクロステートのそれぞれについて、ユーザーモードに前回入って以降の各マイクロステートのカウントの累計値が含まれます。Linux OS ではマイクロステートは存在せず、利用できる間隔カウンタはユーザー CPU 時間だけです。

呼び出しスタックは、データと同時に記録されます。プロファイル間隔の最後で Solaris LWP がユーザーモードでない場合は、LWP またはスレッドが再びユーザーモードになるまで、呼び出しスタックの内容は変わりません。すなわち、呼び出しスタックには、各プロファイル間隔の最後のプログラムカウンタの位置が常に正確に記録されます。

表 6–1 に、各マイクロステートとメトリックスの、Solaris OS における対応関係を示します。

表 6–1 カーネルのマイクロステートとメトリックスとの対応関係

カーネルのマイクロステート	内容の説明	メトリック名
LMS_USER	ユーザーモードで動作	ユーザー CPU 時間
LMS_SYSTEM	システムコールまたはページフォルトで動作	システム CPU 時間
LMS_TRAP	上記以外のトラップで動作	システム CPU 時間
LMS_TFAULT	ユーザーテキストページフォルトでスリープ	テキストページフォルト時間
LMS_DFAULT	ユーザーデータページフォルトでスリープ	データページフォルト時間
LMS_KFAULT	カーネルページフォルトでスリープ	ほかの待ち時間
LMS_USER_LOCK	ユーザーモードロック待ちのスリープ	ユーザーロック時間
LMS_SLEEP	ほかの理由によるスリープ	ほかの待ち時間
LMS_STOPPED	停止 (/proc、ジョブ制御、lwp_stop のいずれか)	ほかの待ち時間
LMS_WAIT_CPU	CPU 待ち	CPU 待ち時間

タイミングメトリックスの精度

このため、ほかの統計的な標本収集手法と同様に、あらゆる誤差の影響を受けます。プログラムの実行時間が非常に短い場合は、少数のプロファイルパケットしか記録されず、多くのリソースを消費するプログラム部分が、呼び出しスタックに反映されないことがあります。このため、目的の関数またはソース行について数百のプロファイルパケットを蓄積するのに十分な時間または十分な回数に渡って、プログラムを実行するようにしてください。

統計的な標本収集の誤差のほかに、データの収集?関連付け方法、システムにおけるプログラムの実行の進み具合を原因とする誤差もあります。次に示す環境などでは、タイミングメトリックスでデータに不正確さやひずみが生じる可能性があります。

- Solaris LWP または Linux スレッドが作成されるとき、最初のプロファイルパケットが記録されるまでの時間はプロファイル間隔より短いですが、プロファイル間隔全体の時間が、最初のプロファイルパケットに記録されるマイクロステートに帰せられます。多数の LWP またはスレッドが作成される場合、誤差はプロファイル間隔の数倍になることがあります。

- Solaris LWP または Linux スレッドが破壊されるとき、最後のプロファイルパケットが記録されてから、少し時間が費やされます。多数の LWP またはスレッドが破壊される場合、誤差はプロファイル間隔の数倍になることがあります。
- プロファイル間隔中に LWP またはスレッドの再スケジューリングが行われることがあります。このため、LWP について記録された状態に、プロファイル間隔の大半を費やしたマイクロステートが反映されないことがあります。Solaris LWP または Linux スレッドの方がそれらを実行するプロセッサの個数より多いほど、誤差は大きくなる可能性があります。
- プログラムがシステムクロックと相関関係を持つ形で動作することがあります。この場合、Solaris LWP または Linux スレッドが費やされた時間のごく一部を表す状態にあるときに、常にプロファイル間隔の時間切れになり、プログラムの特定部分について記録された呼び出しスタックの出現回数が実際より多くなります。マルチプロセッサシステムでは、プロファイルシグナルによって相関関係が生じる場合があります。プログラムの LWP を実行中にプロファイルシグナルによって中断されたプロセッサが、マイクロステートの記録時にトラップ CPU マイクロステートになる可能性があります。
- カーネルは、プロファイル間隔の時間切れになったときにマイクロステート値を記録します。システムが過負荷状態の場合、この値に、プロセスの本当の状態が反映されないことがあります。Solaris OS では、この結果、トラップ CPU または CPU 待ちマイクロステート値が実際より大きくなることがあります。
- システムクロックと外部ソースとの同期がとられている場合、プロファイルパケットに記録されるタイムスタンプはプロファイル間隔を反映しませんが、システムクロックに対して施された調整結果は組み込まれます。 システムクロック調整の結果、プロファイルパケットが失われたかのように見える可能性があります。その時間は通常数秒間であり、調整は一定の増分単位で行われます。
- 動作クロック周波数が動的に変わるマシンで記録される実験は、プロファイリングの不正確さが生じることがあります。

これらの不正確さのほかに、データ収集処理そのものが原因でタイミングメトリックスが不正確になります。記録はプロファイルシグナルによって開始されるため、プロファイルパケットの記録に費やされた時間が、プログラムのメトリックスに反映されることはありません。これは、相関関係の別の例です。記録に費やされたユーザー CPU 時間は、記録されるあらゆるマイクロステート値に配分されます。この結果、ユーザー CPU 時間のメトリックが実際より小さくなり、その他のメトリックスが実際より大きくなります。デフォルトのプロファイル間隔の場合、一般に、データの記録に費やされる時間は CPU 時間の 2、3% 未満です。

タイミングメトリックスの比較

時間ベースの実験のプロファイリングで得られたタイミングメトリックスと、その他の方法で得られた時間を比較する場合は、次の点に注意する必要があります。

シングルスレッドアプリケーションの場合、1 つのプロセスについて記録された Solaris LWP または Linux スレッドの合計時間は、同じプロセスについて `gethrtime`

(3C) によって返される値と比較して、誤差は数十分の 1 パーセントです。CPU 時間の場合は、同じプロセスについて gethrvtime(3C) によって返される値と比較して、数パーセント程度異なることがあります。負荷が大きい場合は、差がさらに大きくなることがあります。ただし、CPU 時間の差は規則的なひずみを表すものではなく、関数、ソース行などについて報告される相対時間に大きなひずみはありません。

Solaris OS の非結合スレッドを使用するマルチスレッドアプリケーションの場合、gethrvtime() によって返される値の差が無意味であることがあります。これは、gethrvtime() は LWP について値を返し、スレッドは LWP ごとに異なることがあるからです。

パフォーマンスアナライザの報告する LWP 時間が、vmstat の報告する時間とかなり異なることがあります。これは、vmstat が CPU 全体にまたがって集計した時間を報告するためです。たとえば、ターゲットプロセスの LWP 数が、そのプロセスが動作するシステムの CPU 数よりも多い場合、アナライザは、vmstat が報告する時間よりもずっと長い待ち時間を報告します。

パフォーマンスアナライザの「統計」タブと er_print 統計ディスプレイに表示されるマイクロステート時間値は、プロセスファイルシステムの /proc 使用報告に基づいており、この報告には、マイクロステートで費やされる時間が高い精度で記録されます。詳細は、proc (4) のマニュアルページを参照してください。これらのタイミング値と <Total> 関数 (プログラム全体を表す) のメトリックスを比較することによって、集計されたタイミングメトリックスのおおよその精度を知ることができます。ただし、「統計」タブに表示される値には、<Total> のタイミングメトリック値に含まれないそのほかの関連要素が含まれることがあります。その原因は、データ収集が一時停止される期間によるものです。

ユーザー CPU 時間とハードウェアカウンタサイクル時間は異なります。なぜなら、ハードウェアカウンタは、CPU モードがシステムモードへ切り替えられたときにオフにされるからです。詳細は、173 ページの「トラップ」を参照してください。

同期待ちトレース

コレクタは、スレッドライブラリ libthread.so 内の関数の呼び出し、またはリアルタイム拡張ライブラリ librt.so の呼び出しをトレースすることによって、同期遅延イベントを収集します。イベント固有のデータは、要求と許可 (トレース対象の呼び出しの始まりと終わり) の高分解能のタイムスタンプと、同期オブジェクト (要求された相互排他ロックなど) のアドレスとで構成されます。スレッド ID と LWP ID は、データが記録された時点での ID です。待ち時間は、要求時刻と許可時刻の時間差で示されます。記録されるイベントは、指定したしきい値を要求と許可の時間差が超えたものだけです。同期待ちトレースデータは、許可時に実験ファイルに記録されます。

遅延の原因となったイベントが完了しないかぎり、待ちスレッドがスケジューリングされている LWP がほかの作業を行うことはできません。この待ち時間は、「同期待ち時間」と「ユーザーロック時間」の両方に反映されます。同期遅延しきい値は短時間の遅延を排除するので、「ユーザーロック時間」が「同期待ち時間」よりも大きくなる可能性があります。

待ち時間は、データ収集のオーバーヘッドによってひずみます。そして、このオーバーヘッドは、収集されたイベントの個数に比例します。オーバーヘッドに費やされる待ち時間の一部は、イベント記録のしきい値を大きくすることによって最小化できます。

ハードウェアカウンタオーバーフローのプロファイリング

ハードウェアカウンタオーバーフローのプロファイルデータには、カウンタ ID とオーバーフロー値が含まれます。この値は、カウンタがオーバーフローするように設定されている値よりも大きくなることがあります。これは、オーバーフローが発生して、そのイベントが記録されるまでの間に命令が実行されるためです。この値は特に、浮動小数点演算やキャッシュミスなどのカウンタよりも、ずっと頻繁に増分されるサイクルカウンタや命令カウンタの場合に大きくなる可能性があります。イベント記録時の遅延はまた、呼び出しスタックとともに記録されたプログラムカウンタのアドレスが正確にオーバーフローイベントに対応しないことを意味します。詳細は、210 ページの「ハードウェアカウンタオーバーフローの関連付け」を参照してください。また、173 ページの「トラップ」も参照してください。トラップおよびトラップハンドラは、ユーザーの CPU 時間とサイクルカウンタによって報告される時間の間の、大きな相違の原因になることがあります。

動作クロック周波数が動的に変わるマシンで記録される実験では、サイクルベースのカウントから時間への変換で不正確さが生じます。

収集されるデータ量は、オーバーフロー値に依存します。選択した値が小さすぎると、次のような影響が出ることがあります。

- データの収集に費やされる時間が、プログラムの実行時間のかなりの部分を占めることがあります。収集実行では、プログラムの実行ではなく、オーバーフローの処理とデータの書き込みに時間のかなりが費やされる場合があります。
- カウントのかなりの部分の原因がデータ収集であることがあります。こうしたカウントは、コレクタ関数の `collector_record_counters` によるものです。この関数のカウントが大きい場合は、オーバーフロー値が小さすぎます。
- データ収集によってプログラムの動作が変わることがあります。たとえば、キャッシュミスのデータの収集では、キャッシュミスの大半が、コレクタの命令のフラッシュとキャッシュからのデータのプロファイリング、プログラム命

令とデータとの置き換えによるものです。プログラムで多くのキャッシュミスが発生するように見えますが、データ収集を行わなければキャッシュミスはごくわずかであった可能性があります。

ヒープトレース

コレクタは、メモリーの割り当てと割り当て解除の関数である `malloc`、`realloc`、`memalign`、`free` に割り込むことによって、これらの関数の呼び出しに関するトレースデータを記録します。メモリーを割り当てるときにこれらの関数を迂回するプログラムの場合、トレースデータは記録されません。別のメカニズムが使用されている Java メモリー管理では、トレースデータは記録されません。

トレース対象の関数は、さまざまなライブラリから読み込まれる可能性があります。パフォーマンスアナライザで表示されるデータは、読み込み対象の関数が属しているライブラリに依存することがあります。

短時間で大量のトレース対象関数を呼び出すプログラムの場合、プログラムの実行に要する時間が大幅に長くなることがあります。延びた時間は、トレースデータの記録に使用されます。

データ空間プロファイリング

データ空間プロファイリングは、メモリ参照に対して使用されるハードウェアカウンタプロファイリングの拡張版です。ハードウェアカウンタプロファイリングでは、メトリックスをユーザー関数、ソース行、および命令に割り当てることができますが、参照されるデータオブジェクトに割り当てることはできません。デフォルトでは、コレクタはユーザー命令アドレスのみを取得します。データ空間プロファイリングが有効な場合、コレクタはデータアドレスも取得します。バックトラッキングとは、データ空間プロファイリングをサポートするパフォーマンス情報の取得に使用されるテクニックです。バックトラッキングが有効な場合、コレクタは、ハードウェアカウンタイベント前に発生したロード命名またはストア命令に戻って、そのイベントの原因になった可能性のある命令の候補を1つ見つけます。

データ空間プロファイリングを可能にするには、ターゲットは、`-xhwcprof` フラグと `-xdebugformat=dwarf -g` フラグを付けて SPARC アーキテクチャー用にコンパイルされた C プログラムである必要があります。さらに、収集されるデータは、メモリー関係のカウンタのハードウェアカウンタプロファイルでなければならず、カウンタ名の前に + 記号を付加する必要があります。パフォーマンスアナライザには、データ空間プロファイリング関係のタブとして、「データオブジェクト」タブと「データレイアウト」タブのほか、メモリーオブジェクト用の各種のタブが含まれています。

データ空間プロファイリングは、プロファイル間隔の前にプラス記号 (+) を付加することで、時間ベースのプロファイリングとともに実施できます。

引数なしで collect を実行すると、ハードウェアカウンタが一覧表示され、それらが、ロード関係なのか、ストア関係なのか、ロード-ストア関係なのかが示されます。27 ページの「ハードウェアカウンタオーバーフローのプロファイルデータ」を参照してください。

MPI トレース

MPI トレースは、VampirTrace データコレクタの修正版をベースにしています。詳細は、Technische Universität Dresden Web サイトにある『Vampirtrace User Manual』を参照してください。

呼び出しスタックとプログラムの実行

呼び出しスタックは、プログラム内の命令を示す一連のプログラムカウンタ (Program Counter、PC) アドレスです。リーフ *PC* と呼ばれる最初の PC はスタックの一番下に位置し、次に実行される命令のアドレスを表します。次の PC はそのリーフ PC を含む関数の呼び出しアドレス、そして、その次の PC がその関数の呼び出しアドレスというようにして、これがスタックの先頭まで続きます。こうしたアドレスはそれぞれ、復帰アドレスと呼びます。呼び出しスタックの記録プロセスでは、プログラムスタックから復帰アドレスが取得されます。これは、スタックの展開と呼ばれています。 展開の失敗については、184 ページの「不完全なスタック展開」を参照してください。

呼び出しスタック内のリーフ PC は、この PC が存在する関数にパフォーマンスデータの排他的メトリックスを割り当てるときに使用されます。スタック上の各 PC は、リーフ PC も含めて、その PC が存在する関数に包括的メトリックスを割り当てるために使用されます。

ほとんどの場合、記録された呼び出しスタック内の PC は、プログラムのソースコードに現れる関数に自然な形で対応しており、パフォーマンスアナライザが報告するメトリックスもそれらの関数に直接対応しています。しかし、プログラムの実際の実行は、単純で直観的なプログラム実行モデルと対応しないことがあり、その場合は、アナライザの報告するメトリックスが混乱を招くことがあります。こうした事例については、185 ページの「プログラム構造へのアドレスのマッピング」を参照してください。

シングルスレッド実行と関数呼び出し

プログラムの実行でもっとも単純なものは、シングルスレッドプログラムが専用のロードオブジェクト内の関数を呼び出す場合です。

プログラムがメモリーに読み込まれて実行が開始されると、初期実行アドレス、初期レジスタセット、スタック (スクラッチデータの格納および関数の相互の呼び出し方法の記録に使用されるメモリー領域) からなるコンテキストが作成されます。初期アドレスは常に、あらゆる実行可能ファイルに組み込まれる `_start()` 関数の先頭位置になります。

プログラムを実行すると、たとえば関数呼び出しや条件文を表すことがある分岐命令があるまで、命令が順実行されます。分岐点では、分岐先が示すアドレスに制御が渡されて、そこから実行が続行されます。通常、分岐の次の命令は実行するようすでにコミットされており、この命令は分岐遅延スロット命令と呼ばれます。ただし、分岐命令には、この分岐遅延スロット命令の実行を無効にするものもあります。

呼び出しを表す命令シーケンスが実行されると、復帰アドレスがレジスタに書き込まれ、呼び出された関数の最初の命令から実行が続行されます。

ほとんどの場合は、この呼び出し先の関数に含まれる最初の数個の命令のどこかで、新しいフレーム (関数に関する情報を格納するためのメモリー領域) がスタックにプッシュされ、そのフレームに復帰アドレスが格納されます。復帰アドレスに使用されるレジスタは、呼び出された関数がほかの関数を呼び出すときに使用できます。関数から制御が戻されようとすると、スタックからフレームがポップされ、関数の呼び出し元のアドレスに制御が戻されます。

共有オブジェクト間の関数の呼び出し

共有オブジェクト内の関数が別の共有オブジェクトの関数を呼び出す場合は、同じプログラム内の単純な関数の呼び出しよりも実行が複雑になります。各共有オブジェクトには、プログラムリンケージテーブル (PLT) が 1 つあり、ここにはそのオブジェクトが参照し、そのオブジェクトの外部にあるすべての関数 (外部関数) のエントリが含まれます。最初は、PLT 内の各外部関数のアドレスは、実際には動的リンカーである `ld.so` 内のアドレスです。外部関数が初めて呼び出されると、制御が動的リンカーに移り、動的リンカーは、その外部関数への呼び出しを解決し、以降の呼び出しのために、PLT のアドレスにパッチを適用します。

3 つの PLT 命令のいずれか 1 つを実行しているときにプロファイリングイベントが発生した場合、PLT PC は削除され、排他的時間はその呼び出し命令に対応することになります。PLT エントリによる最初の呼び出し時にプロファイリングイベントが発生し、かつリーフ PC が PLT 命令のいずれかではない場合、PLT および `ld.so` のコードから発生する PC はすべて、包括的時間を集計する擬似的な関数 `@plt` に帰属します。各共有オブジェクトには、こういった擬似的な関数が 1 つ用意されています。`LD_AUDIT` インタフェースを使用しているプログラムの場合、PLT エントリが絶対にパッチされない可能性があるとともに、`@plt` の非リーフ PC の発生頻度が高くなることが考えられます。

シグナル

シグナルがプロセスに送信されると、各種のレジスタ操作とスタック操作が発生し、シグナル送信時にリーフ PC がシステム関数 `sigacthandler()` の呼び出しの復帰アドレスを示していたかのように見えます。`sigacthandler()` は、関数が別の関数を呼び出すのと同じようにして、ユーザー指定のシグナルハンドラを呼び出します。

パフォーマンスアナライザは、シグナル送信で発生したフレームを通常のフレームとして処理します。シグナル送信時のユーザーコードがシステム関数 `sigacthandler()` の呼び出し元として表示され、`sigacthandler()` はユーザーのシグナルハンドラの呼び出し元として表示されます。`sigacthandler()` とあらゆるユーザーシグナルハンドラ、さらにはそれらが呼び出すほかの関数の包括的メトリックスは、割り込まれた関数の包括的メトリックスとして表示されます

コレクタは `sigaction()` に割り込むことによって、時間データ収集時にはそのハンドラが `SIGPROF` シグナルのプライマリハンドラになり、ハードウェアカウンタオーバーフローのデータ収集時には `SIGEMT` シグナルのプライマリハンドラになるようにします。

トラップ

トラップは命令またはハードウェアによって発行され、トラップハンドラによって捕捉されます。システムトラップは、命令から発行され、カーネルにトラップされるトラップです。すべてのシステムコールは、トラップ命令を使用して実装されます。ハードウェアトラップの例としては、命令 (UltraSPARC III プラットフォームでのレジスタ内容値の `fitos` 命令など) を最後まで実行できないとき、あるいは命令がハードウェアに実装されていないときに、浮動小数点演算装置から発行されるトラップがあります。

トラップが発行されると、Solaris LWP または Linux カーネルはシステムモードになります。Solaris OS 上では、通常、これでマイクロステートはユーザー CPU 状態からトラップ状態、そしてシステム状態に切り替わります。マイクロステートの切り替わりポイントによっては、トラップの処理に費やされた時間が、システム CPU 時間とユーザー CPU 時間を合計したものとして現れることがあります。この時間は、ユーザーコード内でそのトラップを発行した命令、またはシステムコールに割り当てられます。

一部のシステムコールでは、こうした呼び出しをできるかぎり効率良く処理することが重要とみなされます。こうした呼び出しによって生成されたトラップを高速トラップと呼びます。高速トラップを生成するシステム関数には、`gethrtime` および `gethrvtime` があります。これらの関数ではオーバーヘッドを伴うため、マイクロステートは切り替わりません。

その他にも、トラップをできるかぎり効率良く処理することが重要とみなされる環境があります。たとえば、TLB (Translation look aside buffer、変換索引バッファー) ミスやレジスタウィンドウのスピル/フィルの場合も、マイクロステートは切り替わりません。

どちらの場合で、費やされた時間はユーザー CPU 時間として記録されます。ただし、システムモードに CPU モードが切り替えられたため、ハードウェアカウンタは動作していません。このため、これらのトラップの処理に費やされた時間は、なるべく同じ実験で記録された、ユーザー CPU 時間とサイクル時間の差を取ることで求めることができます。

トラップハンドラがユーザーモードに戻るケースもあります。Fortran で 4 バイトメモリー境界に整列された整数に対し、8 バイトのメモリー参照を行うようなトラップです。スタックにトラップハンドラのフレームが現れ、整数ロードまたはストア命令が原因でパフォーマンスアナライザにハンドラの呼び出しが表示される場合があります。

命令がカーネルにトラップされると、そのトラップ命令のあとの命令の実行に長い時間がかかっているようにみえます。これは、カーネルがトラップ命令の実行を完了するまで、その命令の実行を開始できないためです。

末尾呼び出しの最適化

特定の関数がその最後で別の関数を呼び出す場合、コンパイラは特別な最適化を行うことができます。新しいフレームを生成するのではなく、呼び出し先が呼び出し元のフレームを再利用し、呼び出し先用の復帰アドレスが呼び出し元からコピーされます。この最適化の目的は、スタックのサイズ削減、および SPARC プラットフォーム上でのレジスタウィンドウの使用削減にあります。

プログラムのソースの呼び出しシーケンスが、次のようになっていると仮定します。

```
A -> B -> C -> D
```

A -> B -> C -> D および C に対して末尾呼び出しの最適化を行うと、呼び出しスタックは、関数 A が関数 B、C、D を直接呼び出しているかのようになります。

```
A -> B
A -> C
A -> D
```

つまり、呼び出しツリーがフラットになります。-g オプションを指定してコードをコンパイルした場合、末尾呼び出しの最適化は、4 以上のレベルでのみ行われます。-g オプションなしでコードをコンパイルした場合、2 以上のレベルで末尾呼び出しの最適化が行われます。

明示的なマルチスレッド化

Solaris OS では、簡単なプログラムは、単一の LWP (LightWeight Process、軽量プロセス) 上のシングルスレッド内で動作します。マルチスレッド化した実行可能ファイルは、スレッド作成関数を呼び出し、その関数に実行するターゲット関数が渡されます。ターゲットの終了時にスレッドが破棄されます。

Solaris OS では、Solaris スレッドと POSIX スレッド (Pthread) の 2 種類のスレッド実装がサポートされています。Solaris 10 OS 以降、両方のスレッド実装が `libc.so` に含まれます。

Solaris のスレッドでは、新しく作成されたスレッドは、スレッド作成呼び出しで渡された関数を呼び出す `_thread_start()` という関数で実行を開始します。このスレッドによって実行されるターゲットが関係するどの呼び出しスタックでも、スタックの先頭は `_thread_start()` であり、スレッド作成関数の呼び出し元に接続することはありません。このため、作成されたスレッドに関連付けられた包括的メトリックスは、`_thread_start()` と `<Total>` 関数にのみ及びます。スレッドの作成に加えて、Solaris のスレッド実装では、スレッドを実行するために LWP が Solaris 上に作成されます。LWPSolaris スレッドによる作成スレッドはそれぞれ特定の LWP に結合されます。

Solaris 10 OS と Linux OS では、明示的なマルチスレッド化に Pthread を使用できます。

どちらの環境でも、新しいスレッドを作成するには、アプリケーションが Pthread API 関数 `pthread_create()` を呼び出して、関数引数の 1 つとして、ポインタをアプリケーション定義の起動ルーチンに渡します。

Solaris OS では、新しい pthread の実行の開始時に `_lwp_start()` 関数が呼び出されます。Solaris 10 OS では、`_lwp_start()` から中間関数 `_thr_setup()` が呼び出され、その中間関数から `pthread_create()` で指定されたアプリケーション定義の起動ルーチンが呼び出されます。

Linux OS では、新しい pthread の実行の開始時に Linux 固有のシステム関数 `clone()` が呼び出され、この関数から別の内部初期化関数 `pthread_start_thread()` が呼び出され、この関数から、`pthread_create()` で定義されたアプリケーション定義の起動ルーチンが呼び出されます。コレクタで使用できる Linux メトリックス収集関数はスレッドに固有です。したがって、`collect` ユーティリティーを実行すると、これは `pthread_start_thread()` とアプリケーション定義のスレッド起動ルーチンの間に、`collector_root()` という名前のメトリックス収集関数を割り込ませます。

Java テクノロジーベースのソフトウェア実行の概要

典型的な開発者にとっては、Java テクロノジーベースのアプリケーションはほかのプログラムと同じように動作します。このアプリケーションは、一般に `class.main` というメインエントリポイントから始まり、C または C++ アプリケーションの場合と同様に、ほかのメソッドを呼び出すことがあります。

オペレーティングシステムにとっては、Java プログラミング言語で書かれたアプリケーション (純粋なものか、C/C++ が混合しているもの) は JVM ソフトウェアをインスタンス化するプロセスとして動作します。JVM ソフトウェアは C++ ソースからコ

ンパイルされ、_start から実行を開始し、それが main を呼び出すというように処理が進行します。このソフトウェアは .class ファイルまたは .jar ファイルからバイトコードを読み取り、そのプログラムで指定された操作を実行します。指定できる操作の中には、ネイティブ共有オブジェクトの動的な読み込みや、そのオブジェクト内に含まれている各種関数やメソッドへの呼び出しがあります。

JVM ソフトウェアは、従来の言語で記述されたアプリケーションでは一般に行われない多数の動作を行います。起動時に、このソフトウェアはデータ空間に動的に生成されたコードの多数の領域を作成します。これらの領域の 1 つは、アプリケーションのバイトコードメソッドを処理するために使用される実際のインタプリタコードです。

Java テクノロジーベースのアプリケーションの実行中、大半のメソッドは JVM ソフトウェアで解析されます。本書では、これらのメソッドをインタプリタされたメソッドと呼んでいます。Java HotSpot 仮想マシンによって、バイトコードの解析時にパフォーマンスが監視され、頻繁に実行されているメソッドが検出されます。繰り返し実行されているメソッドは、Java HotSpot 仮想マシンによってコンパイルされ、マシンコードが生成される場合があります。マシンコードが生成されたメソッドは、コンパイルされたメソッドと呼びます。仮想マシンでは、その後、メソッドの元のバイトコードを解析せずに、より効率的な、コンパイルされたメソッドが実行されます。コンパイルされたメソッドはアプリケーションのデータ空間に読み込まれ、その後のある時点で読み込み解除することができます。さらに、インタプリタされたコードとコンパイルされたコードの間の変換を行うために、ほかのコードがデータ空間で生成されます。

Java プログラミング言語で書かれたコードは、コンパイルされたネイティブコード、すなわち、C、C++、または Fortran 内へ直接呼び出すこともでき、そのような呼び出しのターゲットをネイティブメソッドと呼びます。

Java プログラミング言語で記述されたアプリケーションは本質的にマルチスレッド型で、ユーザーのプログラム内でスレッドごとに 1 つの JVM ソフトウェアスレッドがあります。Java アプリケーションはまた、シグナル処理、メモリー管理、Java HotSpot 仮想マシンのコンパイルに使用されるハウスキーピングスレッドもいくつかあります。

データの収集は、J2SE の JVMTI にあるさまざまなメソッドを使用して実装されます。

Java 呼び出しスタックとマシン呼び出しスタック

パフォーマンスツールは、各 Solaris LWP または Linux スレッドの存続期間中にイベントを記録するほか、イベント時に呼び出しスタックを記録することによってデータを収集します。任意のアプリケーションの実行の任意の時点で、呼び出しスタックは、プログラムが実行のどの段階まで、またどのように達したかを表します。混合モデル Java アプリケーションが従来の C、C++、および Fortran アプリケーションと異なる 1 つの重要な点は、ターゲットの実行中は常に、意味のある呼

び出しスタックとして、Java 呼び出しスタックとマシン呼び出しスタックがあるという点です。両方の呼び出しスタックがプロファイル時に記録され、解析時に調整されます。

時間ベースのプロファイリングとハードウェアカウンタオーバーフローのプロファイリング

Java プログラムに対する時間ベースのプロファイルおよびハードウェアカウンタオーバーフローのプロファイルは、Java 呼び出しスタックとマシン呼び出しスタックの両方が収集されることを除けば、C や C++、Fortran プログラムに対するのと完全に同じ働きをします。

Java プロセスの表現

Java プログラミング言語で書かれたアプリケーションについては、パフォーマンスデータを表示するための表現方法として、Java ユーザー表現、Java 上級表現、マシン表現があります。デフォルトでは、データが Java ユーザー表現をサポートする場合は、Java ユーザー表現で表示されます。以降では、これらの 3 つの表現の主な違いをまとめます。

ユーザー表現

ユーザー表現は、コンパイルされた Java メソッドとインタプリタされた Java メソッドを名前で表示し、ネイティブメソッドをそれらの自然な形式で表示します。実行中は、実行される特定の Java メソッドのインスタンスが多数存在する可能性があります。つまり、インタプリタバージョンと、場合によっては 1 つ以上のコンパイルバージョンが存在する可能性があります。Java ユーザー表現では、すべてのメソッドが 1 つのメソッドとして集計された状態で表示されます。アナライザでは、この表現がデフォルトで選択されます。

Java メソッド (Java ユーザー表現内) の PC は、メソッド ID とそのメソッドへのバイトコードのインデックスに対応し、ネイティブ関数の PC はマシン PC に対応します。Java スレッドの呼び出しスタックには、Java PC とマシン PC が混在している場合があります。この呼び出しスタックには、Java ユーザー表現を持たない Java ハウスキーピングコードに対応するフレームはありません。状況によっては、JVM ソフトウェアは Java スタックを展開することができず、`<no Java callstack recorded>` という特別な関数を持つシングルフレームが返されます。これは通常、合計時間の 5 〜 10% にしかなりません。

Java ユーザー表現の関数リストは、Java メソッドと呼び出された任意のネイティブメソッドに対するメトリックスを示します。呼び出し元 - 呼び出し先のパネルには、呼び出しの関係が Java ユーザー表現で示されます。

Java メソッドのソースは、コンパイル元の .java ファイル内のソースコードに対応し、各ソース行にメトリックスがあります。Java メソッドの逆アセンブリは作成されたバイトコードのほか、各バイトコードに対するメトリックスとインタリーブされた Java ソース (入手可能な場合) を示します。

Java ユーザー表現のタイムラインは、Java スレッドのみを示します。各スレッドの呼び出しスタックが、その Java メソッドとともに示されます。

Java ユーザー表現のデータ空間プロファイリングは、現在サポートされていません。

上級ユーザー表現

Java 上級表現は、JVM 内部要素の詳細のいくつかを除いては Java ユーザー表現に似ています。Java ユーザー表現では表示されない JVM 内部要素の詳細のいくつかが、Java 上級表現に表されます。Java 上級表現では、タイムラインがすべてのスレッドを示し、ハウスキーピングスレッドの呼び出しスタックはネイティブ呼び出しスタックです。

マシン表現

マシン表現には、JVM ソフトウェアでインタプリタされるアプリケーションからの関数でなく、JVM ソフトウェア自体からの関数が表示されます。また、コンパイルされたメソッドとネイティブメソッドがすべて表示されます。マシン表現は、従来の言語で書かれたアプリケーションの表現と同じように見えます。呼び出しスタックは、JVM フレーム、ネイティブフレーム、およびコンパイル済みメソッドフレームを表示します。JVM フレームの中には、インタプリタされた Java、コンパイルされた Java、およびネイティブコードの間の変移コードを表すものがあります。

コンパイルされたメソッドからのソースは Java ソースに対照して表示され、データは選択されたコンパイル済みメソッドの特定のインスタンスを表します。コンパイルされたメソッドの逆アセンブリは、Java バイトコードでなく作成されたマシンアセンブラコードを示します。呼び出し元 - 呼び出し先の関係はすべてのオーバーヘッドフレームと、インタプリタされたメソッド、コンパイルされたメソッド、ネイティブメソッドの間の遷移を表すすべてのフレームを示します。

マシン表現のタイムラインはすべてのスレッド、LWP または CPU のバーを示し、それぞれの呼び出しスタックはマシン表現呼び出しスタックになります。

OpenMP ソフトウェア実行の概要

OpenMP アプリケーションの実際の実行モデルについては、OpenMP の仕様 (たとえば、『OpenMP Application Program Interface, Version 3.0』の 1.3 節を参照) で説明されています。しかし、仕様には、ユーザーにとって重要と思われるいくつかの実装の

詳細が説明されていません。また、Oracle での実際の実装では、直接記録されたプロファイリング情報からユーザーがスレッド間の相互作用を簡単に理解できないことがわかっています。

ほかのシングルスレッドプログラムが実行されるとき同様に、呼び出しスタックが現在位置と、どのようにしてそこまで到達したかのトレースを、ルーチン内の `_start` と呼ばれる冒頭の命令を始めとして表示します。このルーチンは `main` を呼び出し、その後、main によって処理が進められ、プログラム内のさまざまなサブルーチンが呼び出されます。サブルーチンにループが含まれている場合、プログラムは、ループ終了条件が満たされるまでループ内のコードを繰り返し実行します。その後、実行は次のコードシーケンスへ進み、以後同様に処理が続きます。

プログラムが OpenMP (または、自動並列化処理) によって並列化されると、動作は異なります。並列化されたプログラムの直感的なモデルでは、メインスレッド (マスタースレッド) が、シングルスレッドプログラムとまったく同じように実行されます。並列ループまたは並列領域に到達すると、追加のスレーブスレッドが出現します。それらの各スレッドはマスタースレッドのクローンであり、それらのスレッドすべてが、ループまたは並列領域のコンテンツを互いに異なる作業チャンク用に並列実行します。すべての作業チャンクが完了すると、すべてのスレッドの同期がとられ、スレーブスレッドが消失し、マスタースレッドが処理を続行します。

並列化されたプログラムの実際の動作は、それほど直接的なものではありません。コンパイラが並列領域またはループ用のコード (または、その他の任意の OpenMP 構造) を生成するとき、それらの内部のコードが抽出され、Oracle 実装で *mfunction* と呼ばれる独立した関数が作成されます。この関数は、アウトライン関数、またはループ本体関数とも呼ばれます。mfunction の名前は、OpenMP 構造タイプ、抽出元となった関数の名前、その構造が置かれているソース行の行番号を符号化したものです。これらの関数の名前は、アナライザの上級モードとマシンモードでは次の形式で表示されます。ここで、大括弧内の名前は関数の実際のシンボルテーブル名です。

```
bardo_ -- OMP parallel region from line 9 [_$p1C9.bardo_]
atomsum_ -- MP doall from line 7 [_$d1A7.atomsum_]
```

これらの関数には、ほかのソース構造から生成される別の形式もあり、その場合、名前の中の「`OMP` 並列領域」は、「`MP` コンストラクト」、「`MP doall`」、「`OMP` 領域」のいずれかに置き換えられます。以降の説明では、これらすべてを総称して「並列領域」と言います。

並列ループ内のコードを実行する各スレッドは、mfunction を複数回呼び出すことができ、1 回呼び出すたびにループ内の 1 つの作業チャンクが実行されます。すべての作業チャンクが完了すると、それぞれのスレッドはライブラリ内の同期ルーチンまたは縮小ルーチンを呼び出します。その後、マスタースレッドが続行される一方、各スレーブスレッドはアイドル状態になり、マスタースレッドが次の並列領域に入るまで待機します。すべてのスケジューリングと同期は、OpenMP ランタイムの呼び出しによって処理されます。

並列領域内のコードは、その実行中、作業チャンクを実行しているか、ほかのスレッドとの同期をとっているか、行うべき追加の作業チャンクを取り出している場合があります。また、ほかの関数を呼び出す場合もあり、それによってさらに別の関数が呼び出される可能性もあります。並列領域内で実行されるスレーブスレッド(またはマスタースレッド)は、それ自体が、またはそれが呼び出す関数から、マスタースレッドとして動作し、独自の並列領域に入って入れ子並列を生成する場合があります。

アナライザは、呼び出しスタックの統計的な標本収集に基づいてデータを収集し、すべてのスレッドにまたがってデータを集計し、収集したデータのタイプに基づき、関数、呼び出し元と呼び出し先、ソース行、および命令を対象にパフォーマンスのメトリックスを表示します。アナライザは、OpenMP プログラムのパフォーマンスに関する情報を、ユーザーモード、上級モード、マシンモードという 3 つのモードのいずれかで提示します。

OpenMP プログラムのデータ収集についての詳細は、OpenMP のユーザーコミュニティー Web サイトの An OpenMP Runtime API for Profiling (`http://www.compunity.org/futures/omp-api.html`) を参照してください。

OpenMP プロファイルデータのユーザーモードの表示

プロファイルデータのユーザーモードの表示では、プログラムが実際に178 ページの「OpenMP ソフトウェア実行の概要」で説明されている直観的モデルに従って実行されたかのように情報が提示されます。マシンモードで表示される実際のデータは、ランタイムライブラリ `libmtsk.so` の実装の詳細を取り込んだもので、これは、モデルに対応していません。上級モードでは、モデルに合うように変更されたデータと、実際のデータが表示されます。

ユーザーモードでは、プロファイルデータの表示はモデルにさらに近くなるよう変更されるため、記録されたデータやマシンモードの表示と次の 3 つの点で異なっています。

- 擬似関数は、OpenMP ランタイムライブラリの視点から各スレッドの状態を表すように構築されます。
- 呼び出しスタックは、前述のように、コードの実行方法のモデルに対応するデータを報告するよう操作されます。
- 時間ベースのプロファイル実験用に、2 つの追加パフォーマンスメトリックスが構築され、それらは、有益な作業の実行に費やされた時間と、OpenMP ランタイム内での待機に費やされた時間に対応します。メトリックスは、OpenMP ワークメトリックスとOpenMP 待ちメトリックスです。
- OpenMP 3.0 プログラムでは、3 つ目のメトリックス、OpenMP オーバーヘッドメトリックスが構築されます。

擬似関数

擬似関数は、スレッドが OpenMP ランタイムライブラリ内でいずれかの状態にあったイベントを反映するために構築され、ユーザーモードおよび上級モード呼び出しスタック上に置かれます。

次の疑似関数が定義されています。

<OMP-overhead>	OpenMP ライブラリ内で実行中
<OMP-idle>	作業を待っているスレーブスレッド
<OMP-reduction>	縮約操作を実行中のスレッド
<OMP-implicit_barrier>	暗黙のバリアで待機中のスレッド
<OMP-explicit_barrier>	明示的なバリアで待機中のスレッド
<OMP-lock_wait>	ロックを待っているスレッド
<OMP-critical_section_wait>	critical セクションに入るのを待っているスレッド
<OMP-ordered_section_wait>	ordered セクションに入る順番を待っているスレッド
<OMP-atomic_section_wait>	OpenMP 原子構造で待機中のスレッド

スレッドが、擬似関数のいずれかに対応する OpenMP ランタイム状態にある場合、擬似関数がスタック上のリーフ関数として追加されます。スレッドの実際のリーフ関数は、OpenMP ランタイムのどこかに存在する場合には、リーフ関数として <OMP-overhead> に置き換えられます。そうでない場合、OpenMP ランタイムに入っているすべての PC は、ユーザーモードスタックから除外されます。

OpenMP 3.0 プログラムでは、<OMP-overhead> 擬似関数は使用されません。この擬似関数は、OpenMP オーバーヘッドメトリックスに置き換えられています。

ユーザーモード呼び出しスタック

OpenMP 実験の場合は、OpenMP を使用せずにプログラムがコンパイルされたときに取得されたものと同様の、再構築された呼び出しスタックが表示されます。目標は、実際の処理の詳細すべてを示すことではなく、プログラムを直観的に理解できるようにプロファイルデータを表示することです。OpenMP ランタイムライブラリが特定の操作を実行しているときは、マスタスレッドとスレーブスレッドの呼び出しスタックがまとめられ、呼び出しスタックに <OMP-*> 擬似関数が追加されます。

OpenMP メトリックス

OpenMP プログラムの時間プロファイルイベントを処理するときは、OpenMP システム内の 2 つの状態でそれぞれ費やされた時間に対応する 2 つのメトリックスが示されます。「OpenMP ワーク」と「OpenMP 待ち」です。

スレッドがユーザーコードから実行されたときは、逐次か並列かを問わず、「OpenMP ワーク」に時間が累積されます。スレッドが何かを待って先に進めないときは、その待機が busy-wait (spin-wait) であるかスリープ状態であるかを問わず、「OpenMP 待ち」に時間が累積されます。これら 2 つのメトリックスの合計は、時間プロファイル内の「合計 LWP 時間」メトリックに一致します。

OpenMP 待ちメトリックスとOpenMP ワークメトリックスは、ユーザーモード、上級モード、およびマシンモードに表示されます。

OpenMP プロファイリングデータの上級表示モード

上級表示モードで OpenMP 実験を確認する場合、OpenMP ランタイムが特定の操作を実行しているときの <OMP-*> 形式の擬似関数が、ユーザー表示モードと同じように表示されます。ただし、上級表示モードでは、並列化されたループ、タスクなどを表す、コンパイラにより生成された mfunction が別個に表示されます。ユーザーモードでは、コンパイラにより生成されたこれらの mfunction はユーザー関数に含められます。

OpenMP プロファイリングデータのマシン表示モード

「マシン」モードでは、すべてのスレッドのネイティブな呼び出しスタックと、コンパイラによって生成されたアウトライン関数が表示されます。

OpenMPプロファイルデータ、マシン表現 プログラムの実行のさまざまな局面における実際の呼び出しスタックは、前述の直感的なモデルに示したものとは大きく異なります。マシンモードでは、呼び出しスタックが測定どおりに表示され、変換は行われず、擬似関数も構築されません。ただし、時間プロファイルのメトリックスは依然として示されます。

次に示す各呼び出しスタックでは、`libmtsk` は OpenMP ランタイムライブラリ内の呼び出しスタックに入っている 1 つ以上のフレームを表しています。どの関数がどの順序で表示されるかの詳細は、バリア用のコードまたは縮小を行うコードの内部的な実装と同様に、OpenMP のリリースによって異なります。

1. 最初の並列領域の前

   最初の並列領域の前最初の並列領域に入る前の時点で存在するスレッドは、マスタースレッドただ 1 つだけです。呼び出しスタックはユーザーモードおよび上級モードの場合と同じです。

```
マスター

foo

main

_start
```

2. 並列領域内で実行中

マスター	スレーブ1	スレーブ2	スレーブ3
foo-OMP...			
libmtsk			
foo	foo-OMP...	foo-OMP...	foo-OMP...
main	libmtsk	libmtsk	libmtsk
_start	_lwp_start	_lwp_start	_lwp_start

マシンモードでは、スレーブスレッドはマスターが開始された _start 内ではなく、_lwp_start 内で開始されたものとして示されます。一部のバージョンのスレッドライブラリでは、この関数は _thread_start として表示されます。foo-OMP... の呼び出しは、並列領域に対して生成された mfunction を表します。

3. すべてのスレッドがバリアの位置にある

マスター	スレーブ1	スレーブ2	スレーブ3
libmtsk			
foo-OMP...			
foo	libmtsk	libmtsk	libmtsk
main	foo-OMP...	foo-OMP...	foo-OMP...
_start	_lwp_start	_lwp_start	_lwp_start

スレッドが並列領域内で実行されるときと異なり、スレッドがバリアの位置で待機しているときは、foo と並列領域コード foo-OMP... の間に OpenMP ランタイムからのフレームは存在しません。その理由は、実際の実行には OMP 並列領域関数が含まれていませんが、OpenMP ランタイムがレジスタを操作し、スタック展開で直前に実行された並列領域関数からランタイムバリアコードへの呼び出しが示されるようにするからです。そうしないと、どの並列領域がバリア呼び出しに関連しているかをマシンモードで判定する方法がなくなってしまいます。

4. 並列領域から出たあと

マスター	スレーブ1	スレーブ2	スレーブ3
foo			

マスター	スレーブ 1	スレーブ 2	スレーブ 3
main	libmtsk	libmtsk	libmtsk
_start	_lwp_start	_lwp_start	_lwp_start

スレーブスレッド内では、呼び出しスタック上にユーザーフレームが存在しません。

5. 入れ子の並列領域内にいるとき

マスター	スレーブ 1	スレーブ 2	スレーブ 3	スレーブ 4
	bar-OMP...			
foo-OMP...	libmtsk			
libmtsk	bar			
foo	foo-OMP...	foo-OMP...	foo-OMP...	bar-OMP...
main	libmtsk	libmtsk	libmtsk	libmtsk
_start	_lwp_start	_lwp_start	_lwp_start	_lwp_start

不完全なスタック展開

スタック展開の定義については、171 ページの「呼び出しスタックとプログラムの実行」を参照してください。

スタック展開は、次のようないくつかの場合に失敗します。

- ユーザーコードによってスタックが破壊されている場合。スタックが破壊された原因によっては、プログラムまたはデータ収集コードでコアダンプする場合があります。
- ユーザーコードが、関数呼び出しの標準の ABI 規則に従っていない場合。特に、SPARC プラットフォームで、保存命令が実行される前に復帰レジスタ %o7 が変えられる場合。

  どんなプラットフォームでも、手書きのアセンブラコードには規則違反がある場合があります。
- リーフ PC の関数内の位置が、呼び出し先のフレームがスタックからポップされた後、また関数が復帰する前である場合。
- 呼び出しスタックに約 250 を超えるフレームが含まれている場合、この呼び出しスタックを完全に展開するだけの領域がコレクタにはありません。この場合、呼び出しスタックの _start から特定の時点までの関数の PC は実験ファイルに記録されません。疑似関数 <Truncated-stack> は、記録された一番上のフレームを集計するため、<Total> から呼び出されたものとして示されます。

- x86 プラットフォームで、最適化された関数のフレームをコレクタが展開できなかった場合。

中間ファイル

-E または -P コンパイラオプションを使用して中間ファイルを生成すると、アナライザは元のソースファイルではなく、この中間ファイルを注釈付きソースコードとして使用します。-E を使用して生成された #line 指令は、ソース行へのメトリックスの割り当てで問題が発生する原因となることがあります。

関数が生成されるようにコンパイルされたソースファイルへの参照用の行番号を持たない関数からの命令が存在する場合、次の行が注釈付きのソースに現れます。

function_name -- <instructions without line numbers>

行番号は、次の条件下では欠落することがあります。

- -g オプションを指定しないでコンパイルした場合。
- コンパイルのあとにデバッグ情報がストリップされた場合、またはその情報を含む実行可能ファイルかオブジェクトファイルが移動、削除、変更された場合。
- 関数が、オリジナルのソースファイルではなく、#include ファイルから生成されたコードを含む場合。
- 高レベルの最適化で、コードが異なるファイルの関数からインライン化された場合。
- ソースファイルに、ほかのファイルを参照する #line 指令がある場合。たとえば、-E オプションを使用してコンパイルし、その結果の .i ファイルをコンパイルすることによって起こります。-P フラグを使用してコンパイルした際にも起こりえます。
- 行番号情報を読み取るオブジェクトファイルが見つからない場合。
- 使用したコンパイラが、不完全な行番号テーブルを生成する場合。

プログラム構造へのアドレスのマッピング

アナライザは、呼び出しスタックの内容を処理して PC 値を生成したあとに、それらの PC をプログラム内の共有オブジェクト、関数、ソース行、および逆アセンブリ行 (命令) にマップします。ここでは、これらのマッピングについて説明します。

プロセスイメージ

プログラムを実行すると、そのプログラムの実行可能ファイルからプロセスがインスタンス化されます。プロセスのアドレス空間には、実行可能な命令を表すテキス

トが存在する領域や、通常は実行されないデータが存在する領域などの多数の領域があります。通常、呼び出しスタックに記録される PC は、プログラムのいずれかのテキストセグメント内のアドレスに対応しています。

プロセスの先頭テキストセクションは、実行可能ファイルそのものから生成されます。先頭以外のテキストセクションは、プロセスの開始時に実行可能ファイルとともに読み込まれたか、プロセスによって動的に読み込まれた、共有オブジェクトに対応しています。呼び出しスタック内の PC は、呼び出しスタックの記録時に読み込まれた実行可能ファイルと共有オブジェクトに基づいて解決されます。実行可能ファイルと共有オブジェクトはよく似ているため、これらをまとめてロードオブジェクトと呼びます。

共有オブジェクトは、プログラムの実行途中で読み込みおよび読み込みの解除が可能なため、実行中のタイミングによって PC が対応する関数が異なることがあります。また、共有オブジェクトが読み込み解除されたあとに別のアドレスに再度読み込まれた場合は、異なる時点で異なる複数の PC が同じ関数に対応することもあります。

ロードオブジェクトと関数

実行可能ファイルまたは共有オブジェクトのどちらであっても、ロードオブジェクトには、コンパイラによって生成された命令を含むテキストセクション、データ用のデータセクション、および各種のシンボルテーブルが含まれます。ロードオブジェクトシンボルテーブルシンボルテーブル、ロードオブジェクト すべてのロードオブジェクトには、ELF シンボルテーブルが存在する必要があります。ELF シンボルテーブルには、そのオブジェクト内で大域的に既知の関数すべての名前とアドレスが含まれます。-g オプションを指定してコンパイルしたロードオブジェクトには、追加のシンボル情報が含まれます。この情報は、ELF シンボルテーブルを補足するもので、非大域的な関数に関する情報、関数の派生元のオブジェクトモジュールに関する補足情報、アドレスをソース行に関連付ける行番号情報で構成されます。

「関数」という用語は、ソースコードで記述された高度な操作を表す一連の命令を表します。この用語は、Fortran で使用されるサブルーチンや、C++ および Java プログラミング言語で使用されるメソッドなども表します。サブルーチン関数 サブルーチンFortranサブルーチンメソッド 関数関数はソースコードで明確に記述され、通常、その名前は、一群のアドレスを表すシンボルテーブル内に出現します。

原則として、ロードオブジェクトのテキストセグメント内のアドレスは関数にマップすることができます。関数ロードオブジェクト内のアドレス ロードオブジェクト関数のアドレス呼び出しスタック上のリーフ PC およびほかのすべての PC で、まったく同じマッピング情報が使用されます。関数の多くは、プログラムのソースモデルに直接対応します。以降の節では、そのような対応関係を持たない関数について説明します。

別名を持つ関数

一般に、関数は大域関数として定義されます。このことは、プログラム内のあらゆる部分で関数名が既知であることを意味します。大域関数の名前は、実行可能ファイル内で一意である必要があります。アドレス空間内に同一名の大域関数が複数存在する場合、実行時リンカーはすべての参照をそのうちの 1 つに決定します。その他の関数は実行されず、関数リストにそれらの関数が含まれることはありません。「概要」タブでは、選択した関数を含む共有オブジェクトおよびオブジェクトモジュールを調べることができます。

さまざまな状況で、同じ関数が異なる名前で認識されることがあります。この一般的な例としては、コードの同一部分に対して、いわゆる弱いシンボルと強いシンボルが使用されている場合があります。一般に、強い名前は対応する弱い名前と同じですが、前に下線 (_) が付きます。スレッドライブラリ内の多くの関数には、強い名前、弱い名前、代替内部シンボルに加えて、pthread および Solaris スレッド用に別の名前があります。いずれの場合も、アナライザの関数リストでは、このうちの 1 つの名前だけが使用されます。選択される名前は、アドレスをアルファベット順に並べた最後のシンボルです。ほとんどの場合は、この名前がユーザーの使用する名前に対応しています。「概要」タブでは、選択した関数のすべてのエイリアス (別名) が表示されます。

一意でない関数名

別名を持つ関数は、コードの同一部分に複数の名前があることを意味しますが、場合によっては、複数のコード部分で同じ名前が使用されることがあります。

- モジュール性を実現するために、関数が静的関数として定義されることがあります。これは、その関数名がプログラムの一部 (一般には、コンパイル済みの 1 つのオブジェクトモジュール) でだけ認識されることを意味します。このような場合、プログラムのまったく異なる部分を参照している同じ名前の複数の関数がアナライザに表示されます。「概要」タブでは、こうした関数を区別するために、それら関数のそれぞれにオブジェクトモジュール名が表示されます。また、こうした関数のどの名前が選択されたとしても、その関数のソース、逆アセンブリ、呼び出し元と呼び出し先を表示することができます。
- ライブラリ関数の弱い名前を持つラッパー関数または割り込み関数がプログラムで使用され、そのライブラリ関数の呼び出しに優先されることがあります。一部のラッパー関数は、ライブラリ内の元の関数を呼び出し、その場合は、名前の両方のインスタンスがアナライザの関数リストに表示されます。こうした関数は、元の共有オブジェクトやオブジェクトモジュールが異なるため、それらの情報を基に区別することができます。コレクタも一部のライブラリ関数をラップすることがあり、アナライザには、ラッパー関数と実際の関数の両方が表示されることがあります。

ストリップ済み共有ライブラリの静的関数

ライブラリ内では、ライブラリ内部の関数名がユーザーの使う関数名と衝突しないようにするために、静的関数がよく使用されます。ライブラリをストリップすると、静的関数の名前はシンボルテーブルから削除されます。このような場合、アナライザは、ストリップ済み静的関数を含むライブラリ内のすべてのテキスト領域ごとに擬似的な名前を生成します。この名前は `<static>@0x12345` という形式で、`@` 記号に続く文字列は、ライブラリ内のテキスト領域のオフセット位置を表します。アナライザは、連続する複数のストリップ済み静的関数と単一のストリップ済み静的関数を区別できないため、複数のストリップ済み静的関数のメトリックスがまとめて表示されることがあります。

ストリップ済み静的関数は、その PC が静的関数の保存命令のあとに表示されるリーフ PC である場合を除いて、正しい呼び出し元から呼び出されたように表示されます。シンボル情報がない場合、アナライザは保存アドレスを認識しません。このため、復帰レジスタを呼び出し元として使用すべきかどうかは判断できません。復帰レジスタは常に無視されます。複数の関数が、1 つの `<static>@0x12345` 関数にまとめられることがあるため、実際の呼び出し元または呼び出し先が隣接する関数と区別されないことがあります。

Fortran での代替エントリポイント

Fortran には、コードの一部に複数のエントリポイントを用意し、呼び出し元が関数の途中を呼び出す手段が用意されています。このようなコードをコンパイルしたときに生成されるコードは、メインのエントリポイントの導入部、代替エントリポイントの導入部、関数のコード本体で構成されます。各導入部では、関数があとで復帰するためのスタックが作成され、そのあとで、コード本体に分岐または接続します。

各エントリポイントの導入部のコードは、そのエントリポイント名を持つテキスト領域に常に対応しますが、サブルーチン本体のコードは、エントリポイント名の 1 つだけを受け取ります。受け取る名前は、コンパイラによって異なります。

多くの場合、導入部の時間はわずかで、アナライザに、サブルーチン本体に関連付けられたエントリポイント以外のエントリポイントに対応する関数が表示されることはほとんどありません。通常、代替エントリポイントを持つ Fortran サブルーチンで費やされる時間を表す呼び出しスタックは、導入部ではなくサブルーチンの本体に PC があり、本体に関連付けられた名前だけが呼び出し先として表示されます。同様に、そうしたサブルーチンからのあらゆる呼び出しは、サブルーチン本体に関連付けられている名前から行われたものとみなされます。

クローン生成関数

コンパイラは、通常以上の最適化が可能な関数への呼び出しを見分けることができます。こういった呼び出しの一例としては、引数の一部が定数である関数への呼び出しが挙げられます。コンパイラは、最適化できる特定の呼び出しを見つけると、クローンと呼ばれるこの関数のコピーを作成して、最適化コードを生成します。クローン関数名は、特定の呼び出しを識別する、符号化された名前です。アナライザはこの名前を復号化し、クローン生成関数のインスタンスそれぞれを別々に関数リストに表示します。クローン生成関数はそれぞれ別の命令セットを持っているので、注釈付き逆アセンブリリストには、クローン生成関数が別々に表示されます。クローン生成関数はそれぞれ別の命令セットを持っているので、注釈付き逆アセンブリリストにはクローン生成関数が別々に表示されます。各クローン生成関数のソースコードは同じであるため、注釈付きソースリストでは関数のすべてのコピーについてデータが集計されます。

インライン関数

インライン関数とは、実際に呼び出す代わりに、呼び出し位置にコンパイラが生成した命令が挿入される関数です。2 通りのインライン化があり、ともにパフォーマンス向上のために行われ、アナライザに影響します。

- C++ のインライン関数定義。このようにインライン化する理由は、関数呼び出しが、インライン化した関数よって行われる作業よりも処理時間がかかるためです。呼び出しの設定をするより、単に呼び出し位置に関数のコードを挿入する方が優れています。一般に、アクセス関数は、必要な命令が 1 つだけであることが多いため、インライン化対象として定義されます。`-g` オプションを使用してコンパイルすると、関数のインライン化は無効になります。一方、`-g0` を指定すると有効になり、これが推奨されます。
- 高レベルの最適化 (4 および 5) で行われる明示的または自動的なインライン化。明示的および自動的なインライン化は、`-g` オプションが有効なときにも行われます。この種のインライン化を行うのは、関数呼び出しの時間を節約するための場合もあります。しかし、多くの場合は、レジスタの利用や命令の実行スケジューリングを最適化できる命令数を増やすためです。

いずれのインライン化も、メトリックスの表示に同じ影響を及ぼします。ソースコードに記述されていて、インライン化された関数は、関数リストにも、また、そうした関数のインライン化先の関数の呼び出し先としても表示されません。通常ならば、インライン化された関数の呼び出し位置で包括的メトリックスとみなされるメトリックス (呼び出された関数で費やされた時間を表す) が、実際には呼び出し位置 (インライン化された関数の命令を表す) が原因の排他的メトリックスと報告されます。

注 – インライン化によってデータの解釈が難しくなることがあります。このため、パフォーマンス解析のためにプログラムをコンパイルするときには、インライン化を無効にすることを推奨します。

場合によっては、関数がインライン化されている場合も、いわゆるライン外関数が残されます。一部の呼び出し側ではライン外関数が呼び出され、それ以外では命令がインライン化されます。このような場合、関数は関数リストに含まれますが、関連するメトリックスはライン外呼び出しだけを表します。

コンパイラ生成の本体関数

関数内のループ、または並列化指令のある領域を並列化する場合、コンパイラは、元のソースコードに含まれていない新しい本体関数を作成します。こうした関数については、178 ページの「OpenMP ソフトウェア実行の概要」で説明しています。

アナライザでは、これらの関数は通常の関数として表示され、コンパイラにより生成される名前に加えて、抽出元の関数に基づいて名前が割り当てられます。これらの関数の排他的メトリックスおよび包括的メトリックスは、本体関数で費やされる時間を表します。また、構造の抽出元の関数は各本体関数の包括的メトリックスを示します。これが達成される方法については、178 ページの「OpenMP ソフトウェア実行の概要」で説明しています。

並列ループを含む関数をインライン化した場合、そのコンパイラ生成の本体関数名には、元の関数ではなく、インライン化先の関数の名前が反映されます。

注 – コンパイラ生成本体関数の名前は、-g オプションを指定してコンパイルされたモジュールでのみ復号化することができます。

アウトライン関数

フィードバック最適化コンパイルで、アウトライン関数が作成されることがあります。それらは、通常では実行されないコード、特に、最終的な最適化コンパイル用のフィードバックの生成に使用される「試験実行」の際に実行されないコードを表しています。一般的な例は、ライブラリ関数の戻り値でエラーチェックを実行するコードです。通常、エラー処理コードは実行されません。ページングと命令キャッシュの動作を向上させるため、こういったコードはアドレス空間の別の場所に移動され、新たな別の関数となります。アウトライン関数の名前は、コードの取り出し元関数の名前や特定のソースコードセクションの先頭の行番号を含む、アウトライン化したコードのセクションに関する情報をエンコードします。これらの符号化された名前は、リリースごとに異なります。アナライザは、読みやすい関数名を表示します。

アウトライン関数は実際には呼び出されるのではなく、ジャンプ先になります。動作をユーザーのソースコードモデルにより近づけるため、アナライザは、main 関数からそのアウトライン部分への擬似的な呼び出しを生成します。

アウトライン関数は、適切な包括的および排他的メトリックスを持つ通常の関数として表示されます。また、アウトライン関数のメトリックスは、アウトライン化元の関数の包括的メトリックスとして追加されます。

フィードバックデータを利用した最適化コンパイルの詳細は、『C ユーザーズガイド』の付録 B、『C++ ユーザーズガイド』の付録 A、または『Fortran ユーザーズガイド』の第 3 章で、`-xprofile` コンパイラオプションの説明を参照してください。

動的にコンパイルされる関数

動的にコンパイルされる関数は、プログラムの実行中にコンパイルされリンクされる関数です。コレクタ API 関数を使用して必要な情報をユーザーが提供しないかぎり、コレクタは C や C++ で記述された動的にコンパイルされる関数に関する情報を把握できません。API 関数については、52 ページの「動的な関数とモジュール」を参照してください。情報を提供しなかった場合、関数は `<Unknown>` としてパフォーマンス解析ツールに表示されます。

Java プログラムの場合、コレクタは Java HotSpot 仮想マシンによってコンパイルされるメソッドに関する情報を取得するので、API 関数を使用して情報を提供する必要がありません。ほかのメソッドの場合、パフォーマンスツールはメソッドを実行する JVM ソフトウェアの情報を表示します。Java ユーザー表現では、すべてのメソッドがインタプリタされたバージョンとマージされます。マシン表現では、HotSpot でコンパイルされたバージョンが個別に表示され、JVM 関数はインタプリタされたメソッドごとに表示されます。

`<Unknown>` 関数

特定の条件では、PC が既知の関数にマップされないことがあります。このような場合、PC は `<Unknown>` という名前の特別な関数にマップされます。

PC が `<Unknown>` にマップされるのは、次のような場合です。

- C または C++ で記述された関数が動的に生成され、この関数に関する情報がコレクタ API 関数によってコレクタに提供されない場合。コレクタ API 関数の詳細については、52 ページの「動的な関数とモジュール」を参照してください。
- Java メソッドが動的にコンパイルされるが、Java プロファイリングが無効である場合。

- PC が実行可能ファイルまたは共有オブジェクトのデータセクション内のアドレスに対応している場合。たとえば、SPARC V7 版の `libc.so` のデータセクションには、複数の関数 (`.mul`、`.div` など) があります。コードがデータセクションにあるため、SPARC V8 または SPARC V9 プラットフォームで動作していることをライブラリが検出したときに、動的に書き換えてマシン命令を利用できるようになります。
- 実験ファイルに記録されない実行可能ファイルのアドレス空間内の共有オブジェクトに PC が対応する場合。
- PC が既知のロードオブジェクト内に存在しない場合。もっとも考えられる原因は、展開に失敗して、PC 値として記録された値が PC ではなく、別のワードである場合です。PC が復帰レジスタのとき、既知のロードオブジェクト内に存在しないように見える場合は、`<Unknown>` 関数に割り当てられずに無視されます。
- コレクタにシンボリック情報がない JVM ソフトウェアの内部部分に PC がマップしている場合。

`<Unknown>` 関数の呼び出し元および呼び出し先は、呼び出しスタックの前および次の PC に対応しており、通常どおり処理されます。

OpenMP の特殊な関数

擬似関数は、スレッドが OpenMP ランタイムライブラリ内の何らかの状態にあったイベントを反映するために構築され、ユーザーモード呼び出しスタック上に置かれます。次の疑似関数が定義されています。

`<OMP-overhead>`	OpenMP ライブラリ内で実行中
`<OMP-idle>`	作業を待っているスレーブスレッド
`<OMP-reduction>`	縮約操作を実行中のスレッド
`<OMP-implicit_barrier>`	暗黙のバリアで待機中のスレッド
`<OMP-explicit_barrier>`	明示的なバリアで待機中のスレッド
`<OMP-lock_wait>`	ロックを待っているスレッド
`<OMP-critical_section_wait>`	critical セクションに入るのを待っているスレッド
`<OMP-ordered_section_wait>`	ordered セクションに入る順番を待っているスレッド

<JVM-System> 関数

ユーザー表現では、<JVM-System> 関数は、JVM ソフトウェアが Java プログラムの実行以外のアクションを行うために使用した時間を示します。JVM ソフトウェアは、ガベージコレクションや HotSpot コンパイルなどのタスクを、この時間間隔内で実行します。<JVM-System> はデフォルトで関数リストに表示されます。

<no Java callstack recorded> 関数

<no Java callstack recorded> 関数は <Unknown> 関数に似ていますが、Java スレッド用に、Java ユーザー表現でのみ使用されます。コレクタが Java スレッドからイベントを受信すると、ネイティブスタックを展開し、JVM ソフトウェアを呼び出して対応する Java スタックを取得します。その呼び出しが何らかの理由で失敗すると、擬似関数 <no Java callstack recorded> でアナライザ内にイベントが表示されます。JVM ソフトウェアが呼び出しスタックの報告を拒否する可能性があるのは、デッドロックを回避するためか、Java スタックを展開すると過剰な同期化が発生するときです。

<Truncated-stack> 関数

呼び出しスタックの個々の関数のメトリックスを記録するためにアナライザが使用するバッファーのサイズは制限されています。呼び出しスタックのサイズが大きくなってバッファーが満杯になった場合に、呼び出しスタックがそれ以上大きくなると、アナライザは関数のプロファイル情報を減らすようになります。ほとんどのプログラムでは、排他的 CPU 時間の大部分はリーフ関数に費やされるため、アナライザは、エントリ関数 _start() および main() を始めとするもっとも重要度の低いスタック下部の関数のメトリックスをドロップします。ドロップされた関数のメトリックスは、1 つの擬似関数 <Truncated-stack> にまとめられます。<Truncated-stack> 関数は、Java プログラムでも表示される場合があります。

<Total> 関数

<Total> 関数は、プログラム全体を表すために使用される擬似的な構成概念です。すべてのパフォーマンスメトリックスは、呼び出しスタック上の関数のメトリックスとして加算されるほかに、<Total> という特別な関数のメトリックスに加算されます。この関数は関数リストの先頭に表示され、そのデータを使用してほかの関数のデータの概略を見ることができます。呼び出し元 - 呼び出し先リストでは、任意のプログラム実行のメインスレッドにおける _start() の名目上の呼び出し元、また作成されたスレッドの _thread_start() の名目上の呼び出し元として表示されます。スタックの展開が不完全であった場合、<Total> 関数は、<Truncated-stack> の呼び出し元として表示される可能性があります。

ハードウェアカウンタオーバーフロープロファイルに関連する関数

次の関数は、ハードウェアカウンタオーバーフロープロファイルに関連します。

- `collector_not_program_related`: カウンタはプログラムに関連しません。
- `collector_hwcs_out_of_range`: カウンタは、オーバーフローシグナルを生成せずにオーバーフロー値を超えたように見えます。値が記録され、カウンタがリセットされます。
- `collector_hwcs_frozen`: カウンタは、オーバーフロー値を超えて停止されたように見えますが、オーバーフローシグナルが消失したように見えます。値が記録され、カウンタがリセットされます。
- `collector_hwc_ABORT`: 一般に特権付きプロセスがカウンタの制御権を取得したときに、ハードウェアカウンタの読み取りに失敗し、ハードウェアカウンタの収集が終了しました。
- `collector_record_counter`: ハードウェアカウンタイベントの処理中および記録中に蓄積されたカウントで、ハードウェアカウンタオーバーフロープロファイルのオーバーヘッドの一部を占めます。このカウントが `<Total>` カウントの大きな割合を占める場合は、オーバーフロー間隔を増やすこと (すなわち、より低い分解能の構成) が推奨されます。

インデックスオブジェクトへのパフォーマンスデータのマッピング

インデックスオブジェクトは、各パケットに記録されたデータからインデックスを計算できる要素のセットを表します。事前定義されているインデックスオブジェクトセットは、スレッド、CPU、標本、秒です。その他のインデックスオブジェクトは、`er_print indxobj_define` コマンドから直接実行するか、`.er.rc` ファイル内で定義できます。アナライザでインデックスオブジェクトを定義するには、「表示」メニューから「データ表示方法を設定」を選択し、「タブ」タブを選択し、「カスタムインデックスオブジェクトを追加」ボタンをクリックします。

パケットごとにインデックスが計算され、パケットに関連付けられているメトリックスが、そのインデックスのインデックスオブジェクトに追加されます。インデックス `-1` は、`<Unknown>` インデックスオブジェクトにマップしています。インデックスオブジェクトの階層表示は意味がないので、インデックスオブジェクトのメトリックスはすべて排他的メトリックスです。

プログラムデータオブジェクトへのデータアドレスのマッピング

メモリー演算に対応するハードウェアカウンタイベントからの PC が、原因と思われるメモリー参照命令に正しくバックトラックするように処理されると、アナライザは、コンパイラからハードウェアプロファイルサポート情報内に提供された命令識別子と記述子を使用して、関連するプログラムデータオブジェクトを生成します。

データオブジェクトという用語は、ソースコードに記述されているプログラムの定数、変数、配列、および構造体や共用体などの集合体のほか、別個の集合体要素を示す場合に使用します。データオブジェクトのタイプとそのサイズはソース言語によって異なります。多くのデータオブジェクトの名前は明示的にソースプログラム内で付けられますが、名前が付けられないものもあります。データオブジェクトの中には、ほかの単純なデータオブジェクトから生成または集計され、より複雑なデータオブジェクトの集合になるものもあります。

各データオブジェクトは、1 つのスコープに関連付けられています。スコープとは、そのオブジェクトが定義され、そのオブジェクトを参照できるソースプログラムの領域のことで、大域 (ロードオブジェクトなど)、特定のコンパイルユニット (オブジェクトファイル)、または関数のいずれかになります。同一のデータオブジェクトを異なるスコープで定義したり、特定のデータオブジェクトを異なるスコープで異なる方法で参照することができます。

バックトラッキングを有効にして収集された、メモリー操作に関するハードウェアカウンタイベントからのデータ派生メトリックスは、関連するプログラムのデータオブジェクトタイプに属するものとされ、そのデータオブジェクトを含む集合体と、<Unknown> や <Scalars> などすべてのデータオブジェクトを含むと見なされる <Total> 擬似データオブジェクトに伝搬します。<Unknown> の各種サブタイプは、<Unknown> の集合体まで伝搬します。次の節では、<Total>、<Scalars>、および <Unknown> の各データオブジェクトについて説明します。

データオブジェクト記述子

データオブジェクトは、宣言された型と名前の組み合わせで記述します。単純なスカラーデータオブジェクト {int i} は、int 型の変数 i を記述しているのに対して、{const+pointer+int p} は、int 型 p への定数ポインタを記述しています。型名のスペースは「_」(アンダースコア) に置き換えられ、名前の付いていないデータオブジェクトは「-」(ハイフン)、たとえば {double_precision_complex -} という名前で表されます。

集合体全体も同様に、foo_t 型の構造体の場合は {structure:foo_t} と表します。集合体の要素では、その要素のコンテナを追加指定する必要があります。たとえば、前述の foo_t 型の構造体のメンバーである int 型の i の場合は

{structure:foo_t}.{int i} となります。集合体はそれ自体、(さらに大きい) 集合体の要素になることも可能で、対応する記述子は集合体記述子を連結したもの、最終的にはスカラー記述子になります。

完全修飾された記述子は、データオブジェクトを明確にするために必ずしも必要ではありませんが、データオブジェクトの識別を支援するために一般的な完全指定を示します。

<Total> データオブジェクト

<Total> データオブジェクトは、プログラムのデータオブジェクト全体を表すために使用される擬似的な構造です。あらゆるパフォーマンスメトリックスは、異なるデータオブジェクト (およびそのオブジェクトが属する集合体) のメトリックスとして加算されるほかに、<Total> という特別なデータオブジェクトに加算されます。このデータオブジェクトはデータオブジェクトリストの先頭に表示され、そのデータを使用してほかのデータオブジェクトのデータの概略を見ることができます。

<Scalars> データオブジェクト

集合体要素のパフォーマンスメトリックスは、関連する集合体のメトリック値に加算されますが、すべてのスカラー定数および変数のパフォーマンスメトリックスは擬似的な <Scalars> データオブジェクトのメトリック値にさらに加算されます。

<Unknown> データオブジェクトとその要素

さまざまな状況下で、特定のデータオブジェクトにイベントデータをマップできない場合があります。このような場合、データは <Unknown> という特別なデータオブジェクトと、次に説明するいずれかの要素にマップされます。

- トリガー PC を持つモジュールが -xhwcprof を使用してコンパイルされていない

  トリガー PC を持つモジュールが -xhwcprof を使用してコンパイルされていない イベントの原因となっている命令またはデータオブジェクトは、オブジェクトコードがハードウェアカウンタプロファイルサポートを指定してコンパイルされていなかったので、識別されませんでした。

- バックトラッキングで有効な分岐先が検出できなかった

  バックトラッキングで有効な分岐先を検出できなかったイベントの原因となっている命令は、コンパイルオブジェクト内で提供されたハードウェアプロファイルサポート情報が、バックトラッキングの妥当性を検証するには不十分だったため、識別されませんでした。

- バックトラッキングで分岐先をトラバースした

  バックトラッキングで分岐先をトラバースしたイベントの原因となっている命令またはデータオブジェクトは、バックトラッキングで命令ストリーム内から制御転送ターゲットが検出されたため、識別されませんでした。

- 識別する記述子がコンパイラから提供されなかった

識別する記述子がコンパイラから提供されなかったバックトラッキングで原因と思われるメモリー参照命令を判別しましたが、それに関連するデータオブジェクトはコンパイラで指定されませんでした。

- タイプ情報がない

  タイプ情報がないバックトラッキングでイベントの原因と思われる命令を判別しましたが、その命令は、コンパイラによってメモリー参照命令として識別されませんでした。

- コンパイラが提供したシンボリック情報から判別不能

  コンパイラが提供したシンボリック情報から判別不能バックトラッキングで原因と思われるメモリー参照命令を判別しましたが、その命令はコンパイラによって識別されなかったため、それに関連するデータオブジェクトも判別できません。コンパイラのテンポラリは一般に識別されません。

- ジャンプ命令または呼び出し命令によってバックトラッキングが阻止された

  バックトラッキングが、ジャンプ命令または呼び出し命令によって阻止されたイベントの原因となっている命令は、バックトラッキングで命令ストリームの中から分岐命令または呼び出し命令が検出されたため、識別されませんでした。

- バックトラッキングでトリガー PC が検出されなかった

  バックトラッキングでトリガー PC が検出されなかったイベントの原因である命令を、最大のバックトラッキング範囲内から検出できませんでした。

- トリガー命令のあとでレジスタが変更されたため、仮想アドレスを特定できなかった

  トリガー命令のあとでレジスタが変更されたため、仮想アドレスを判別できなかったレジスタがハードウェアカウンタのスキッド中に上書きされたため、データオブジェクトの仮想アドレスを判別できませんでした。

- メモリー参照命令で有効な仮想アドレスが指定されなかった

  メモリー参照命令で有効な仮想アドレスが指定されなかった データオブジェクトの仮想アドレスが有効であるように見えませんでした。

メモリーオブジェクトへのパフォーマンスデータのマッピング

メモリーオブジェクトは、キャッシュ行、ページ、およびメモリーバンクなど、メモリーサブシステム内のコンポーネントです。このオブジェクトは、記録された仮想アドレスや物理アドレスから計算されたインデックスから決定されます。メモリーオブジェクトは、仮想ページおよび物理ページについて 8K バイト、64K バイト、512K バイト、および 4M バイトのサイズで事前定義されています。それ以外は、`er_print` ユーティリティーで `mobj_define` コマンドによって定義できます。また、アナライザの「メモリーオブジェクトを追加」ダイアログボックスを使用して

カスタムメモリーオブジェクトを定義することもできます。このダイアログボックスを開くには、「データ表示方法の設定」ダイアログボックスで「カスタムオブジェクトを追加」ボタンをクリックします。

◆ ◆ ◆ 第 7 章

7 注釈付きソースと逆アセンブリデータについて

注釈付きソースコードと注釈付き逆アセンブリコードは、関数内のパフォーマンス低下の原因になっているソース行または命令を特定するのに役立ち、コンパイラがコードに対して行なった変換処理に関するコメントを表示します。ここでは、注釈の処理と、注釈付きコードを解釈する上での問題点をいくつか説明します。

この章では、次の内容について説明します。

注釈付きソースコード

実験の注釈付きソースコードは、パフォーマンスアナライザで「アナライザ」ウィンドウの左の区画にある「ソース」タブを選択することで表示できます。または、実験を実行しなくても、`er_src` ユーティリティーを使用して注釈付きソースコードを表示できます。ここでは、パフォーマンスアナライザでソースコードを表示する方法について説明します。`er_src` ユーティリティーを使用した注釈付きソースコードの表示の詳細は、217 ページの「実験なしのソース/逆アセンブリの表示」を参照してください。

アナライザの注釈付きソースには、次の情報が含まれています。

- 元のソースファイルの内容
- 実行可能ソースコードの各行のパフォーマンスメトリックス
- 特定のしきい値を超えたメトリックスを含むコード行の強調表示
- インデックス行
- コンパイラのコメント

パフォーマンスアナライザの「ソース」タブのレイアウト

「ソース」タブは、いくつかの列に分かれています。ウィンドウの左側には個々のメトリックスを表示する固定幅の列が表示されます。右側の残りの列には、注釈付きソースが表示されます。

元のソース行の識別

注釈付きソースで黒字で表示されるすべての行は、元のソースファイルから取得されたものです。注釈付きソース列の各行の先頭の番号は、元のソースファイルの行番号に対応します。別の色で文字が表示されている行は、インデックス行かコンパイラのコメント行です。

「ソース」タブのインデックス行

ソースファイルとは、オブジェクトファイルを生成するためにコンパイルされるか、またはバイトコードに解釈されるファイルのことです。オブジェクトファイルには通常、ソースコード内の関数、サブルーチン、またはメソッドに対応する実行可能コードの領域が 1 つ以上含まれています。アナライザはオブジェクトファイルを解析して、それぞれの実行可能領域を関数として識別します。そして、オブジェクトコード内で見つけた関数を、オブジェクトコードに関連付けられたソースファイル内の関数、ルーチン、サブルーチン、またはメソッドにマップしようとします。アナライザは解析が成功すると、注釈付きソースファイル内の、オブジェクトコードで検出された関数の最初の命令に対応する場所に、インデックス行を追加します。

注釈付きソースは、「関数」タブのリストにインライン関数が表示されていない場合でも、インライン関数を含むすべての関数のインデックス行を表示します。「ソース」タブには、インデックス行が赤いイタリック体で、テキストが山括弧内に示されます。もっとも単純な種類のインデックス行は、関数のデフォルトコンテキストに対応しています。関数のデフォルトソースコンテキストは、その関数の最初の命令が帰するソースファイルとして定義されます。次の例は、C 関数 `icputime` のインデックス行を示しています。

```
                  578. int
                  579. icputime(int k)
0.       0.       580. {
                       <Function: icputime>
```

この例で分かるように、インデックス行は最初の命令に続く行に表示されます。C ソースの場合は、最初の命令は、関数本体の先頭が開く括弧に対応します。Fortran ソースの場合は、各サブルーチンのインデックス行が、`subroutine` キーワードを含む行に続きます。また、次の例に示すように、`main` 関数のインデックス行が、アプリケーションの起動時に実行される最初の Fortran ソース命令に続きます。

```
                                1. ! Copyright (c) 2006, 2010, Oracle and/or its affiliates. All Rights Reserv
                                2. ! @(#)omptest.f 1.11 10/03/24 SMI
                                3. ! Synthetic f90 program, used for testing openmp directives and the
                                4. !       analyzer
                                5.
0.      0.      0.      0.      6.        program omptest
                                   <Function: MAIN>
                                7.
                                8. !$PRAGMA C (gethrtime, gethrvtime)
```

場合によっては、アナライザは、オブジェクトコード内に見つけた関数を、そのオブジェクトコードに関連付けられたソースファイル内のプログラミング命令を使ってマップできないことがあります。たとえば、ヘッダーファイルのように、コードが `#include` されている場合や、ほかのファイルからインライン化されている場合があります。

また、特別なインデックス行やコンパイラのコメントではない特殊な行は、赤で示されます。たとえば、コンパイラの最適化の結果、ソースファイルに記述されているコードに対応しないオブジェクトコード内の関数について、特別なインデックス行が作成される場合があります。詳細は、211 ページの「「ソース」タブ、「逆アセンブリ」タブ、「PC」タブの特別な行」を参照してください。

コンパイラのコメント

コンパイラのコメントは、コンパイラによって最適化されたコードがどのように生成されたかを示します。インデックス行や元のソース行と区別できるように、コンパイラのコメント行は青く表示されます。コンパイラのさまざまな段階で、実行可能ファイルにコメントが挿入されることがあります。各コメントは、ソースの特定の行に関連付けられます。注釈付きソースの書き込み時には、ソース行に対してコンパイラが生成するコメントが、ソース行の直前に挿入されます。

コンパイラのコメントは、最適化するためにソースコードに対して行われた変換の大部分に関する情報を提供します。こうした変換には、ループの最適化、並列化、インライン化、パイプライン化などがあります。次に、コンパイラのコメントの例を示します。

```
0.      0.      0.      0.       28.       SUBROUTINE dgemv_g2 (transa, m, n, alpha, b, ldb,   &
                                  29.      &                    c, incc, beta, a, inca)
                                  30.       CHARACTER (KIND=1) :: transa
                                  31.       INTEGER   (KIND=4) :: m, n, incc, inca, ldb
                                  32.       REAL      (KIND=8) :: alpha, beta
                                  33.       REAL      (KIND=8) :: a(1:m), b(1:ldb,1:n), c(1:n)
                                  34.       INTEGER            :: i, j
                                  35.       REAL      (KIND=8) :: tmr, wtime, tmrend
                                  36.       COMMON/timer/ tmr
                                  37.
                            Function wtime_ not inlined because the compiler has not seen
                            the body of the routine
0.      0.      0.      0.       38.       tmrend = tmr + wtime()
```

```
                              Function wtime_ not inlined because the compiler has not seen
                              the body of the routine
                              Discovered loop below has tag L16
0.       0.       0.       0.        39.       DO WHILE(wtime() < tmrend)

                              Array statement below generated loop L4
0.       0.       0.       0.        40.       a(1:m) = 0.0
                                      41.

                              Source loop below has tag L6
0.       0.       0.       0.        42.       DO j = 1, n       ! <=-----\ swapped loop indices

                              Source loop below has tag L5
                              L5 cloned for unrolling-epilog.  Clone is L19
                              All 8 copies of L19 are fused together as part of unroll and jam
                              L19 scheduled with steady-state cycle count = 9
                              L19 unrolled 4 times
                              L19 has 9 loads, 1 stores, 8 prefetches, 8 FPadds,
                              8 FPmuls, and 0 FPdivs per iteration
                              L19 has 0 int-loads, 0 int-stores, 11 alu-ops, 0 muls,
                              0 int-divs and 0 shifts per iteration
                              L5 scheduled with steady-state cycle count = 2
                              L5 unrolled 4 times
                              L5 has 2 loads, 1 stores, 1 prefetches, 1 FPadds, 1 FPmuls,
                              and 0 FPdivs per iteration
                              L5 has 0 int-loads, 0 int-stores, 4 alu-ops, 0 muls,
                              0 int-divs and 0 shifts per iteration
0.210    0.210    0.210    0.        43.          DO i = 1, m
4.003    4.003    4.003    0.050     44.             a(i) = a(i) + b(i,j) * c(j)
0.240    0.240    0.240    0.        45.          END DO
0.       0.       0.       0.        46.       END DO
                                      47.       END DO
                                      48.
0.       0.       0.       0.        49.       RETURN
0.       0.       0.       0.        50.       END
```

「ソース」タブに表示されるコンパイラコメントの種類は、「データ表示方法の設定」ダイアログボックスの「ソース/逆アセンブリ」タブを使って設定できます。詳細は、112 ページの「データ表示オプションの設定」を参照してください。

共通部分式の削除

非常に一般的な最適化の例として、1 つの式が複数の場所に存在するときに、この式のコードを 1 つの場所にまとめることによってパフォーマンスを向上させることができます。たとえば、コードブロックの `if` と `else` の分岐の両方で同じ演算が記述されている場合、コンパイラはその演算を `if` 文の直前に移動することができます。実際にそのようにした場合、コンパイラは以前出現した一方の式に基づいて、命令に行番号を割り当てます。割り当てられた行番号が `if` 構造の分岐の 1 つに対応していて、実際にはもう一方の分岐が常に実行される場合、注釈付きソースでは、実行されない分岐内の行のメトリックスが表示されます。

ループの最適化

コンパイラは、数種類のループの最適化を行うことができます。一般的なものを次に示します。

- ループの展開
- ループのピーリング
- ループの入れ換え
- ループの分散
- ループの融合

ループの展開では、ループ本体内でループを数回反復し、それに応じてループインデックスを調整します。ループの本体が大きくなるほど、コンパイラはより効率的に命令をスケジュールできます。また、ループインデックスの増分や条件検査操作によるオーバーヘッドが減少します。残りのループは、ループのピーリングを使って処理されます。

ループのピーリングでは、ループから多数のループの反復を取り除き、これらをループの前か後ろに適宜移動します。

ループの入れ換えは、メモリーのストライドを最小限に抑えてキャッシュ行のヒット率を最大限に上げるために、入れ子のループの順序を変更します。

ループの融合は、隣り合ったループや近接したループを 1 つのループにまとめます。ループの融合からは、ループの展開と同じような利点がもたらされます。さらに、最適化済みの 2 つのループで共通のデータにアクセスする場合は、ループの融合によってループのキャッシュの局所性が改善されて、コンパイラは命令レベルの並列化機能をさらに活用することが可能になります。

ループの分散はループの融合の反対で、ループは複数のループに分割されます。この最適化は、ループ内の計算回数が過度に多くなり、パフォーマンス低下の原因となるレジスタのスピルが発生する場合に適しています。また、ループの分裂は、ループに条件文が含まれている場合にも有効です。場合によっては、条件文を含むものと含まないものの 2 つにループを分割できます。これによって、条件文を含まないループにおけるソフトウェアのパイプライン化の機会が増えます。

場合によって、入れ子のループでは、コンパイラはループの分裂を適用してループを分割し、次にループの融合を実行して異なる方法でループをまとめ直すことでパフォーマンスを改善します。この場合は、次のようなコンパイラのコメントが表示されます。

```
Loop below fissioned into 2 loops
Loop below fused with loop on line 116
[116]    for (i=0;i<nvtxs;i++) {
```

関数のインライン化

インライン関数を使用して、コンパイラは、実際の関数呼び出しを行う代わりに、関数が呼び出されている場所に関数の命令を直接挿入します。つまり、C/C++ マクロと同様に、それぞれの呼び出し場所でインライン関数の命令の複製が作成されます。コンパイラは、高レベルの最適化 (4 および 5) で明示的または自動的なインライン化を実行します。インライン化によって関数呼び出しの負荷が減り、レジスタの使用や命令のスケジュールを最適化するための命令がさらに提供されますが、その代わりに、コードのメモリー使用量が多くなります。次に、コンパイラコメントのインライン化の例を示します。

```
                Function initgraph inlined from source file ptralias.c
                    into the code for the following line
0.       0.         44.       initgraph(rows);
```

注 – コンパイラのコメントは、アナライザの「ソース」タブ内で 2 行にまたがって折り返されることはありません。

並列化

Sun、Cray、または OpenMP の並列化指令が含まれているコードの場合、複数プロセッサ上での並列実行用にコンパイルできます。コンパイラのコメントは、並列化が実行されている場所と実行されていない場所、およびその理由を示します。次に、並列化コンピュータのコメントの例を示します。

```
0.       6.324       9. c$omp  parallel do shared(a,b,c,n) private(i,j,k)
                    Loop below parallelized by explicit user directive
                    Loop below interchanged with loop on line 12
0.010    0.010      [10]            do i = 2, n-1

                    Loop below not parallelized because it was nested in a parallel loop
                    Loop below interchanged with loop on line 12
0.170    0.170       11.               do j = 2, i
```

並列実行とコンパイラ生成の本体関数の詳細は、178 ページの「OpenMP ソフトウェア実行の概要」を参照してください。

注釈付きソースの特別な行

「ソース」タブには、特殊な場合のためのほかの注釈を表示できます。これらの注釈は、コンパイラのコメントの形で、またはインデックス行と同じ色で特別な行に表示できます。詳細は、211 ページの「「ソース」タブ、「逆アセンブリ」タブ、「PC」タブの特別な行」を参照してください。

ソース行メトリックス

実行可能コードの各行のソースコードメトリックスは、固定幅の列に表示されます。メトリックスは、関数リストのものと同じです。実験のデフォルト値は、.er.rc ファイルを使って変更できます。詳細は、149 ページの「デフォルト値を設定するコマンド」を参照してください。また、表示されるメトリックスとしきい値の強調表示も、「データ表示方法の設定」ダイアログボックスを使ってアナライザで変更できます。詳細は、112 ページの「データ表示オプションの設定」を参照してください。

注釈付きソースコードは、ソース行レベルでのアプリケーションのメトリックスを示します。注釈付きソースは、アプリケーションの呼び出しスタックに記録された PC (プログラムカウント) を読み取り、各 PC をソース行にマッピングすることによって作成されます。注釈付きソースファイルを作成するために、アナライザは、最初に特定のオブジェクトモジュール (.o ファイル) またはロードオブジェクト内に生成されたすべての関数を特定し、各関数のすべての PC のデータをスキャンします。注釈付きソースを作成するには、アナライザが、すべてのオブジェクトモジュールまたはロードオブジェクトを検出して読み取り、PC からソース行へのマッピングを特定できる必要があります。また、表示するソースファイルを読み取って、注釈付きのコピーを作成できる必要もあります。アナライザはソースファイル、オブジェクトファイル、および実行可能ファイルを次のデフォルトの場所で順に検索し、正しいベース名のファイルが見つかると検索を停止します。

- 実験の保管ディレクトリ
- 現在の作業ディレクトリ
- 実行可能ファイルまたはコンパイルオブジェクトに記録されている絶対パス名

デフォルト値は、addpath または setpath 指令により、またはアナライザの「データ表示方法の設定」ダイアログボックスで変更できます。

addpath または setpath によって設定されたパスのリストを使用してファイルを検出できない場合は、pathmap コマンドを使用して 1 つ以上のパス再マッピングを指定できます。pathmap コマンドでは、*old-prefix* と *new-prefix* を指定できます。ソースファイル、オブジェクトファイル、または共有オブジェクトのパス名が *old-prefix* で指定した接頭辞で始まる場合、古い接頭辞は *new-prefix* で指定した接頭辞に置き換えられます。結果のパスは、ファイルの検索に使用されます。複数の pathmap コマンドが提供されており、それぞれが、ファイルが見つかるまで試行されます。

コンパイル処理では、要求される最適化レベルに応じて多くの段階があり、変換によって命令とソース行のマッピングに混乱が生じることがあります。最適化によっては、ソース行の情報が完全に失われたり、混乱が生じたりすることがあります。コンパイラは、さまざまな発見手法によって命令のソース行を追跡しますが、こうした手法は絶対ではありません。

ソース行メトリックスの解釈

命令のメトリックスについては、実行対象の命令を待っている間に発生したメトリックスとして解釈する必要があります。イベントが記録されるときに実行中である命令がリーフ PC と同じソース行に存在している場合、メトリックスはこのソース行を実行した結果であると解釈できます。ただし、実行中の命令とリーフ PC が存在しているソース行がそれぞれ異なる場合、リーフ PC が存在しているソース行のメトリックスの少なくとも一部は、実行中命令のソース行が実行待ちしていた間に集計されたメトリックスであると解釈する必要があります。この一例としては、1 つのソース行で計算された値が次のソース行で使用される場合が挙げられます。

メトリックスの解釈方法がもっとも問題となるのは、キャッシュミスやリソース待ち行列ストールなど、実行が大幅に遅延している場合や、命令が直前の命令の結果を待っている場合です。このような場合、ソース行のメトリックスが異常に高く見えることがあります。コード内のほかのソース行を調べて、こういった高メトリック値の原因である行を突きとめてください。

メトリックの形式

表 7–1 に、注釈付きソースコードの行に表示可能な 4 種類のメトリックスを示します。

表 7–1 注釈付きソースコードのメトリックス

メトリック	意味
(空白)	プログラムに、このコード行に対応する PC が存在しません。コメント行は常にこの空白になります。また、次の場合の見かけ上のコード行も空白になります。 ■ 最適化中に、見かけ上のコード部分のすべての命令が削除されている。 ■ コードが別の場所で繰り返されていて、コンパイラによって共通する部分式が認識され、その行のすべての命令に繰り返し部分の行番号が付けられている。 ■ コンパイラによって、その行の命令に不正な行番号が付けられている。
`0.`	プログラム内のいくつかの PC がこの行から派生したものとしてタグ付けされていますが、それらの PC を参照しているデータがありません。統計的にサンプリングされたかトレースされた呼び出しスタックに、そのような PC は存在しません。`0.` メトリックは、その行が実行されなかったことを意味するのではなく、プロファイリングデータパケットや記録されたトレースデータパケットに統計として表示されなかったことだけを意味します。
`0.000`	この行の少なくとも 1 つの PC がデータに表れていますが、メトリック値の計算でゼロに丸められました。
`1.234`	この行に属するすべての PC のメトリックスの合計が、表示されているゼロ以外の数値になりました。

注釈付き逆アセンブリコード

注釈付き逆アセンブリは、関数またはオブジェクトモジュールの命令のアセンブリコードのリストを提供します。このリストには、各命令に関連付けられているパフォーマンスメトリックスが含まれています。注釈付き逆アセンブリは複数の方法で表示することができ、どの方法で表示されるかは、行番号のマッピング情報およびソースファイルが存在するかどうか、また注釈付き逆アセンブリが要求されている関数のオブジェクトモジュールが既知かどうかによって決まります。

- オブジェクトモジュールが既知でない場合、アナライザによって単に指定された関数の命令が逆アセンブルされ、逆アセンブリでソース行は表示されません。
- オブジェクトモジュールが既知の場合、オブジェクトモジュール内のすべての関数が逆アセンブルされます。
- ソースファイルが利用可能で、行番号データが記録されている場合、アナライザによって、表示方式に応じてソースと逆アセンブリコードがインタリーブされます。
- コンパイラによってオブジェクトコードにコメントが挿入されている場合、対応する表示方式が設定されていれば、逆アセンブリでそれらのコメントもインタリーブされます。

逆アセンブリコードの各命令には、注釈として次の情報が付けられます。

- コンパイラによって報告されたソース行番号
- 相対アドレス
- 命令の 16 進表現 (要求があった場合)
- 命令のアセンブラの ASCII 表現

呼び出しアドレスの解決が可能な場合、それらのアドレスは関数名などのシンボルに変換されます。メトリックスは、命令行について表示されます。対応する表示方式が設定されていれば、インタリーブされるソースコードについても表示することができます。表示可能なメトリック値は、表 7–1 に示されているソースコードの注釈で説明しているとおりです。

複数の場所で `#include` が指定されているコードの逆アセンブリリストでは、そのコードが `#include` されている回数だけ逆アセンブリ命令が繰り返されます。ソースコードは、ファイル内で最初に出現する逆アセンブリコードの繰り返しブロックのみインタリーブされます。たとえば、`inc_body.h` というヘッダーに定義されているコードブロックが、`inc_body`、`inc_entry`、`inc_middle`、および `inc_exit` という 4 つの関数によって `#include` されている場合、逆アセンブリ命令のブロックは `inc_body.h` の逆アセンブリリストに 4 回現れますが、ソースコードは逆アセンブリ命令の 4 つのブロックの最初のもののみインタリーブされます。「ソース」タブに切り替えると、逆アセンブリコードの繰り返しにそれぞれ対応するインデックス行が表示されます。

インデックス行は「逆アセンブリ」タブ内に表示される場合があります。「ソース」タブの場合とは異なり、これらのインデックス行を直接ナビゲーションの目的に使用することはできません。ただし、インデックス行の直下の命令の 1 つにカーソルを置いて「ソース」タブを選択すると、インデックス行で参照されているファイルに移動できます。

ほかのファイルのコードを `#include` するファイルでは、ソースコードのインタリーブなしで、インクルードされたコードが逆アセンブリ命令として表示されます。ただし、これらの命令の 1 つにカーソルを置いて「ソース」タブを選択すると、`#include` されているコードを含むファイルが開かれます。このファイルを表示した状態で「逆アセンブリ」タブを選択すると、インタリーブされたソースコードとともに逆アセンブリコードが表示されます。

インライン関数の場合はソースコードと逆アセンブリコードをインタリーブできますが、マクロの場合はできません。

コードが最適化されていない場合、各命令の行番号は逐次順であり、ソース行と逆アセンブリされた命令は予想どおりにインタリーブされます。最適化されている場合は、あとの行の命令が前の行の命令よりも前に表示されることがあります。アナライザのインタリーブアルゴリズムでは、命令が行 *N* にあると表示される場合は、常に、行 *N* とその行までのすべてのソース行がその命令の前に挿入されます。最適化を行なった結果、制御転送命令とその遅延スロット命令の間にソースコードが現れます。ソース行の *N* に関連するコンパイラのコメントは、その行の直前に挿入されます。

注釈付き逆アセンブリの解釈

注釈付き逆アセンブリを解釈するのは簡単ではありません。リーフ PC とは、次に実行する命令のアドレスです。このため、命令の属性メトリックスは、命令の実行待ちに費やされた時間とみなされます。ただし、命令の実行は必ずしも順に行われるわけではなく、呼び出しスタックの記録に遅延があることもあります。注釈付き逆アセンブリコードを利用するには、実験の記録先であるハードウェアと、そのハードウェアが命令を読み込み、実行する方法を理解しておいてください。

次では、注釈付き逆アセンブリコードを解釈する上での問題点をいくつか説明します。

命令発行時のグループ化

命令は、命令発行グループと呼ばれるグループ単位で読み込まれ、発行されます。グループに含まれる命令は、ハードウェア、命令の種類、すでに実行された命令、およびほかの命令またはレジスタの依存関係によって異なります。その結果、ある命令が常に前の命令と同じクロックで実行され、次に実行される命令として現れない場合、その命令の出現回数は実際よりも少なくなることを意味します。また、呼び出しスタックが記録されるときに、「次」に実行する命令が複数存在する可能性もあります。

命令発行規則はプロセッサの種類ごとに異なり、キャッシュ行内の命令位置合わせに依存します。リンカーはキャッシュ行よりも高い精度による命令位置合わせを強制的に行うので、関連性がないと思える関数を変更すると、異なる命令の位置合わせが生じる可能性があります。位置合わせが異なると、パフォーマンスの向上や劣化が発生することがあります。

次の例では、同じ関数をわずかに異なる状況でコンパイルしてリンクしています。2つの出力例は、er_print ユーティリティーからの注釈付き逆アセンブリリストを示しています。2 つの例の命令は同じですが、位置合わせが異なっています。

この例の命令位置合わせでは、cmp と bl,a の 2 つの命令を別々のキャッシュ行にマップし、この 2 つの命令の実行待ちに多大な時間が消費されます。

```
   Excl.     Incl.
User CPU  User CPU
    sec.      sec.
                            1. static int
                            2. ifunc()
                            3. {
                            4.     int i;
                            5.
                            6.     for (i=0; i<10000; i++)
                               <function: ifunc>
   0.010     0.010              [ 6]    1066c:  clr         %o0
   0.        0.                 [ 6]    10670:  sethi       %hi(0x2400), %o5
   0.        0.                 [ 6]    10674:  inc         784, %o5
                            7.         i++;
   0.        0.                 [ 7]    10678:  inc         2, %o0
## 1.360     1.360              [ 7]    1067c:  cmp         %o0, %o5
## 1.510     1.510              [ 7]    10680:  bl,a        0x1067c
   0.        0.                 [ 7]    10684:  inc         2, %o0
   0.        0.                 [ 7]    10688:  retl
   0.        0.                 [ 7]    1068c:  nop
                            8.     return i;
                            9. }
```

この例の命令位置合わせでは、cmp と bl,a の 2 つの命令を 1 つのキャッシュ行にマップし、この 2 つの命令の内 1 つの命令のみの実行待ちに多大な時間が消費されます。

```
   Excl.     Incl.
User CPU  User CPU
    sec.      sec.
                            1. static int
                            2. ifunc()
                            3. {
                            4.     int i;
                            5.
                            6.     for (i=0; i<10000; i++)
                               <function: ifunc>
   0.        0.                 [ 6]    10684:  clr         %o0
   0.        0.                 [ 6]    10688:  sethi       %hi(0x2400), %o5
   0.        0.                 [ 6]    1068c:  inc         784, %o5
                            7.         i++;
```

```
   0.        0.                [ 7]    10690:  inc         2, %o0
## 1.440     1.440             [ 7]    10694:  cmp         %o0, %o5
   0.        0.                [ 7]    10698:  bl,a        0x10694
   0.        0.                [ 7]    1069c:  inc         2, %o0
   0.        0.                [ 7]    106a0:  retl
   0.        0.                [ 7]    106a4:  nop
                           8.      return i;
                           9. }
```

命令発行遅延

特定のリーフ PC の示す命令の発行前に遅延があると、そのリーフ PC の出現回数が多くなることがあります。これは、次のケースを初めとして、さまざまな多くの理由で発生する可能性があります。

- 命令がカーネルにトラップされたときのように、前の命令の実行に時間がかかり、割り込みが不可能な場合。
- 算術演算命令が必要とするレジスタの内容が前の命令によって設定されていて、その命令がまだ完了していない場合。このような遅延の例としては、たとえば、データキャッシュミスが発生したロード命令があります。
- 浮動小数点演算命令が、別の浮動小数点演算命令の終了待ちになっている場合。このような状況は、平方根や浮動小数点除算などのパイプライン化が不可能な命令で発生します。
- 命令を含むメモリーワードが命令キャッシュに含まれていない場合 (I キャッシュミス)。
- UltraSPARC® III プロセッサ上では、ロード命令でキャッシュミスが発生すると、ミスが解決されないかぎり、その後の命令は、読み込み中のデータを使用する命令であるかどうかに関係なく、すべてブロックされます。UltraSPARC II プロセッサの場合には、読み込み中のデータ項目を使用する命令だけがブロックされます。

ハードウェアカウンタオーバーフローの関連付け

UltraSPARC プラットフォームでの TLB ミスを除き、オーバーフローにより生成される割り込みの処理に要する時間などのいくつかの理由から、ハードウェアカウンタのオーバーフローイベントの呼び出しスタックは、オーバーフローが発生した時点ではなく命令シーケンスの後ろの方で記録されます。 サイクルおよび発行された命令などの一部のカウンタの場合、この遅延は問題になりません。しかし、キャッシュミスや浮動小数点演算をカウントするようなカウンタの場合は、そのオーバーフローの原因となっているものとは別の命令がメトリックの原因とされます。多くの場合、イベントを発生させた PC は、記録されている PC の数命令前のもので、逆アセンブリリストでその命令を正確に特定できます。ただし、この命令範囲内に分岐先がある場合、イベントを発生させた PC に対 応する命令を見分けることは難しいか、または不可能です。メモリーアクセスイベントをカウントするハードウェアカウンタについて、コレクタはカウンタ名の前に「+」が付いている場合、イベントを発生させた PC を検索します。この方法で記録されたデータ

は、データ空間プロファイリングをサポートします。詳細は、170 ページの「データ空間プロファイリング」および63 ページの「-h *counter_definition_1*...[, *counter_definition_n*]」を参照してください。

「ソース」タブ、「逆アセンブリ」タブ、「PC」タブの特別な行

アウトライン関数

フィードバック最適化コンパイルで、アウトライン関数が作成されることがあります。これらは、「ソース」タブと「逆アセンブリ」タブで、特別なインデックス行として表示されます。「ソース」タブでは、注釈は、アウトライン関数に変換されたコードブロックに表示されます。

```
                  Function binsearchmod inlined from source file ptralias2.c into the
0.          0 .      58.         if( binsearchmod( asize, &element ) ) {
0.240       0.240    59.             if( key != (element << 1) ) {
0.          0.       60.                 error |= BINSEARCHMODPOSTESTFAILED;
                        <Function: main -- outline code from line 60 [_$o1B60.main]>
0.040       0.040  [ 61]                 break;
0.          0.       62.             }
0.          0.       63.         }
```

「逆アセンブリ」タブでは、アウトライン関数は通常、ファイルの末尾に表示されます。

```
                        <Function: main -- outline code from line 85 [_$o1D85.main]>
0.          0.          [ 85] 100001034:  sethi       %hi(0x100000), %i5
0.          0.          [ 86] 100001038:  bset        4, %i3
0.          0.          [ 85] 10000103c:  or          %i5, 1, %l7
0.          0.          [ 85] 100001040:  sllx        %l7, 12, %l5
0.          0.          [ 85] 100001044:  call        printf ! 0x100101300
0.          0.          [ 85] 100001048:  add         %l5, 336, %o0
0.          0.          [ 90] 10000104c:  cmp         %i3, 0
0.          0.          [ 20] 100001050:  ba,a        0x1000010b4
                        <Function: main -- outline code from line 46 [_$o1A46.main]>
0.          0.          [ 46] 100001054:  mov         1, %i3
0.          0.          [ 47] 100001058:  ba          0x100001090
0.          0.          [ 56] 10000105c:  clr         [%i2]
                        <Function: main -- outline code from line 60 [_$o1B60.main]>
0.          0.          [ 60] 100001060:  bset        2, %i3
0.          0.          [ 61] 100001064:  ba          0x10000109c
0.          0.          [ 74] 100001068:  mov         1, %o3
```

アウトライン関数の名前は角括弧内に表示され、コードの取り出し元関数の名前や、ソースコード内のセクションの先頭の行番号を含む、アウトライン化したコードのセクションに関する情報をエンコードします。これらの符号化された名前は、リリースごとに異なります。アナライザは、読みやすい関数名を表示します。詳細は、190 ページの「アウトライン関数」を参照してください。

アプリケーションのパフォーマンスデータの収集中にアウトライン関数が呼び出されると、アナライザは注釈付き逆アセンブリに特別な行を表示して、その関数の包括的メトリックスを示します。詳細は、216 ページの「包括的メトリックス」を参照してください。

コンパイラ生成の本体関数

関数内のループ、または並列化指令のある領域を並列化する場合、コンパイラは、元のソースコードに含まれていない新しい本体関数を作成します。こうした関数については、178 ページの「OpenMP ソフトウェア実行の概要」で説明しています。

コンパイラは、並列構造の種類、構造の取り出し元関数の名前、元のソースにおける構造の先頭の行番号、並列構造のシーケンス番号などをエンコードする本体関数に、符号化名を割り当てます。これらの符号化された名前は、マイクロタスクライブラリのリリースごとに異なりますが、より分かりやすい名前に復号化されて表示されます。

次に、関数リストに表示されるような一般的なコンパイラ生成の本体関数を示します。

```
7.415      14.860      psec_ -- OMP sections from line 9 [_$s1A9.psec_]
3.873       3.903      craydo_ -- MP doall from line 10 [_$d1A10.craydo_]
```

この例で分かるように、構造が抽出された関数の名前が最初に示され、次に並列構造の種類、並列構造の行番号、コンパイラ生成の本体関数の符号化名が角括弧に表示されます。同様に、逆アセンブリコード内に特別なインデックス行が生成されます。

```
0.       0.           <Function: psec_ -- OMP sections from line 9 [_$s1A9.psec_]>
0.       7.445        [24]    1d8cc:  save        %sp, -168, %sp
0.       0.           [24]    1d8d0:  ld          [%i0], %g1
0.       0.           [24]    1d8d4:  tst         %i1

0.       0.           <Function: craydo_ -- MP doall from line 10 [_$d1A10.craydo_]>
0.       0.030        [ ?]    197e8:  save        %sp, -128, %sp
0.       0.           [ ?]    197ec:  ld          [%i0 + 20], %i5
0.       0.           [ ?]    197f0:  st          %i1, [%sp + 112]
0.       0.           [ ?]    197f4:  ld          [%i5], %i3
```

Cray の指令では、関数はソースコード行番号に関連付けされていない可能性があります。このような場合、行番号の代わりに [?] が表示されます。注釈付きソースコードにインデックス行が表示される場合は、次のようにインデックス行は行番号なしで命令を示します。

```
                  9. c$mic  doall shared(a,b,c,n) private(i,j,k)

                Loop below fused with loop on line 23
```

```
                  Loop below not parallelized because autoparallelization
                    is not enabled
                  Loop below autoparallelized
                  Loop below interchanged with loop on line 12
                  Loop below interchanged with loop on line 12
3.873     3.903         <Function: craydo_ -- MP doall from line 10 [_$d1A10.craydo_],
                      instructions without line numbers>
0.        3.903     10.            do i = 2, n-1
```

注 – インデックス行やコンパイラのコメント行は、実際の表示では折り返されません。

動的にコンパイルされる関数

動的にコンパイルされる関数は、プログラムの実行中にコンパイルされリンクされる関数です。コレクタ API 関数`collector_func_load()` を使用して必要な情報をユーザーが提供しないかぎり、コレクタは C や C++ で記述された動的にコンパイルされる関数に関する情報を持っていません。「関数」タブ、「ソース」タブ、「逆アセンブリ」タブに表示される情報は、次のように、`collector_func_load()` に渡される情報によって異なります。

- 情報が提供されていない場合、つまり `collector_func_load()` が呼び出されていない場合、動的にコンパイルされて読み込まれた関数は、関数リストに `<Unknown>` として表示されます。関数ソースも逆アセンブリコードも、アナライザには表示されません。
- ソースファイル名と行番号のテーブルが提供されていない場合に、関数の名前、サイズ、およびアドレスが指定されている場合は、動的にコンパイルされて読み込まれる関数の名前とそのメトリックスが関数リストに表示されます。注釈付きソースコードは利用可能で、逆アセンブリ命令を表示できます。ただし、行番号は不明であることを示すために `[?]` で示されます。
- ソースファイル名を指定し、行番号テーブルを提供しないと、ソースファイル名を指定しない場合と同様の情報がアナライザによって表示されます。ただし、注釈付きソースの先頭には、関数が行番号のない命令で構成されていることを示す特別なインデックス行が表示されます。次に例を示します。

  ```
  1.121     1.121          <Function func0, instructions without line numbers>
                        1. #include        <stdio.h>
  ```

- ソースファイル名と行番号テーブルが提供されている場合、関数とそのメトリックスは、従来の方法でコンパイルされた関数と同じように、「関数」タブ、「ソース」タブ、および「逆アセンブリ」タブに表示されます。

コレクタ API 関数の詳細については、52 ページの「動的な関数とモジュール」を参照してください。

Java プログラムでは、ほとんどのメソッドが JVM ソフトウェアによってインタプリタされます。別個のスレッドで動作する Java HotSpot 仮想マシンは、インタプリタの

実行中にパフォーマンスを監視します。監視プロセス中仮想マシンは、1 つ以上のインタプリタを行なっているメソッドを取り出し、それらのメソッド用のマシンコードを生成し、元のマシンコードをインタプリタするのではなくさらに効率の良いマシンコードバージョンを実行することを決定する場合があります。

Java プログラムでは、コレクタ API 関数を使用する必要はなく、次の例に示すように、アナライザがメソッドのインデックス行の下にある特別な行を使用して、注釈付き逆アセンブリリストに Java HotSpot でコンパイルされたコードが存在することを認識します。

```
                   11.    public int add_int () {
                   12.        int       x = 0;
2.832     2.832          <Function: Routine.add_int: HotSpot-compiled leaf instructions>
0.        0.             [ 12] 00000000: iconst_0
0.        0.             [ 12] 00000001: istore_1
```

逆アセンブリリストには、コンパイルされた命令ではなく、インタプリタされたバイトコードのみが示されます。デフォルトでは、コンパイルされたコードのメトリックスは、特別な行の隣りに表示されます。排他的および包括的 CPU 時間は、インタプリタされたバイトコードの各行に示されているすべての包括的および排他的 CPU 時間の合計とは異なります。通常は、何回かメソッドが呼び出されると、コンパイルされた命令の CPU 時間は、インタプリタされたバイトコードの CPU 時間の合計より多くなります。なぜなら、インタプリタされたコードは、メソッドが最初に呼び出されたときに一度だけ実行されるのに対し、コンパイルされたコードはその後も実行されるからです。

注釈付きソースには、Java HotSpot でコンパイルされた関数は表示されません。その代わりに、行番号なしで命令を示す特別なインデックス行が表示されます。たとえば、前述の逆アセンブリの抜粋に対応する注釈付きソースは、次のようになります。

```
                        11.    public int add_int () {
2.832     2.832           <Function: Routine.add_int(), instructions without line numbers>
0.        0.            12.       int       x = 0;
                          <Function: Routine.add_int()>
```

Java ネイティブ関数

ネイティブコードは、元は C、C++、または Fortran で記述されたコンパイル済みのコードで、Java コードから Java Native Interface (JNI) を介して呼び出されます。次の例は、デモプログラム `jsynprog` に関連付けられたファイル `jsynprog.java` の注釈付き逆アセンブリからの抜粋です。

```
                        5. class jsynprog
                           <Function: jsynprog.<init>()>
0.        5.504            jsynprog.JavaCC() <Java native method>
0.        1.431            jsynprog.JavaCJava(int) <Java native method>
```

```
0.       5.684          jsynprog.JavaJavaC(int) <Java native method>
0.       0.             [  5] 00000000: aload_0
0.       0.             [  5] 00000001: invokespecial <init>()
0.       0.             [  5] 00000004: return
```

ネイティブメソッドは Java ソースに含まれていないため、`jsynprog.java` の注釈付きソースの先頭には、行番号なしで命令を示す特別なインデックス行を使って各 Java ネイティブメソッドが表示されます。

```
0.       5.504          <Function: jsynprog.JavaCC(), instructions without line
                           numbers>
0.       1.431          <Function: jsynprog.JavaCJava(int), instructions without line
                           numbers>
0.       5.684          <Function: jsynprog.JavaJavaC(int), instructions without line
                           numbers>
```

注 – 実際の注釈付きソースの表示では、インデックス行は折り返されません。

クローン生成関数

コンパイラは、通常以上の最適化が可能な関数への呼び出しを見分けることができます。このような呼び出しの一例として、渡される引数の一部が定数である関数への呼び出しがあります。コンパイラは、最適化できる特定の呼び出しを見つけると、クローンと呼ばれるこの関数のコピーを作成して、最適化コードを生成します。

注釈付きソースでは、コンパイラのコメントは、クローン生成関数が作成されたかどうかを示します。

```
0.       0.       Function foo from source file clone.c cloned,
                   creating cloned function _$c1A.foo;
                   constant parameters propagated to clone
0.       0.570     27.    foo(100, 50, a, a+50, b);
```

注 – 実際の注釈付きソースの表示では、コンパイラのコメント行は折り返されません。

クローン関数名は、特定の呼び出しを識別する、符号化された名前です。前述の例では、コンパイラのコメントは、クローン生成関数の名前が `_$c1A.foo` であることを示しています。次に示すように、この関数は関数リストに表示されます。

```
0.350     0.550     foo
0.340     0.570     _$c1A.foo
```

逆アセンブリリストクローン生成関数はそれぞれ別の命令のセットを持っているため、注釈付きにはクローン生成関数が別々に表示されます。これらはソースファイ

ルに関連付けられていないため、命令はいずれのソース行番号とも関連付けられていません。次に、クローン生成関数の注釈付き逆アセンブリの最初の数行を示します。

```
0.       0.          <Function: _$c1A.foo>
0.       0.          [?]    10e98:  save        %sp, -120, %sp
0.       0.          [?]    10e9c:  sethi       %hi(0x10c00), %i4
0.       0.          [?]    10ea0:  mov         100, %i3
0.       0.          [?]    10ea4:  st          %i3, [%i0]
0.       0.          [?]    10ea8:  ldd         [%i4 + 640], %f8
```

静的関数

静的関数は、ライブラリ内でよく使用されます。これは、ライブラリ内部で使用される関数名が、ユーザーが使用する可能性のある関数名と衝突しないようにするためです。ライブラリをストリップすると、静的関数の名前はシンボルテーブルから削除されます。このような場合、アナライザは、ストリップ済み静的関数を含むライブラリ内のすべてのテキスト領域ごとに擬似的な名前を生成します。この名前は `<static>@0x12345` という形式で、`@` 記号に続く文字列は、ライブラリ内のテキスト領域のオフセット位置を表します。アナライザは、連続する複数のストリップ済み静的関数と単一のストリップ済み静的関数を区別できないため、複数のストリップ済み静的関数のメトリックスがまとめて表示されることがあります。静的関数の例は、次に示す `jsynprog` デモの関数リストで参照できます。

```
0.       0.       <static>@0x18780
0.       0.       <static>@0x20cc
0.       0.       <static>@0xc9f0
0.       0.       <static>@0xd1d8
0.       0.       <static>@0xe204
```

「PC」タブでは、前述の関数は、次のようにオフセットとともに示されます。

```
0.       0.       <static>@0x18780 + 0x00000818
0.       0.       <static>@0x20cc + 0x0000032C
0.       0.       <static>@0xc9f0 + 0x00000060
0.       0.       <static>@0xd1d8 + 0x00000040
0.       0.       <static>@0xe204 + 0x00000170
```

ストリップ済みライブラリ内で呼び出された関数は、「PC」タブで次のように表示される場合もあります。

```
<library.so> -- no functions found + 0x0000F870
```

包括的メトリックス

注釈付き逆アセンブリでは、アウトライン関数が要した時間にタグを付けるための特別な行が存在します。

次に、アウトライン関数が呼び出されたときに表示される注釈付き逆アセンブリの例を示します。

```
0.       0.        43.         else
0.       0.        44.         {
0.       0.        45.                 printf("else reached\n");
0.       2.522         <inclusive metrics for outlined functions>
```

分岐先

注釈付き逆アセンブリリストに表示される疑似行の <branch target> (分岐先) は、有効アドレスを検索するためのバックトラッキングにおいて、バックトラッキングのアルゴリズムが分岐先に対して実行されたために失敗した命令の PC に対応します。

ストア命令とロード命令の注釈

-xhwcprof オプションでコンパイルすると、コンパイラは、ストア (st) 命令とロード (ld) 命令についての追加情報を生成します。逆アセンブリリストに、注釈付きの st 命令と ld 命令が表示されます。

実験なしのソース/逆アセンブリの表示

実験を実行しなくても、er_src ユーティリティーを使用して、注釈付きソースコードや注釈付き逆アセンブリコードを表示できます。メトリックスが表示されないことを除けば、この表示は、アナライザで生成されるものと同じです。er_src コマンドの構文は次のとおりです。

```
er_src [ -func | -{source,src} item tag | -{disasm,dis} item tag |
-{cc,scc,dcc} com_spec | -outfile filename | -V ] object
```

object は、実行可能ファイル、共有オブジェクト、またはオブジェクトファイル (.o ファイル) の名前です。

item は、関数名、または実行可能オブジェクトや共有オブジェクトの構築に使用されたソースまたはオブジェクトファイルの名前です。*item* は、*function'file'* の形式でも指定できます。この場合、er_src は指定されたファイルのソースコンテキストに、指定された関数のソースまたは逆アセンブリを表示します。

tag は、同じ名前の関数が複数存在する場合に、参照する *item* を決定するために使用されるインデックスです。これは必須ですが、関数の解決に不要な場合は無視されます。

特別な item および tag の all -1 は、オブジェクトのすべての関数に対して、注釈付きソースまたは逆アセンブリを生成するように er_src に指示します。

注 – 実行可能ファイルや共有オブジェクトに `all -1` を使用した結果生成される出力は、非常に大きくなることもあります。

次に、`er_src` ユーティリティーに使用可能なオプションについて説明します。

`-func`

指定オブジェクトのすべての関数を一覧表示します。

`-{source,src}` *item tag*

リストされた item の注釈付きソースを示します。

`-{disasm,dis}` *item tag*

リストに逆アセンブリを含めます。デフォルトでは、リストに逆アセンブリは含まれません。ソースがない場合は、コンパイラのコメントなしで逆アセンブリのリストが生成されます。

`-{cc,scc,dcc}` *com-spec*

表示するコンパイラのコメントクラスを指定します。*com-spec* は、コロンで区切られたクラスのリストです。*com-spec* は、`-scc` オプションが指定されている場合はソースのコンパイラのコメントに、`-dcc` オプションが指定されている場合は逆アセンブリのコメントに、`-cc` が指定されている場合は両方のコメントに適用されます。これらのクラスについては、132 ページの「ソースリストと逆アセンブリリストを管理するコマンド」を参照してください。

コメントクラスは、デフォルト値ファイルで指定することができます。最初にシステム全体の `er.rc` デフォルト値ファイルが読み取られ、次にユーザーのホームディレクトリの `.er.rc` ファイルが存在する場合読み取られます。そして現在のディレクトリの `.er.rc` ファイルが読み取られます。ホームディレクトリの `.er.rc` ファイル内のデフォルト値はシステムのデフォルト値よりも優先され、現在のディレクトリの `.er.rc` ファイル内のデフォルト値は、ユーザーのホームおよびシステムのデフォルト値よりも優先されます。これらのファイルは、アナライザと `er_print` ユーティリティーによっても使用されますが、`er_src` ユーティリティーが使用するのは、ソースおよび逆アセンブリのコンパイラのコメントに関する設定の部分だけです。デフォルト値ファイルの詳細は、149 ページの「デフォルト値を設定するコマンド」を参照してください。`er_src` ユーティリティーは、デフォルト値ファイル内の `scc` および `dcc` 以外のコマンドを無視します。

-outfile*filename*

リストの出力用ファイル *filename* を開きます。デフォルトの場合、またはファイル名がダッシュ (-) の場合は、出力は `stdout` に書き込まれます。

-V

現在のリリースバージョンを表示します。

◆ ◆ ◆ 第 8 章

実験の操作

この章では、コレクタおよびパフォーマンスアナライザとともに利用できるユーティーリティーについて説明します。

この章では、次の内容について説明します。

実験の操作

実験は、コレクタによって作成されたディレクトリ内に格納されます。実験の操作に、cp、mv、rm など通常の UNIX コマンドを使用し、ディレクトリに適用することができます。Forte Developer 7 (Sun ONE Studio 7, Enterprise Edition for Solaris) より前のリリースの実験には当てはまりません。このため、これらの UNIX コマンドのような働きを持つ、実験のコピー、移動、および削除用の 3 つのユーティリティーが用意されています。次に、これらのユーティリティー er_cp(1)、er_mv(1)、er_rm(1) を説明します。

実験のデータには、プログラムによって使用された各ロードオブジェクトのアーカイブファイルが含まれます。これらのアーカイブファイルには、ロードオブジェクトの絶対パスとその最終修正日付が含まれています。実験を移動またはコピーしたときにこの情報が変更されることはありません。

er_cp ユーティリティーを使った実験のコピー

次の 2 つの形式の er_cp コマンドを使用できます。

```
er_cp [-V] experiment1 experiment2
er_cp [-V] experiment-list directory
```

最初の形式の er_cp コマンドは、*experiment1* を *experiment2* にコピーします。*experiment2* が存在する場合、er_cp はエラーメッセージを出力して終了します。2 つ目の形式は、リスト中の空白で区切られた一群の実験をディレクトリにコピーします。コピー先のディレクトリにコピー対象の実験と同じ名前の実験がすでに存在する場合、er_mv ユーティリティーはエラーメッセージを出力して終了します。-V オプションは、er_cp ユーティリティーのバージョンを表示します。このコマンドでは、Forte Developer 7 リリースより前のソフトウェアリリースで作成された実験はコピーされません。

er_mv ユーティリティーを使った実験の移動

次の 2 つの形式の er_mv コマンドを使用できます。

```
er_mv [-V] experiment1 experiment2
er_mv [-V] experiment-list directory
```

最初の形式の er_mv コマンドは、*experiment1* を *experiment2* に移動します。*experiment2* が存在する場合、er_mv ユーティリティーはエラーメッセージを出力して終了します。2 つ目の形式のコマンドは、リスト中の空白で区切られた一群の実験をディレクトリに移動します。移動先のディレクトリに移動対象の実験と同じ名前の実験がすでに存在する場合、er_mv ユーティリティーはエラーメッセージを出力して終了します。-V オプションは、er_mv ユーティリティーのバージョンを表示します。このコマンドでは、Forte Developer 7 リリースより前のソフトウェアリリースで作成された実験は移動されません。

er_rm ユーティリティーを使った実験の削除

実験または実験グループのリストを削除します。実験グループが削除されるときは、グループ内の実験が削除されてから、グループファイルが削除されます。

er_rm コマンドの構文は次のとおりです。

```
er_rm [-f] [-V] experiment-list
```

-f オプションを使用するとエラーメッセージが表示されず、実験が存在するかどうかにかかわらずコマンドは正常に終了します。-V オプションは、er_rm ユーティリティーのバージョンを表示します。このコマンドは、Forte Developer 7 リリースより前のソフトウェアリリースで作成された実験を削除します。

そのほかのユーティリティー

そのほかのユーティリティーとして、通常の状況で使用する必要のないものがいくつかあります。ここでは、完全な説明を行うためこれらのユーティリティーについて解説し、それらを使用することが必要となる可能性がある状況についても説明します。

er_archive ユーティリティー

er_archive コマンドの構文は次のとおりです。

```
er_archive [-nqAF] experiment
er_archive -V
```

er_archive ユーティリティーは、実験が正常に完了したとき、または実験でパフォーマンスアナライザか er_print ユーティリティーが開始されたとき、自動的に実行されます。er_archive ユーティリティーは、カーネルプロファイリングセッションが Ctrl-C または kill コマンドにより強制終了された場合も、er_kernel により自動的に実行されます。このユーティリティーは、実験で参照されている共有オブジェクトのリストを読み取り、それぞれについてアーカイブファイルを構築します。それぞれの出力ファイル名には .archive というサフィックスが付加され、共有オブジェクトについての関数およびモジュールのマッピングが含まれます。

対象のプログラムが異常終了すると、コレクタによって er_archive ユーティリティーが実行されない場合があります。実験が異常終了した場合、その実験が記録されたものとは別のマシンからその実行を検査するには、データが記録されているマシン上で、その実験に対して er_archive ユーティリティーを実行する必要があります。実験がコピーされる先のマシンのロードオブジェクトが利用できることを確認するには、-A オプションを使用します。

アーカイブファイルは、実験で参照されているすべての共有オブジェクトについて生成されます。これらのアーカイブには、すべてのオブジェクトファイルとそのロードオブジェクト内のすべての関数のアドレス、サイズ、名前、ロードオブジェクトの絶対パス、および最終変更日時を示すタイムスタンプが含まれます。

er_archive ユーティリティーの実行時に共有オブジェクトが見つからない場合、オブジェクトのタイムスタンプが実験に記録されているものと異なる場合、または実験が記録されたものとは異なるマシン上で er_archive ユーティリティーが実行された場合は、アーカイブファイルに警告が含まれます。er_archive ユーティリティが手動で実行された場合 (-q フラグを使用せずに実行された場合) は、stderr にも警告が出力されます。

次に、er_archive ユーティリティーに使用可能なオプションについて説明します。

-n

名前付きの実験のみをアーカイブし、子孫はアーカイブしません。

-q

stderr に警告を出力しません。警告はアーカイブファイルに組み入れられ、パフォーマンスアナライザ、または er_print ユーティリティーの出力に表示されます。

-A

すべてのロードオブジェクトを実験に書き込むことを要求します。この引数を使用して生成された実験は、その実験が記録されているものとは別のマシンに簡単にコピーできます。

-F

アーカイブファイルを強制的に書き込み、または再書き込みします。この引数を使用すると、er_archive を手動で実行し、警告を含むファイルを再書き込みできます。

-V

er_archive ユーティリティーのバージョン番号の情報を書き込み、終了します。

er_export ユーティリティー

er_export コマンドの構文は次のとおりです。

er_export [-V] *experiment*

er_export ユーティリティーは、実験に含まれている raw データを ASCII テキストに変換します。ファイルの書式と内容は変更される可能性があり、どのような用途でも特定の書式や内容に依存するべきではありません。このユーティリティーは、パフォーマンスアナライザが実験を読み取れない場合にのみ使用されることを意図したものです。この出力を使用して、ツール開発者は raw データを調べて問題を解析できます。-V オプションは、バージョン番号の情報を表示します。

◆ ◆ ◆ 第 9 章

9 カーネルプロファイリング

この章では、Solaris OS が負荷を実行中に、Oracle Solaris Studio のパフォーマンスツールを使用してカーネルのプロファイリングを行う方法について説明します。カーネルプロファイリングは、Oracle Solaris Studio ソフトウェアを Solaris 10 OS 上で実行している場合に使用できます。カーネルプロファイリングは Solaris 9 OS および Linux システムでは利用できません。

この章では、次の内容について説明します。

カーネル実験

カーネルプロファイルを er_kernelユーティリティーで記録できます。

er_kernel ユーティリティーは、Oracle Solaris 10 オペレーティングシステムに組み込まれている包括的な動的トレース機能である DTrace を使用します。

er_kernel ユーティリティーは、カーネルプロファイルデータを取り込み、そのデータをアナライザの実験としてユーザープロファイルと同じ形式で記録します。この実験は、er_print ユーティリティーまたはパフォーマンスアナライザによって処理できます。カーネル実験は、関数データ、呼び出し元と呼び出し先のデータ、命令レベルのデータ、およびタイムラインを示すことができますが、ほとんどの Solaris OS モジュールが行番号テーブルを保持していないため、ソース行データを示すことはできません。

カーネルプロファイリング用のシステムの設定

er_kernel ユーティリティーを使用してカーネルプロファイリングを行う前に、DTrace へのアクセスを設定しておく必要があります。

通常、DTrace は root ユーザーだけに制限されています。root 以外のユーザーとして er_kernelユーティリティーを 実行するには、特別な権限の割り当てを受けて、sys グループのメンバーになる必要があります。必要な権限を割り当てるには、次の行を /etc/user_attr ファイルに追加します。

```
username::::defaultpriv=basic,dtrace_kernel,dtrace_proc
```

自分自身を sys グループに追加するには、自分のユーザー名を /etc/group ファイル内の sys行に追加します。

er_kernel ユーティリティーの実行

er_kernel ユーティリティーを実行すると、カーネルのみ、またはカーネルと実行中の負荷の両方をプロファイリングできます。er_kernel コマンドの詳細な説明については er_kernel (1) のマニュアルページを参照してください。

▼ カーネルのプロファイリング

1. 次のように入力し、実験を収集します。

   ```
   % er_kernel -p on
   ```

2. 任意の負荷を別のシェルで実行します。

3. 負荷が完了したら、**Ctrl-C** キーを押して **er_kernel** ユーティリティーを終了します。

4. デフォルトでは **ktest.1.er** という名前の結果の実験を、パフォーマンスアナライザまたは **er_print** ユーティリティーに読み込みます。

   カーネルの時間プロファイルによって、「KCPU サイクル」というラベルの付いたパフォーマンスメトリックが 1 つ生成されます。パフォーマンスアナライザでは、「関数」タブのカーネル関数について示され、「呼び出し元 - 呼び出し先」タブでは呼び出し先と呼び出し元について示され、「逆アセンブリ」タブでは命令について示されます。「ソース」タブにはデータは表示されません。カーネルモジュールは、通常、出荷時点ではファイルおよび行シンボルテーブル情報 (スタブ) を含んでいないからです。

   er_kernel ユーティリティーへの -p on 引数を、高分解能プロファイルの場合は < -p high に、低分解能プロファイルの場合は -p low に置き換えることができます。 負荷

の実行に 2 ～ 20 分を要すると思われる場合は、デフォルトの時間プロファイルが適切です。実行に要する時間が 2 分未満と思われる場合は -p high を使用し、20 分を超えると思われる場合は -p low を使用します。

-t 所要時間引数を追加でき、これを追加すると er_kernel ユーティリティーは所要時間で指定された時間に従って自動的に終了します。

-t 所要時間は、m (分) または s (秒) のサフィックスを付けた単数で指定できます。これは、実験を強制終了するまでの時間 (分または秒) を示します。 デフォルトでは、所要時間は秒です。所要時間はハイフンで区切られた 2 つの数で指定することもできます。これは、1 つ目の時間が経過するまでデータ収集を停止し、そして、データ収集を始める時間を示しています。2 つ目の時間が経過すると、データ収集が終了されます。2 つ目の時間がゼロの場合、初めてプログラムが停止したあと、そのプログラムの実行の終わりまで、データの収集が実行されます。実験が終了しても、ターゲットプロセスは最後まで実行できます。

期間または間隔が指定されていない場合、er_kernel は停止するまで実行されます。停止させるには、Ctrl-C (SIGINT) を押します。あるいは、kill コマンドを使用し、SIGINT か SIGQUIT か SIGTERM を er_kernel プロセスに送信しても停止させることもできます。er_kernel プロセスは、これらの信号のいずれかを受け取ると、実験を終了し (-A off が指定されていない限り)、er_archive を実行します。er_archive ユーティリティーは、実験で参照されている共有オブジェクトのリストを読み取り、それぞれについてアーカイブファイルを構築します。

-v引数を追加すると、実行に関するより多くの情報を画面に出力できます。 -n 引数を使用すると、実際には何も記録せずに、記録される実験のプレビューを表示できます。

デフォルトでは、er_kernel ユーティリティーによって生成される実験の名前は ktest.1.er で、続けて実験が生成されると番号が順に増えていきます。

▼ 負荷の下でのプロファイリング

プログラムでもスクリプトでも、負荷として使用する単一のコマンドがある場合、次のようにします。

1 次のように入力し、実験を収集します。

```
% er_kernel -p on load
```

2 次のように入力し、実験を解析します。

```
% analyzer ktest.1.er
```

er_kernelユーティリティーは子プロセスをフォークし、休眠期間だけ一時停止したあと、子プロセスが指定された負荷を実行します。負荷が終了すると、er_kernel ユーティリティーは再び休眠期間だけ一時停止し、そのあと終了します。実験は、負荷の実行中、およびその前後の休眠期間での Solaris OS の動作を示します。休眠期間の長さは、er_kernel コマンドへの -q 引数によって秒単位で指定できます。

▼ カーネルと負荷の両方のプロファイリング

負荷として使用する単一のプログラムがあり、そのプロファイルをカーネルプロファイルと一緒に表示することに関心がある場合は、次のようにします。

1 **er_kernel** コマンドと **collect** コマンドの両方を次のように入力することにより、カーネルプロファイルとユーザープロファイルの両方を収集します。

```
% er_kernel collect load
```

2 2 つのプロファイルを一緒に解析するには、次のように入力します。

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

ktest.1.erアナライザによって表示されるデータは、test.1.er からのカーネルプロファイルと <literal>test.1.er</literal> からのユーザープロファイルの両方を示します。タイムラインを使用すると、2 つの実験間の相関関係がわかります。

注 – スクリプトを負荷として使用し、そのさまざまな部分のプロファイリングを行うには、スクリプト内の各種コマンドの前に collect コマンドと適切な引数を付加します。

特定のプロセスまたはカーネルスレッドのプロファイリング

er_kernel ユーティリティーを 1 つ以上の -T 引数を使用して起動すると、次のように特定のプロセスまたはスレッドのプロファイリングを指定できます。

- -T *pid*/ *tid* (特定のプロセスとカーネルスレッドの場合)
- -T 0/ *did* (特定の純カーネルスレッドの場合)

ターゲットのスレッドは、それらのスレッドについて er_kernel ユーティリティーを起動する前に作成されている必要があります。

1 つ以上の -T 引数を指定した場合は、Kthr 時間というラベルの付いた追加メトリックが生成されます。データは、CPU 上で実行されているかどうかに関わらず、プロファイリングされたすべてのスレッドについて取得されます。プロセスが中断されている (関数 <SLEEPING>) か CPU を待っている (関数 <STALLED>) かを示すために、特殊な単一フレームの呼び出しスタックが使用されます。

Kthr 時間 メトリックスが高く、KCPU サイクルメトリックスが低い関数は、プロファイリングされたスレッドが何か別のイベントを待って大量の時間を消費している関数です。

カーネルプロファイルの分析

カーネル実験内に記録されたフィールドのいくつかは、ユーザーモード実験での同じフィールドとは異なる意味を持っています。ユーザーモード実験には、単一のプロセス ID についてのデータのみが含まれています。カーネル実験には、多数の異なるプロセス ID に適用できるデータが含まれています。その情報を適切に示すために、アナライザのいくつかのフィールドラベルは、次の表に示すとおり、2 つのタイプの実験で異なる意味を持っています。

表 9–1 アナライザにおけるカーネル実験のフィールドラベルの意味

アナライザのラベル	ユーザーモード実験での意味	カーネル実験での意味
LWP	ユーザープロセス LWP ID	プロセスの PID。カーネルスレッドの場合は 0。
Thread	プロセス内のスレッド ID	カーネルスレッドのカーネル DID

たとえば、カーネル実験で少数のプロセス ID にのみフィルタを実行する場合は、「データをフィルタ」ダイアログボックスの「LWP」フィルタフィールドに、対象とする PID (単数または複数) を入力します。

索引

F

I

J

L

M

N

O

P

R

き

く

け

こ

さ

し

す

せ

と

に

ね

は