プログラムのパフォーマンス解析

Sun™ Studio 9

Sun Microsystems, Inc.
www.sun.com

Part No. 817-7883-10
2004 年 7 月, Revision A

原典： Performance Analyzer : Sun Studio 9
Part No: 817-6696-10
Revision A

目次

図目次

表目次

はじめに

このマニュアルでは、Sun™ Studio 9 で利用できるパフォーマンス解析ツールについて説明しています。

- 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、パフォーマンス解析を行います。広範囲の性能データの統計的プロファイリングと多数のシステムコールの監視を行い、そのデータを関数、ソース行、命令レベルでアプリケーションのプログラム構造に関連付けます。
- `prof` および `gprof` は、CPU の使用に関する統計的プロファイリングを行い、関数レベルの実行回数情報を提供するツールです。
- `tcov` は、関数およびソース行レベルの実行回数情報を提供するツールです。

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と、Solaris™ オペレーティングシステム、Linux または UNIX® オペレーティングシステムのコマンド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。

内容の紹介

第 1 章 では、パフォーマンス解析ツールの紹介をするとともに、それらツールの働きとどのようなときに使用すべきかを簡単に説明しています。

第 2 章 では、コレクタが収集したデータについての説明と、収集したデータのパフォーマンスメトリックへの変換処理とについて説明しています。

第 3 章 では、コレクタを使用し、アプリケーションからタイミングデータ、同期遅延データ、ハードウェアイベントデータを収集する方法を説明しています。

第 4 章 では、パフォーマンスアナライザの起動方法と、このツールを使用してコレクタが収集したパフォーマンスデータを解析する方法を説明しています。

第 5 章 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を説明しています。

第 6 章 では、コレクタが収集したデータのパフォーマンスメトリックへの変換処理と、アプリケーションのプログラム構造へのメトリックの対応付け方法を説明しています。

第 7 章 では、パフォーマンスアナライザのソースおよび逆アセンブリウィンドウの使用方法とそれらウィンドウに表示される情報の意味について説明しています。

第 8 章 では、実験ファイルを操作して変換したり、実験をせずに注釈付きソースや逆アセンブリコードを表示したりするユーティリティを紹介しています。

付録 A では、UNIX のプロファイリングツールである prof、gprof、tcov を取り上げています。これらのツールから、タイミングおよび実行回数統計情報を得ることができます。

書体と記号について

次の表と記述は、このマニュアルで使用している書体と記号について説明しています。

書体または記号	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、コーディング例。	.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。 machine_name% You have mail.
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表わします。	machine_name% **su** Password:
AaBbCc123 または**ゴシック**	コマンド行の可変部分。実際の名前または実際の値と置き換えてください。	rm *filename* と入力します。 rm **ファイル名** と入力します。
『 』	参照する書名を示します。	『SPARCstorage Array ユーザーマニュアル』
「 」	参照する章、節、または、強調する語を示します。	第 6 章「データの管理」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、バックスラッシュは、継続を示します。	machinename% grep '^#define \ XV_VERSION_STRING'
➤	階層メニューのサブメニューを選択することを示します。	作成: 「返信」➤「送信者へ」

コードの記号	意味	記法	コード例
[ ]	角括弧にはオプションの引数が含まれます。	O[*n*]	-O4, -O
{ }	中括弧には、必須オプションの選択肢が含まれます。	d{y|n}	-dy
|	「パイプ」または「バー」と呼ばれる記号は、その中から 1 つだけを選択可能な複数の引数を区切ります。	B{dynamic|static}	-Bstatic
:	コロンは、コンマ同様に複数の引数を区切るために使用されることがあります。	R*dir*[:*dir*]	-R/local/libs:/U/a
...	省略記号は、連続するものの一部が省略されていることを示します。	-xinline=*f1*[,...*fn*]	-xinline=alpha,dos

シェルプロンプトについて

シェル	プロンプト
UNIX の C シェル	machine_name%
UNIX の Bourne シェルと Korn シェル	machine_name$
スーパーユーザー (シェルの種類を問わない)	#

コンパイラとツールおよびマニュアルページへのアクセス

コンパイラとツールおよびマニュアルページは、/usr/bin/ と /usr/share/man のディレクトリにはインストールされません。 コンパイラとツールにアクセスするには、PATH 環境変数を正しく設定しておく必要があります (xix ページの「コンパイラとツールへのアクセス方法」を参照)。 また、マニュアルページにアクセスするには、MANPATH 環境変数を正しく設定しておく必要があります (xx ページの「マニュアルページへのアクセス方法」を参照)。

PATH 変数についての詳細は、csh(1)、sh(1)、および ksh(1) のマニュアルページを参照してください。 MANPATH 変数についての詳細は、man(1) のマニュアルページを参照してください。 このリリースにアクセスするために PATH および MANPATH 変数を設定する方法の詳細は、『インストールガイド』を参照するか、システム管理者にお問い合わせください。

注 – この節に記載されている情報は Sun Studio のコンパイラとツールが Solaris プラットフォームでは、/opt ディレクトリ、Linux プラットフォームでは、/opt/sun にインストールされていることを想定しています。 製品ソフトウェアが /opt 以外のディレクトリ (Solaris プラットフォーム)、およびデフォルトのディレクトリ以外にインストールされている場合は、システム管理者に実際のパスをお尋ねください。

コンパイラとツールへのアクセス方法

PATH 環境変数を変更して、コンパイラとツールにアクセスできるようにする必要があるかどうか判断するには以下を実行します。

▼ PATH 環境変数を設定する必要があるかどうか判断する

1. **次のように入力して、 PATH 変数の現在値を表示します。**

```
% echo $PATH
```

2. **Solaris プラットフォームでは、出力内容から `/opt/SUNWspro/bin` を含むパスの文字列を検索します。 Linux プラットフォームでは、出力内容から `/opt/sun/sunstudio9/bin` を含むパスの文字列を検索します。**

   パスがある場合は、PATH 変数はコンパイラとツールにアクセスできるように設定されています。 このパスがない場合は、次の手順に従って、 PATH 環境変数を設定してください。

▼ PATH 環境変数を設定してコンパイラとツールにアクセスする

1. **C シェルを使用している場合は、ホームの `.cshrc` ファイルを編集します。 Bourne シェルまたは Korn シェルを使用している場合は、ホームの `.profile` ファイルを編集します。**

2. **Solaris プラットフォームでは、次のパスを PATH 環境変数に追加します。 Sun ONE Studio ソフトウェアまたは Forte Developer ソフトウェアをインストールしている場合は、インストール先へのパスの前に、次のパスを追加します。**

   `/opt/SUNWspro/bin`

   Linux プラットフォームでは、次のパスを MANPATH 環境変数に追加します。

   `/opt/SUNWspro/bin`

マニュアルページへのアクセス方法

マニュアルページにアクセスするために MANPATH 変数を変更する必要があるかどうかを判断するには以下を実行します。

▼ MANPATH 環境変数を設定する必要があるかどうか判断する

1. **次のように入力して、`collect` のマニュアルページを表示します。**

```
% man collect
```

2. **出力された場合、内容を確認します。**

   `collect`(1) のマニュアルページが見つからないか、表示されたマニュアルページがインストールされたソフトウェアの現バージョンのものと異なる場合は、この節の指示に従って、MANPATH 環境変数を設定してください。

▼ MANPATH 環境変数を設定してマニュアルページにアクセスする

1. C シェルを使用している場合は、ホームの `.cshrc` ファイルを編集します。 Bourne シェルまたは Korn シェルを使用している場合は、ホームの `.profile` ファイルを編集します。
2. Solaris プラットフォームでは、次のパスを `MANPATH` 環境変数に追加します。

   ```
   /opt/SUNWspro/man
   ```

   Linux プラットフォームでは、次のパスを `MANPATH` 環境変数に追加します。

   ```
   /opt/sun/sunstudio9/man
   ```

統合開発環境へのアクセス方法

Sun Studio 統合開発環境 (IDE) には、C や C++、Fortran アプリケーションを作成、編集、構築、デバッグ、パフォーマンス解析するためのモージュールが用意されています。

IDE を起動するコマンドは、`sunstudio` です。 このコマンドの詳細は、`sunstudio`(1) のマニュアルページを参照してください。

IDE が正しく動作するかどうかは、IDE がコアプラットフォームを検出できるかどうかに依存します。 このため、`sunstudio` コマンドは、次の 2 つの場所でコアプラットフォームを探します。

- コマンドは、最初にデフォルトのインストールディレクトリを調べます。Solaris プラットフォームでは、`/opt/netbeans/3.5M`、Linux プラットフォームでは、`/opt/sun/netbeans/3.5M` です。
- このデフォルトのディレクトリでコアプラットフォームが見つからなかった場合は、IDE が含まれているディレクトリとコアプラットフォームが含まれているディレクトリが同じであるか、同じ場所にマウントされているとみなします。 たとえば Solaris プラットフォームでは、IDE が含まれているディレクトリへのパスが `/foo/SUNWspro` の場合は、`/foo/netbeans/3.5M` ディレクトリにコアプラットフォームがないか調べます。 Linux プラットフォームでは、IDE が含まれているディレクトリへのパスが `/foo/sunstudio9` の場合は、`/foo/netbeans/3.5M` ディレクトリにコアプラットフォームがないか調べます。

`sunstudio` が探す場所のどちらにもコアプラットフォームをインストールしていないか、マウントしていない場合、クライアントシステムの各ユーザーは、コアプラットフォームがインストールされているか、マウントされている場所 (*/installation_directory*`/netbeans/3.5M`) を、`SPRO_NETBEANS_HOME` 環境変数に設定する必要があります。

Solaris プラットフォームでは、Forte Developer ソフトウェアまたは Sun ONE Studio ソフトウェアがインストールされている場合、IDE の各ユーザーはまた、`$PATH` のそのパスの前に、*/installation_directory*`/SUNWspro/bin` を追加する必要があります。

Linux プラットフォームでは、IDE の各ユーザーはまた、`$PATH` のそのパスの前に、*/installation_directory*`/sunstudio9/bin` を追加する必要があります。

`$PATH` には、*/installation_directory*`/netbeans/3.5M/bin` のパスは追加しないでください。

コンパイラとツールのマニュアルへのアクセス

マニュアルには、以下からアクセスできます。

- 製品マニュアルは、ご使用のローカルシステムまたはネットワークの製品にインストールされているマニュアルの索引から入手できます。Solaris プラットフォーム：`file:/opt/SUNWspro/docs/ja/index.html`
  Linux プラットフォーム：`file:/opt/sun/sunstudio9/docs/index.html`

  製品ソフトウェアが `/opt` 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。
- マニュアルは、`docs.sun.com` の Web サイトで入手できます。 以下に示すマニュアルは、インストールされている製品のマニュアルの索引から入手できます (`docs.sun.com` Web サイトでは入手できません)。。
  - 『Standard C++ Library Class Reference』
  - 『標準 C++ ライブラリ・ユーザーズガイド』
  - 『Tools.h++ クラスライブラリ・リファレンスマニュアル』
  - 『Tools.h++ ユーザーズガイド』
- リリースノートは、`docs.sun.com` で入手できます。
- IDE の全コンポーネント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、多くのウィンドウおよびダイアログにある「ヘルプ」ボタンを使ってアクセスできます。

インターネットの Web サイト (`http://docs.sun.com`) から、サンのマニュアルを参照したり、印刷したり、購入することができます。 マニュアルが見つからない場合はローカルシステムまたはネットワークの製品とともにインストールされているマニュアルの索引を参照してください。

注 – Sun では、本マニュアルに掲載した第三者の Web サイトのご利用に関しましては責任はなく、保証するものでもありません。 また、これらのサイトあるいはリソースに関する、あるいはこれらのサイト、リソースから利用可能であるコンテンツ、広告、製品、あるいは資料に関して一切の責任を負いません。 Sun は、これらのサイトあるいはリソースに関する、あるいはこれらのサイトから利用可能であるコンテンツ、製品、サービスのご利用あるいは信頼によって、あるいはそれに関連して発生するいかなる損害、損失、申し立てに対する一切の責任を負いません。

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以下の表に示す場所から参照することができます。 製品ソフトウェアが `/opt` 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。

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 (サードパーティ製マニュアルは除く)	形式 : HTML 場所 : `http://docs.sun.com`
サードパーティ製マニュアル • 『Standard C++ Library Class Reference』 • 『標準 C++ ライブラリ・ユーザーズガイド』 • 『Tools.h++ クラスライブラリ・リファレンスマニュアル』 • 『Tools.h++ ユーザーズガイド』	形式 : HTML 場所 : `file:/opt/SUNWspro/docs/index.html` のマニュアル索引 (Solaris プラットフォーム)
Readme およびマニュアルページ	形式 : HTML 場所 : `file:/opt/SUNWspro/docs/index.html` のマニュアル索引 (Solaris プラットフォーム) 形式 : HTML 場所 : `file:/opt/sun/sunstudio9/docs/index.html` のマニュアル索引 (Linux プラットフォーム)
オンラインヘルプ	形式 : HTML場所 : IDE または、analyzer 内の「ヘルプ」メニュー
リリースノート	形式 : HTML 場所 : `http://docs.sun.com`

コンパイラとツールに関する関連マニュアル

以下の表は、Solaris プラットフォームで file:/opt/SUNWspro/docs/ja/index.html および http://docs.sun.com から参照できるマニュアルの一覧です。 製品ソフトウェアが /opt 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。

マニュアルタイトル	内容の説明
C ユーザーズガイド	Sun Studio 9 C プログラミング言語コンパイラについて説明しています。また、ANSI C コンパイラの詳細情報も記載されています。
C++ ユーザーズガイド	Sun Studio 9 C++ コンパイラの使用方法を説明しています。また、コマンド行コンパイラオプションの詳細情報も記載されています。
Fortran ユーザーズガイド	Sun Studio 9 Fortran コンパイラのコンパイル時環境およびコマンド行オプションについて説明しています。
OpenMP API ユーザーズガイド	OpenMP 多重処理 API の概要とその Forte Developer 実装の詳細について説明しています。
Fortran プログラミングガイド	並列処理、最適化、共有ライブラリの作成などを含むプログラミング技法について説明しています。
dbx コマンドによるデバッグ	デバッガの使用法についてのリファレンスマニュアルです。 Solaris™ プロセスへの接続と切り離し、および制御下の環境でのプログラムの実行について説明しています。
パフォーマンス解析 Readme	パフォーマンスアナライザの新機能、既知の問題点、制限事項および互換性の問題について説明しています。
analyzer(1), collect(1), collector(1), er_print(1), er_src(1), および libcollector(3) のマニュアルページ	パフォーマンスアナライザのコマンド行ユーティリティについて説明しています。

以下の表は、Linux プラットフォームで file:/opt/sun/sunstudio9/docs/index.html および http://docs.sun.com から参照できるマニュアルの一覧です。 製品ソフトウェアが/opt/sun 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。

マニュアルタイトル	説明
パフォーマンス解析 Readme	パフォーマンスアナライザの新機能、既知の問題点、制限事項および互換性の問題について説明しています。
`analyzer`(1), `collect`(1), `collector`(1), `er_print`(1), `er_src`(1), および `libcollector`(3) のマニュアルページ	パフォーマンスアナライザのコマンド行ユーティリティについて説明しています。
`dbx` コマンドによるデバッグ	デバッガの使用法についてのリファレンスマニュアルです。 Solaris™ プロセスへの接続と切り離し、および制御下の環境でのプログラムの実行について説明しています。

関連する Solaris マニュアル

次の表では、`docs.sun.com` の Web サイトで参照できる関連マニュアルについて説明します。

マニュアルコレクション	マニュアルタイトル	説明
Solaris™ Reference Manual Collection	マニュアルページのセクションのタイトルを参照	Solaris のオペレーティング環境に関する情報を提供しています。
Solaris™ Software Developer Collection	リンカーとライブラリ	Solaris のリンクエディタと実行時リンカーの操作について説明しています。
Solaris™ Software Developer Collection	マルチスレッドのプログラミング	POSIX と Solaris スレッド API、同期オブジェクトのプログラミング、マルチスレッド化したプログラムのコンパイル、およびマルチスレッド化したプログラムのツール検索について説明します。
Solaris™ Software Developer Collection	SPARC Assembly Language Reference Manual	SPARC® プロセッサでのアセンブリ言語について説明します。
Solaris™ 9 Update Collection	Solaris Tunable Parameters Reference Manual	Solaris の調整可能なパラメータに関する参照情報を提供しています。

開発者向けのリソース

http://developers.sun.com/prodtech/cc にアクセスし、Compiler Collection というリンクをクリックして、以下のようなリソースを利用できます。リソースは頻繁に更新されます。

- プログラミング技術と最適な演習に関する技術文書
- プログラミングに関する簡単なヒントを集めた知識ベース
- コンパイラとツールのコンポーネントのマニュアル、ソフトウェアと共にインストールされるマニュアルの訂正
- サポートレベルに関する情報
- ユーザーフォーラム
- ダウンロード可能なサンプルコード
- 新しい技術の紹介

http://developers.sun.com でも開発者向けのリソースが提供されています。

技術サポートへの問い合わせ

製品についての技術的なご質問がございましたら、以下のサイトからお問い合わせください (このマニュアルで回答されていないものに限ります)。

http://sun.co.jp/service/contacting

第1章

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、最適化されたルーチンのライブラリ、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発見、および問題が発生するコード部分の発見に役立つツールについて説明します。

統合開発環境からのパフォーマンスアナライザの起動

統合開発環境 (IDE) からのパフォーマンスアナライザの起動については、デフォルトのインストールディレクトリにある次の文書索引 (documentation index) からアクセス可能なパフォーマンスアナライザの Readme をご覧ください。

`file:/opt/SUNWspro/docs/index.html`

Sun Studio 9 のコンパイラとツールが `/opt` ディレクトリにインストールされていない場合は、システム管理者に実際のパスをお尋ねください。

パフォーマンス解析ツール

その中でも、このマニュアルでは特に、コレクタとパフォーマンスアナライザを取り上げます。これらは、使用しているアプリケーションに関するパフォーマンスデータを収集、解析するために使用する 1 組のツールです。いずれのツールも、コマンド行とグラフィカルユーザーインタフェースのどちらからでも使用できます。

パフォーマンスの調整はソフトウェア開発者にとって主要な仕事ではないかもしれませんが、コレクタとパフォーマンスアナライザは開発者向けの設計になっています。これらのツールは、一般に使わ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。`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全体のうちのどのぐらいがアプリケーション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アプリケーション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

コレクタは、プロファイリングと呼ばれる統計方法を使用し、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、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です。コレクタは C、C++、および Fortran のプログラムに関する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、コレクタの詳細については第 3 章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、IDE、`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を表示します。これらのメトリックは、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
- ハードウェアカウンタメトリック
- 同期遅延メトリック
- メモリー割り当てメトリック
- MPI トレースメトリック

パフォーマンスアナライザは、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。また、プログラムのアドレス空間における関数の読み込み順序を変更するために「マップファイル」を作成することで、パフォーマンスを改善することもできます。

パフォーマンスアナライザの詳細は、第 4 章および IDE またはパフォーマンスアナライザ GUI のオンラインヘルプ、またコマンド行解析ツールの `er_print` については第 5 章を参照してください。

第 6 章では、パフォーマンスアナライザとそのデータ、たとえば、データ収集の機能、パフォーマンスメトリック、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理解に関する内容について説明しています。パフォーマンスデータではなくコンパイラのコメントが含まれている注釈付きのソースコードリストと逆アセンブリコードリストは、`er_src` ユーティリティによって表示できます (詳細は第 8 章を参照)。

第 7 章では、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 章では、実験をコピー、移動、削除、アーカイブ、およびエクスポートする方法を説明します。

`er_print` コマンド

`er_print` コマンドは、パフォーマンスアナライザによって提供されるすべての表示 (「タイムライン」表示を除く) をプレーンテキストで提示します。

`prof`、`gprof`、および `tcov` ツール

このマニュアルでは、次のパフォーマンスツールについても説明しています。

- `prof` および `gprof`

  `prof` と `gprof` はプロファイルデータを生成する UNIX® ツールであり、Solaris™ 7、8、および 9 のオペレーティングシステム (SPARC® プラットフォーム版) に組み込まれています。いずれのツールも、x86 プラットフォームでも提供されサポートされています。

- `tcov tcov`

  `tcov` は、各関数の呼び出し回数と各ソース行の実行回数を報告するコードカバレッジツールです。

`prof`、`gprof`、`tcov` についての詳細は、付録 A 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

注 – 以下は、「パフォーマンスアナライザ」ウィンドウの簡単な概要です。取り上げたタブの機能および特徴の詳細は、第 4 章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と包括的メトリックをまとめた、アプリケーションの関数の一覧が表示されます。この一覧の内容は、ロードオブジェクト、スレッド、軽量プロセス (LWP)、CPU、タイムスライスに基づいて表示することができます。

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い部分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付きのソース行と、コンパイラのコメントが表示され、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表示されます。

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。

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、プロセスの統計を表示するタブもあります。

パフォーマンスアナライザは、キーボードばかりでなく、マウスを使って操作することもできます。

第2章

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。

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ます。これらのどのツールも、「コレクタ」という用語で呼ばれます。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ます。これらのどのツールも、「解析ツール」という用語で呼ばれます。

この章では、以下について説明します。

- コレクタが収集するデータの内容
- プログラム構造へのメトリックの対応付け

パフォーマンスデータの収集と格納については、第 3 章を参照してください。

パフォーマンスアナライザによるパフォーマンスデータの解析については、第 4 章を参照してください。

er_print によるパフォーマンスデータの解析については、第 5 章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、大域データという 3 種類のデータを収集します。

- プロファイルデータは、一定の間隔でプロファイルイベントを記録することで収集されます。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は特定のイベントに関する情報であり、いずれのデータもパフォーマンスメトリックに変換されます。大域データはメトリック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のマーカを提供します。大域データは、特定の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 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が記録されます。

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、各イベント固有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 6 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 (Solaris™ のみ)
- 同期待ちトレースデータ (Solaris™ のみ)
- 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 (Solaris™ のみ)

この 5 種類のデータ、これらのデータから求めるメトリック、およびメトリックの使用方法について、以降の項で説明します。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、これはメトリックに変換できません。

時間データ

時間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によって異なります。

Solaris 時間プロファイル

Solaris™ のもと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記録されます。この間隔をプロファイル間隔といいます。この情報は整数型の配列に格納され、カーネルの管理する 10 個のマイクロアカウンティング状態のそれぞれに、1 つの配列要素が使用されます。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` 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。

時間ベースのデータをもとに計算されるメトリックの定義を下表に記載します。

表 2-1 Solaris タイミングメトリック

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 で費やした LWP 時間。一般的な「時計時間」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。
他の待ち時間	カーネルページ待機中に使用される LWP 時間。あるい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、全 LWP にまたがって時計時間以外の時間が集計されます。上記定義の時計時間は、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役立てることができます。
- 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消費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、他のプロセスが 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場合、要求対象のロックをスレッドが取得できないでいることを意味します。
- テキストページフォルト時間が大きいということは、リンカーによって生成されたコードが、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。この種の問題は、マップファイルを作成、利用することによって解決できます (パフォーマンスアナライザのオンラインヘルプの「マップファイルの作成と利用」を参照)。
- 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を意味します。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 時間プロファイル

Linux 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™ オペレーティングシステムの場合ほど正確には、実際のシステム CPU 時間である時間の割合を判断できない場合があります。アナライザは軽量プロセス (LWP) のデータであるかのように情報を表示しますが、現実には Linux オペレーティングシステム上に LWP はなく、表示される LWP ID は実際にはスレッド ID です。

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

一般に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一般に使用されます。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。現在、ハードウェアカウンタのオーバーフロープロファイルは、Solaris™ システムにのみ提供されています。

UltraSPARC® III プロセッサファミリーと IA プロセッサファミリーには、イベントのカウントに利用可能なレジスタが 2 つあります。コレクタは、このいずれかまたは両方のレジスタからデータを収集できます。レジスタごとに、オーバーフローをトレースするカウンタの種類を選択し、オーバ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、どちらのレジスタも利用できるものもあれば、一方のレジスタしか利用できないものもあります。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データをカウントメトリックに変換します。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るとともに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はここに記載してありません。次項では、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。

一部のハードウェアカウンタは、同じ情報もしくは関連性のある情報を示します。たとえば、分岐予測ミスが発生するとまちがった命令が命令キャッシュに読み込まれることになり、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と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、命令変換ルックアサイドバッファ (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オーバーフローは、イベントと対応するイベントカウンタにオーバーフローを発生させた命令の後によく送られる命令です。これは「スキッド」と呼ばれ、カウンタオーバーフロープロファイルの解釈を困難にします。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)。メモリー参照イベントに関するこのような追加情報があると、各種のデータ指向解析が容易になります。

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に対する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。現在、ハードウェアカウンタのプロファイルは、Solaris™ システムでのみ利用できます。

以下は、別名があるカウンタのカウンタリストに含まれるエントリの表示例です。この例の出力行はすべて、印刷用に書式が整えられています。実際の出力では、行は区切られません。

```
CPU サイクル (cycles = Cycle_cnt/*) 9999991 hi=1000003, lo=100000007
(CPU-cycles)

命令の実行 (insts = Instr_cnt/*) 9999991 hi=1000003, lo=100000007
(Events)

D$ 読み込み失敗 (dcrm = DC_rd_miss/1) 100003 hi=10007, lo=1000003 ロー
ド (Events)
```

最初の行の最初のフィールド「`CPU Cycles`」はメトリック名です。第 2 のフィールド「`cycles`」には、`collect` の `-h` counter... 引数で使用できる別名が示されます。これはまた、`er_print` で使用される名前です。第 3 のフィールド「`Cycle_cnt/*`」には、`cputrack(1)` で使用される内部名とそのカウンタを使用できるレジスタ番号が示されます。レジスタ番号は、0 か 1、または * です。この例の * は、カウンタがどちらのレジスタでも使用できることを示します。次のフィールドはオーバーフロー間隔であり、その次のフィールドは高分解能オーバーフロー間隔であり、最後のフィールドは低分解能オーバーフロー間隔です。「`(CPU-cycles)`」は、カウンタが CPU サイクル単位でカウントすることを示し、時間に変換することができます。

第 2 の行は「`(Events)`」で終了しており、これは、イベントをカウントするものの、時間に変換できないことを示します。

上記の例の 3 行目に示すように、行には低分解能値とカウンタ単位の間に追加フィールドがある場合があり、これはカウンタを「ロード」、「ストア」のいずれで起動できるかを示したり、カウンタがプログラム関連でないかどうかを示します。

表 2-2 に、UltraSPARC® および IA ハードウェアの両方で使用可能な、カウンタの別名をまとめています。UltraSPARC® ハードウェアで利用できる別名は、ほかにもあります。

表 2-2 SPARC および IA ハードウェアで使用可能なハードウェアカウンタの別名

カウンタの別名	メトリック名	内容の説明
cycles	CPU サイクル	いずれかのレジスタでカウントされる CPU サイクル
insts	実行された命令	いずれかのレジスタでカウントされる、実行された命令

この例の出力行はすべて、印刷用に書式が整えられています。実際の出力では、行は区切られません。別名の付かないカウンタの出力行は次のとおりです。

```
Cycle_cnt Events (reg. 0) 1000003 hi=100003, lo=9999991
(CPU-cycles)

Instr_cnt Events (reg. 0) 1000003 hi=100003, lo=9999991 (Events)

DC_rd Events (reg. 0) 1000003 hi=100003, lo=9999991 ロード (Events)
```

この行で、第 1 のフィールド「Cycle_cnt」には、cputrack(1) で使用される内部名とそのカウンタを使用できるレジスタ番号が示されます。文字列「Cycle_cnt Events」は、このカウンタのメトリック名です。行の残りの部分の形式は、別名付きカウンタの場合と同じです。

行端の「(CPU-cycles)」で示されるサイクル単位でカウントする別名付きカウンタと別名なしカウンタのいずれの場合も、報告されたメトリックはデフォルトで包括的時間と排他的時間に変換されますが、オプションとしてイベントカウントとして表示することができます。行端の「(Events)」で示されるイベント単位でカウントするカウンタの場合、報告されたメトリックは包括的および排他的イベントカウントです。

カウンタ名の後の「ロード」、「ストア」、または「ロード - ストア」で示されるように、メモリー演算に関連するハードウェアカウンタの場合、オーバーフローしたカウンタ上のイベントを発生させた正確な命令と有効アドレスを検索することを要求するために、カウンタの名前の前に「+」記号が付くことがあります。これを行うとデータ空間プロファイリングが可能になります。詳細は 72 ページの「「データオブジェクト」タブ」と 73 ページの「「データレイアウト」タブ」を参照してください。

カウンタがプログラム関連でない場合、プロファイリングにカウンタを使用すると警告が出され、プロファイリングでは呼び出しスタックが記録されませんが、疑似関数「collector_not_program_related」に費やされる時間が表示されます。スレッドと LWP ID は記録されますが、意味がありません。プログラミングに無関係な

イベントをカウントするカウンタの名前の後に、文字列「not-program-related」が表示されます。そのようなカウンタを使用すると、関数「collector_not_program_related」内のメトリックが報告され、収集前に警告が出されます。

カウンタリストでは、別名のあるカウンタが最初に置かれ、その後にレジスタ 0 で使用可能なカウンタ、レジスタ 1 で使用可能なカウンタが続きます。別名のあるカウンタは、別名が付いた状態と別名がない状態の計 2 回現れます。別名がないものには、カウンタに異なるオーバーフロー値を割り当てることができます。別名のあるサイクルカウンタのデフォルトオーバーフロー値は、時間データとほぼ同じデータ収集速度をもたらすように選択されています。他のカウンタのほうがアプリケーションの実際の動作に対してはるかに敏感です。

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 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のトレースといいます。また、ロック待ちに費やされる時間を待ち時間といいます。現在、同期待ちトレースは、Solaris™ システムでのみ利用できます。

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。

Java™ プログラムの同期トレースは、スレッドが Java™ モニタを取得しようとしたときに生成されるイベントに基づいています。これらのイベントに関してはマシンと Java™ の呼び出しスタックがともに収集されますが、JVM™ 内で使用される内部ロックに関しては同期トレースデータが収集されません。マシン表現では、スレッド同期が _lwp_mutex_lock への呼び出しに委譲され、これらの呼び出しはトレースされないので同期データは表示されません。

同期待ちトレースデータは、次のメトリックに変換されます。

表 2-3　同期待ちトレースメトリック

メトリック	定義
同期待ちカウント	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劣化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` 上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`、`deallocate` は C 標準ライブラリ関数を呼び出すので、これらのルーチンも間接的にトレースされます。

Java™ プログラムの場合、ヒープトレースデータは、すべてのオブジェクト割り当てイベント (ユーザーコードで生成される) とオブジェクト割り当て解除イベント (ガーベッジコレクションで生成される) を記録します。また、`malloc`、`free` 等の C/C++ メモリー管理関数を使用すると、記録されるイベントも生成されます。これらのイベントは、ネイティブコードから発生するか、JVM 自体から発生します。

JVM によって、非常に大きな分割単位でメモリーの割り当てと割り当て解除が行われます。Java™ コードからのメモリー割り当てはすべて、JVM とそのガベージコレクタで処理されます。ガベージコレクタでは C/C++ メモリーマップ関数 `mmap` が使われています。

ヒープトレースデータは、次のメトリックに変換されます。

表 2-4 メモリー割り当て (ヒープトレース) メトリック

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各呼び出しで割り当てられたバイト数の合計
リーク	対応する割り当て解除関数の呼び出しを持た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を収集すれば、プログラム内のメモリーリークを見つけたり、十分なメモリーが割り当てられていない場所を確認したりできます。

メモリーリークには、デバッグツール dbx などで使用される、もう 1 つの定義があります。その定義は、「プログラムのデータ空間のどこにもポインタを持たない動的に割り当てられたメモリーブロック」です。ここで使用されるリークの定義にはこの代替定義を含みますが、ポインタが存在するメモリーも含みます。

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に関するデータを収集できます。現在、MPI トレースは、Solaris™ システムでのみ利用できます。データ収集の対象となる関数を以下に示します。

`MPI_Allgather`	`MPI_Allgatherv`	`MPI_Allreduce`
`MPI_Alltoall`	`MPI_Alltoallv`	`MPI_Barrier`
`MPI_Bcast`	`MPI_Bsend`	`MPI_Gather`
`MPI_Gatherv`	`MPI_Irecv`	`MPI_Isend`
`MPI_Recv`	`MPI_Reduce`	`MPI_Reduce_scatter`
`MPI_Rsend`	`MPI_Scan`	`MPI_Scatter`
`MPI_Scatterv`	`MPI_Send`	`MPI_Sendrecv`
`MPI_Sendrecv_replace`	`MPI_Ssend`	`MPI_Wait`
`MPI_Waitall`	`MPI_Waitany`	`MPI_Waitsome`
`MPI_Win_fence`	`MPI_Win_lock`	

MPI トレースデータは、次のメトリックに変換されます。

表 2-5 MPI トレースメトリック

メトリック	定義
MPI 受信	データを受信する MPI 関数の受信操作回数
MPI 受信バイト数	MPI 関数で受信したバイト数
MPI 送信	データを送信する MPI 関数の送信操作回数
MPI 送信バイト数	MPI 関数で送信したバイト数
MPI 時間	MPI 関数のすべての呼び出しに使用した時間
他の MPI 呼び出し	その他の MPI 関数の呼び出し数

受信または送信したバイト数は、呼び出しにおいて与えられるバッファサイズです。この値は、実際に受信または送信したバイト数よりも大きいことがあります。大域通信関数と集合通信関数においては、直接的なプロセッサ間通信が行われるとともにデータ再送やデータ転送の最適化が行われないという前提に基づき、送信または受信されるバイト数が最大値となります。

トレースされる MPI ライブラリ関数を、MPI 送信関数、MPI 受信関数、MPI 送受信関数、その他の関数に分類して表 2-6 にまとめます。

表 2-6 送信、受信、送受信、その他への MPI 関数の分類

カテゴリ	関数
MPI 送信関数	`MPI_Bsend`、`MPI_Isend`、`MPI_Rsend`、`MPI_Send`、`MPI_Ssend`
MPI 受信関数	`MPI_Irecv`、`MPI_Recv`
MPI 送受信関数	`MPI_Allgather`、`MPI_Allgatherv`、`MPI_Allreduce`、`MPI_Alltoall`、`MPI_Alltoallv`、`MPI_Bcast`、`MPI_Gather`、`MPI_Gatherv`、`MPI_Reduce`、`MPI_Reduce_scatter`、`MPI_Scan`、`MPI_Scatter`、`MPI_Scatterv`、`MPI_Sendrecv`、`MPI_Sendrecv_replace`
その他の MPI 関数	`MPI_Barrier`、`MPI_Wait`、`MPI_Waitall`、`MPI_Waitany`、`MPI_Waitsome`、`MPI_Win_fence`、`MPI_Win_lock`

MPI トレースデータを収集すれば、MPI 呼び出しによって MPI プログラムのパフォーマンスに問題が生じている場所を確認できます。パフォーマンスに関する問題の例としては、負荷平衡、同期遅延、通信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、標本パケットと呼ばれるパケット単位でコレクタが記録します。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 データといったカーネルからの実行統計、コンテキストスイッチ、および各種のページの常駐性 (ワーキングセットとページング) 統計が入っています。標本パケットに記録されるデータは、プログラムにとって大域的であり、パフォーマンスメトリックには変換されません。標本パケットを記録するプロセスのことを、標本収集と呼びます。

標本パケットは、次の状況で記録されます。

- 「デバッグ」ウィンドウや dbx において、ブレークポイントに達するなど何らかの理由でプログラムが停止したとき (これを行う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。
- 「デバッグ」->「パフォーマンスツールキット」->「コレクタを有効に」を選択するか、`dbx collector sample record` コマンドを使用したとき
- 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 (27 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信号が送信されたとき (`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
- 派生プロセスが作成される前と後

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ればイベント固有データをフィルタ処理できるので、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の対応付け

メトリックは、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、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 章を参照してください。

メトリックは、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。関数とロードオブジェクト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、プログラム内のあらゆる関数のメトリックを計算します。こうして得られたメトリックを関数レベルのメトリックといいます。

関数レベルのメトリック: 排他的、包括的、属性

パフォーマンスアナライザが求める関数レベルのメトリックは、排他的、包括的、属性の 3 種類があります。

- 関数の排他的メトリックは、関数本体内で発生したイベントから求められます。他の関数への呼び出しから発生したメトリックは含まれません。
- 包括的メトリックは、関数本体内とその関数が呼び出した関数内で発生したイベントから求められます。これには、他の関数への呼び出しから発生したメトリックが含まれます。
- 属性メトリックは、他の関数からの呼び出しまたは他の関数への呼び出しが原因で発生したメトリックです。つまり、属性メトリックは他の関数に原因があるメトリックということになります。

呼び出しスタックの一番下のみに現れる関数 (リーフ関数) では、その関数の排他的メトリックと包括的メトリックは同じになります。

排他的および包括的メトリックは、ロードオブジェクトについても求められます。ロードオブジェクトの排他的メトリックは、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を集計することによって求められるメトリックです。これに対し、ロードオブジェクトの包括的メトリックは、関数に対するのと同じ方法で求められるメトリックです。

関数の排他的および包括的メトリックは、その関数を通るあらゆる記録経路に関する情報を提供します。属性メトリックは、関数を通る特定の経路に関する情報を提供します。1 つのメトリックの内のどれだけの部分が特定の関数呼び出しに対応している

かを示します。呼び出しにかかわっている 2 つの関数を呼び出し元と呼び出し先と呼びます。呼び出しツリーにおいて、それぞれの関数の属性メトリックは次の意味を持ちます。

- 関数の呼び出し元の属性メトリックは、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を示します。呼び出し元の属性メトリックも合計したものが、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を示します。この場合、属性メトリックの合計と関数の排他的メトリックは、その関数の包括的メトリックに等しくなります。

呼び出し元または呼び出し先の属性メトリックと包括的メトリックを比較すると、さらに有用な情報が得られます。

- 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、その包括的メトリックのうち、他の関数への呼び出し、およびその呼び出し元自体の仕事が原因の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、その包括的メトリックのうち、他の関数からのその呼び出し先への呼び出しが占める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としては、以下があります。

- 排他的メトリックを参考に、メトリック値が大きい関数を発見する。
- 包括的メトリックを参考に、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調べます。
- 属性メトリックを参考に、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特定します。

属性メトリックの意味：例

図 2-1 は、呼び出しツリーにおける排他的、包括的、属性メトリックの関係例を部分的に表しています。ここでは、中央の関数の関数 C に注目します。この図には、関数のすべての呼び出しが含まれているわけではないことに注意してください。

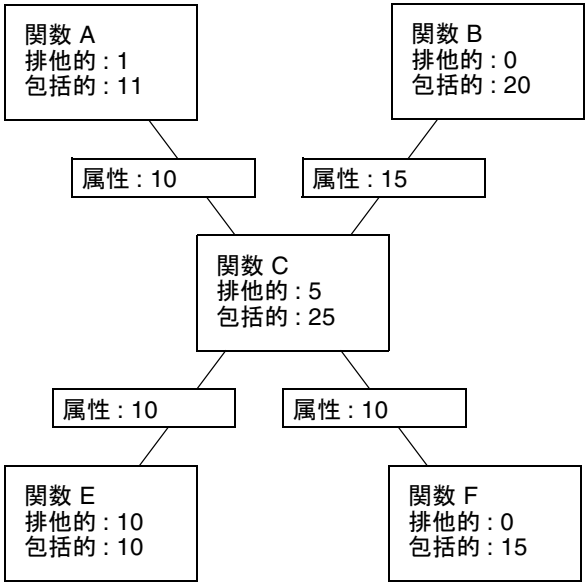

図 2-1 呼び出しツリーにおける排他的、包括的、属性メトリックの関係

関数 C は、関数 E および関数 F の 2 つの関数を呼び出し、これら 2 つの関数は、関数 C の包括的メトリックのうちの 10 単位の原因になっています。これらは、呼び出し先が原因の属性メトリックです。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。

関数 E では、呼び出し先が原因の属性メトリックと呼び出し元の包括的メトリックが同じですが、関数 F では異なります。このことは、関数 E が関数 C によってのみ呼び出されるの対し、関数 F が関数 C 以外の関数によっても呼び出されることを意味します。また、関数 E では、排他的メトリックと包括的メトリックが同じですが、関数 F では異なります。このことは、関数 F が他の関数を呼び出し、関数 E が他の関数を呼び出さないことを意味します。

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、関数 A の 10 単位と関数 B の 15 単位が、関数 C の包括的メトリックの原因になっています。これらは、呼び出し元が原因の属性メトリックです。この合計 (10+15) は、関数 C の包括的メトリックに等しくなります。

関数 A では、呼び出し元が原因の属性メトリックが、その包括的メトリックと排他的メトリックの差と等しくなりますが、関数 B では等しくありません。このことは、関数 A が関数 C のみ呼び出し、関数 B が関数 C 以外の関数も呼び出すことを意味します (実際には、関数 A は他の関数を呼び出している場合がありますが、時間が短かすぎて、実験データには現れないことがあります) 。

関数レベルのメトリック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ではなく、その関数全体のメトリックを表示します。このため、一連の再帰呼び出しのメトリックを 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) から求められる排他的メトリック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はその影響を受けます。

包括的メトリックは、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求められます。再帰呼び出しスタックにおいて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複数回加算されないようにします。

属性メトリックは、包括的メトリックから求められます。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仕事を終えた場合、再帰関数の包括的メトリックの原因になっているのは、その再帰関数であり、呼び出しを開始した関数は関わっていません。これは、再帰関数の上位にあるあらゆる呼び出しの包括的メトリックは、メトリックの複数回のカウントを回避するために、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が実行に要する時間は、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、およびデータの収集方法と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 章を参照してください。

この章では、以下について説明します。

- プログラムのコンパイルとリンク
- データ収集と解析のためのプログラムの準備
- データ収集に関する制限事項
- 収集データの格納場所
- 必要なディスク容量の概算
- データの収集
- `collect` コマンドによるデータの収集
- `dbx` の `collector` サブコマンドによるデータの収集
- 動作中のプロセスからのデータの収集
- MPI プログラムからのデータの収集
- ppgsz での `collect` の使用

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どのようなオプションを使用してもデータの収集と解析を行えるのが普通ですが、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を及ぼすオプションもあります。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、以下に説明します。

ソースコード情報

注釈付き「ソース」と「逆アセンブリ」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て、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=(stabs|dwarf) で指定されているように、STABS または DWARF2 のいずれかとすることができます。

現在は SPARC® 用の C コンパイラのためだけにハードウェアカウンタプロファイルの収集を可能にするデバッグ情報でコンパイルオブジェクトを作成するには、-xhwcprof -xdebugformat=dwarf と最適化レベルを指定してコンパイルします。(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、各成分オブジェクト (.o) 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。このことは 実行可能ファイルばかりでなく各種オブジェクトファイル内にも STABS シンボルを残す -xs オプションでリンクされている場合の STABS 形式のデバッグシンボルに対しても当てはまります。これは、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。実行可能ファイルとライブラリ自体にあるすべてのデバッグ用シンボルとともに、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。これは、コレクタを実行したときに、その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するべきではありません。システムライブラリを静的リンクしてしまうと、トレースデータを収集できなくなることがあります。コレクタライブラリ libcollector.so とのリンクも避けてください。

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、コンパイラが実行順序を変更できるため、プログラム内の行の順序どおりにコードが実行されなくなります。この場合、パフォーマンスアナライザは、このようにして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、コンパイラがテール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。最適化によって、展開の失敗が発生することがあります。詳細は、123 ページの「テール呼び出しの最適化」を参照してください。

Java プログラムのコンパイル

`javac` 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

ほとんどのプログラムの場合、データ収集と解析のためにプログラムに対して行う作業は特にありません。以下の処理のうち、いずれか 1 つでも行うプログラムの場合には、下記の該当する説明を読んでください。

- 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
- 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したい場合にも、該当する説明を読んでください。

動的割り当てメモリーの利用

多くのプログラムは、以下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)

初期値の設定としてメモリーの割り当て方法が明示的に規定されているのでない限り、プログラムが動的に割り当てられたメモリーの初期内容に依存することのないよう注意する必要があります。たとえば、malloc(3C) については、マニュアルページ内の calloc と malloc を比較してください。

動的に割り当てられたメモリーを使用するプログラムを単独で実行すると、ときどき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。そのときの症状としては、予期しない浮動小数点演算動作、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。これは、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たメモリーの内容に依存するアプリケーションのどれにも、潜在的な問題があります。別に明示的に規定されていない限り、オペレーティングシステムは、どのような内容であれ、動的に割り当てられたメモリー上の内容を自由に提供するためです。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た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。

こうした潜在的な問題の発見に役立つと思われるツールには、以下があります。

- 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

  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。

- Purify

システムライブラリの使用

コレクタは、さまざまなシステムライブラリの関数の上で割り込み処理することによって、トレースデータを収集し、完全なデータ収集を行います。以下は、コレクタがライブラリ関数の呼び出しで割り込みを行う状況を示しています。

- 同期待ちトレースデータの収集。コレクタは、Solaris™ スレッドライブラリ libthread.so[1] の関数上で割り込み処理を行います。同期待ちトレースは、Linux または Solaris 10 では利用できません。
- ヒープトレースデータの収集。コレクタは、malloc、realloc、memalign、および free の関数上で割り込み処理を行います。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 libmpi.so の関数上で割り込み処理を行います。MPI トレースは、Linux では利用できません。
- 時計データの完全性の確保。コレクタは setitimer で割り込み処理を行い、プログラムがプロファイルタイマ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 の関数で割り込み処理を行い、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、戻り値 -1 で復帰します。
- 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)、system(3C)、system(3F)、sh(3F)、popen(3C)、exec(2) の関数とそのバリアントで割り込み処理を行います。vfork の呼び出しは、fork1 の呼び出しに内的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 で割り込みを行なって、シグナルハンドラがこれらのシグナル用の専用シグナルハンドラであるかどうかを確認します。

割り込みが成功しない状況もあります。

- 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となったりする可能性があります。

1. Solaris 8 上のデフォルトのスレッドライブラリ /usr/lib/libthread.so (T1) では、プロファイル時に問題が発生します。LWP にスレッドがスケジュールされていないと、プロファイリング割り込みが破棄される場合があります。この場合は、本当の LWP 時間を大きく下回る LWP 合計時間が報告されることがあります。状況によっては、内部ライブラリの相互排他ロック (mutex) にアクセスする際にセグメンテーション違反も発生して、アプリケーションがクラッシュする場合もあります。これは、LD_LIBRARY_PATH 設定の先頭に /usr/lib/lwp を付加して、代替スレッドライブラリ (/usr/lib/lwp/libthread.so、これが T2 です) を使用することで回避できます。Solaris 9 では、デフォルトライブラリは T2 で、これは libc ライブラリに組み込まれています。

シグナルハンドラの使用

コレクタは、全実験用の SIGPROF と**ハードウェアカウンタ実験専用の** SIGEMT の 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。コレクタはこれらのシグナルのそれぞれを対象とし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、他のシグナルは、インストールされている他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の専用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。フラグが設定されると、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場合には、プログラムの動作が変わる可能性があります。動作が変化する重要な例としては、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 I/O ライブラリ libaio.so があります。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は必ず、キャンセルシグナルの到着が非同期入出力操作の取り消しに間に合わないほど遅れます。

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、その後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、SIGPROF と SIGEMT のシグナルが渡され、かつパフォーマンスデータが失われないことを確実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の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 の使用

setuid(2) の使用とパフォーマンスデータの収集を困難にする、ダイナミックローダによって課される制約があります。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。

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には、コレクタ共有ライブラリ `libcollector.so` に入っている API 関数をプログラムで使用します。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_thread_pause(unsigned int t);
void collector_thread_resume(unsigned int t);
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、28 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

C/C++ インタフェースにアクセスする方法は 2 通りあります。

- `collectorAPI.h` を取り込み、`-lcollectorAPI` (土台の `libcollector.so` API 関数の有無をチェックする本当の関数が含まれている API) を使用してリンクする。

  この方法では、API ライブラリとリンクする必要があります。この環境は、あらゆる環境下で有効です。有効な実験がない場合、API 呼び出しは無視されます。

- `libcollector.h` (土台の `libcollector.so` API 関数の有無をチェックするマクロを含む API) を取り込む。

  この方法は、main 実行可能ファイル内と、プログラムの起動と同時にデータ収集を開始する場合にのみ有効です。`dbx` を使用して、プロセスに接続する場合、あるいはプロセスが `dlopen` する共有ライブラリ内で使用した場合は、必ずしも機能しないことがあります。この方法は、下位互換性を維持する目的で提供されています。

注意 – どんな言語を使用している場合も、プログラムを `-lcollector` とリンクすることは避けてください。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。

Fortran インタフェース

Fortran API の `libfcollector.h` ファイルには、ライブラリに対する Fortran インタフェースが定義されています。このライブラリを使用するには、`-lcollectorAPI` を使用してアプリケーションを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"
```

注意 – どんな言語を使用している場合も、プログラムを `-lcollector` とリンクすることは避けてください。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、お使いのアプリケーションは `<installation-directory>/lib/collector.jar` (ここで、`<installation-directory>` は Sun のコンパイラとツールがインストールされているディレクトリ) を指すクラスパスで呼び出さなければならないことに注意してください。

```
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.threadPause(Thread thread)
CollectorAPI.threadResume(Thread thread)
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 関数の呼び出しを迂回するマクロが入っています。この場合、関数は動的に読み込まれません。ただし、一部環境ではうまく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。マクロを使用していないため、collectorAPI.h を利用する方が安全であり、この関数は関数を直接参照します。

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には、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。collector_thread_pause() および collector_thread_resume() 以外は、API 関数が行うアクションの対象はプロセスであって、個々のスレッドではありません。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、他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。データ収集を個々のスレッドレベルで制御するには、collector_thread_pause() 関数と collector_thread_resume() 関数を使用します。これらの関数の使用方法として、1 つのマスタスレッドにそれ自体を含むすべてのスレッドに対してすべての呼び出しを行わせる方法と、各スレッドにそれ自体に対してのみ呼び出しを行わせる方法があります。その他の使用方法では、結果が予測できないものになるおそれがあります。

C/C++、Fortran および Java API 関数

以下では、データ収集に関係する API 関数について説明します。

- C と 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他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な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()`

  実験へのイベント固有データの書き込みを `collector_pause()` を呼び出した後に再開します。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 と C++ のみ : `collector_thread_pause(unsigned int t)`

  Java: `CollectorAPI.threadPause(Thread t)`

  引数リストで指定したスレッドから実験へのイベント固有データの書き込みを停止します。引数 t は、C/C++ の場合は POSIX スレッド識別子、Java の場合は Java スレッドです。実験がすでに終了したか、実験がアクティブでないか、あるいは、そのスレッドに対するデータの書き込みがすでにオフになっている場合、呼び出しは無視されます。この関数は、たとえデータの書き込みが大域的に有効でも、指定したスレッドからのデータの書き込みを停止します。デフォルトでは、個々のスレッドのデータの記録がオンに設定されます。

- C と C++ のみ : `collector_thread_resume(unsigned int t)`

  Java: `CollectorAPI.threadResume(Thread t)`

  引数リストで指定したスレッドから実験へのイベント固有データの書き込みを再開します。引数 t は、C/C++ の場合は POSIX スレッド識別子、Java の場合は Java スレッドです。実験がすでに終了したか、実験がアクティブでないか、あるいは、そのスレッドに対するデータの書き込みがすでにオンになっている場合、呼び出しは無視されます。データは、データの書き込みが大域的に有効にされ、スレッドに対して有効にされているときのみ、実験に書き込まれます。

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

  Java: `CollectorAPI.terminate`

  データを収集している実験を終了します。以降、データの収集は行われませんが、プログラムは正常に動作を続けます。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、コンパイル時にそのデータ空間に関数を動的に取り込む場合、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で見るには、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。

以下では、データ収集に関係する API 関数について説明します。

- `collector_func_load()`

実験での記録のため、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下表に示します。

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る関数の名前。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`	関数の記述に使用する任意の文字列。`NULL` を使用できます。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されます。
`sourcename`	関数の構築元であるソースファイルのパス。`NULL` を使用できます。注釈付きソースリストには、ソースファイルが使用されます。
`vaddr`	関数が読み込まれたアドレス。

表 3-1　collector_func_load() のパラメータリスト (続き)

パラメータ	定義
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。オフセットは数字の昇順にする必要があります。行番号の順序は任意です。lntable が NULL である場合、関数のソースリストは利用できません。ただし、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、オペレーティング環境、プログラムの実行方法、またはコレクタそのもの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。

異なるデータ型の同時収集に対する制限はありません。すなわち、どのようなデータ型でも他のデータ型とともに収集できます。

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化に使用する時計の分解能は、特定のオペレーティング環境により異なります。最大値は 1 秒です。プロファイル間隔の値は、時計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計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。

Solaris 8 オペレーティングシステムの初期バージョンでは、プロファイルにシステムクロックが使用されます。高分解能のシステムクロックを有効にすることを選択しないかぎり、このシステムクロックの分解能は 10 ミリ秒です。このためには、`/etc/system` ファイルに次の行を追加してシステムを再起動します (スーパーユーザー権限が必要です)。

```
set hires_tick=1
```

Solaris 9 のオペレーティングシステムと Solaris 8 の後期のバージョンのオペレーティングシステムでは、高分解能のプロファイル化に高分解能のシステムクロックを有効にする必要はありません。

時間プロファイルによるランタイムのディストーションとディレーション

時間プロファイルでは、SIGPROF シグナルがターゲットに送られたときにデータを記録します。それによってディレーションが発生し、そのシグナルが処理されて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、ディレーションは大きくなります。一定の範囲までは、時間プロファイルによりある程度のディストーションが表れますが、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ディレーションが大きくなることから生まれます。

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、56 ページの「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。

トレースによるランタイムのディストーションとディレーション

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をディレートさせます。時間プロファイルが完了すると、時間データはトレースイベントで誘導されたディレーションによりディストートされます。

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ついては、以下の制限事項があります。

- 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。その他のシステムでは、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行えません。UltraSPARC® III プロセッサより前の 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。
- 1 つの実験で最大 2 つのハードウェアカウンタのデータを記録できます。3 つ以上のハードウェアカウンタ、または同じレジスタを使用するカウンタのデータを記録するには、複数の実験を行う必要があります。
- `cpustat`(1) が動作しているシステムで、ハードウェアカウンタのオーバーフローデータを収集することはできません。これは、`cpustat` がすべての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 `libcpc`(3) を使用して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` ライブラリ関数上で割り込み処理を行い、コレクタからの呼び出しではなかった場合には `-1` の戻り値で復帰します。
- `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について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ランタイムのディストーションとディレーション

ハードウェアカウンタのプロファイルでは、SIGEMT がターゲットに送られたときにデータを記録します。それによってディレーションが発生し、そのシグナルが処理されて呼び出しスタックが展開されます。時間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で他の部分より高速にイベントを生成する場合があり、そのコード部分にディレーションが表示されます。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が大きくディストートする場合があります。同様に、イベントによっては、スレッド間で不均等に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

派生プロセスに関するデータを収集するには、次の制限事項があります。

- コレクタでの作業対象とする派生プロセスすべてについてデータを収集するには、`-F on` または `-F all` オプションを指定して `collect` コマンドを使用する必要があります。`collect` オプションの `-F on` 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` を使用すると、コレクタは、`system`、`popen`、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たどります。
- 個々の派生プロセスのデータを収集するには、プロセスに `dbx` を接続する必要があります。詳細は、56 ページの「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。
- 個々の派生プロセスのデータを収集するには、別個の `dbx` を使って各プロセスに接続し、コレクタを有効にする必要があります。ただし、Java™ のデータ収集は、現在 `dbx` のもとではサポートされていないことに注意してください。

Java プロファイルに関する制限事項

Java™ プログラムに関するデータを収集することはできますが、次の制限事項があります。

- バージョン 1.4.2_02 以上の Java™ 2 Software Development Kit を使用する必要があります。クラッシュすることがあるため、1.4.2 や 1.4.2_01 は使わないでください。Java 仮想マシン[2]へのパスは、次の 4 つの環境変数のうちの 1 つで指定する必要があります。`JDK_1_4_HOME`、`JDK_HOME`、`JAVA_PATH`、`PATH` コレクタはこれらの環境変数に定義されている `java` のバージョンが ELF 実行可能ファイルであるかどうかを確認し、ELF 実行可能ファイルでない場合には、使用した環境変数とフルパス名を示すエラーメッセージを出力します。
- データの収集には、`collect` コマンドを使用する必要があります。`dbx collector` サブコマンドや IDE のデータ収集機能を使用することはできません。
- JVM 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。
- 64 ビットの JVM™ を使用するには、`-j on` フラグを使用して 64 ビット JVM をターゲットとして指定する必要があります。64 ビット JVM によるデータ収集では、`java -d64` 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。

2. 「Java 仮想マシン (JVM)」という用語は、Java プラットフォーム用仮想マシンを意味します。

- 非 Java アプリケーションは、`libjvm.so` を動的に開いて、Java™ クラスをこれに渡すことができます。このようなアプリケーションのプロファイルをサポートするための対策は、初期収集呼び出しに `-j on` を設定しないで、呼び出された JVM に `-Xruncollector` オプションが渡されるようにすることです。

1.4.2_02 より前の JVM バージョンを使用すると、次のようにデータが損なわれます。

- JVM 1.4.2_01: このバージョンでは、データの収集中に JVM がクラッシュすることがあります。
- JVM 1.4.2: このバージョンでは、データの収集中に JVM がクラッシュすることがあります。
- JVM 1.4.1: Java™ 表現は正しく記録され表示されますが、すべての JVM ハウスキーピングが JVM 関数そのものとして表示されます。データ空間で JVM コードを実行するのに費やされるある時間は、JVM から提供されるコード領域の名前とともに表示されます。大量の時間が `<Unknown>` 関数に表示されるのは、JVM で作成されたコード領域の中に名前が付いていないものがあるからです。また、JVM 1.4.1 には、プロファイルされているプログラムをクラッシュさせるようなさまざまなバグがあります。
- JVM 1.4.0: Java™ 表現は不可能であり、大量の時間が <**未知**> 関数に示されます。
- 1.4.0 より前の JVM : 1.4.0 より前の JVM による Java™ アプリケーションのプロファイル化はサポートされません。

Java プログラミング言語で書かれたアプリケーションのランタイムのディストーションとディレーション

Java™ プロファイル化では JVMPI インタフェースを使用するので、実行のディストーションとディレーションが発生するおそれがあります。

時間とハードウェアカウンタのプロファイル化の場合、データ収集プロセスは JVM™ へさまざまな呼び出しを作成し、シグナルハンドラでのイベントのプロファイル化を処理します。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は、Java™ プログラムのランタイムをディレートさせます。そのようなディレーションは通常 10% より少なくなります。

デフォルトのガーベッジコレクタは JVMPI をサポートしますが、サポートを行わないガーベッジコレクタもあります。そのようなガーベッジコレクタを指定するデータ収集の実行は、致命的エラーになります。

ヒーププロファイルの場合、データ収集プロセスではメモリーの割り当てとガーベッジコレクションを記述する JVMPI イベントを使用するため、ランタイムで大きいディレーションが発生するおそれがあります。大半の Java™ アプリケーションではこれらのイベントを数多く生成するので、実験が大きくなり、データ処理の拡張性問

題が発生します。さらに、これらのイベントを要求すると、ガーベッジコレクタはインライン化された割り当てを無効にするので、さらに長くなった割り当てパスのための追加 CPU 時間のコストがかかります。

同期トレースの場合は、データ収集でその他の JVMPI イベントを使用するので、アプリケーション内のモニタ競合の量に比例してディレーションが発生します。

収集データの格納場所

プログラムの実行中に収集されたデータを「実験」といいます。実験は、1 つのディレクトリに格納される 1 組のファイルで構成されます。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自分専用アーカイブも作成します。これらのオブジェクトには、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誤りが含まれることがあります。このため、できる限り、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中に格納されます。

実験名

実験のデフォルト名は、`test.1.er` です。接尾辞 `.er` は、必須です。この接頭辞の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。

実験名 *.n*`.er` (*n* は正の整数) という形式の名前を使用する場合、標本コレクタは、以降の実験名の *n* の部分を自動的に 1 ずつインクリメントします。たとえば、`mytest.1.er`、`mytest.2.er`、`mytest.3.er` のようになります。コレクタはまた、実験がすでに存在する場合も *n* をインクリメントし、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* のインクリメントを繰り返します。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、デフォルト時に現在のディレクトリに格納される実験グループファイルにおいて定義されます。実験グループファイルは、1 行のヘッダー行の後に 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。ファイル名の末尾が `.erg` でない場合はエラーとなり、ファイル名は受け付けられません。実験グループを作成すると、そのグループ名で実行したすべての実験がグループに追加されます。

最初の行が、次の行であるプレーンテキストファイルを作成し、実験グループを作成することができます。

```
#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。

MPI プロセスごとに実験が 1 つ作成される MPI プログラムから収集された実験では、デフォルトの実験名が異なります。デフォルトの実験名は `test.`*m*`.er` で、*m* はそのプロセスの MPI ランクです。*group*`.erg` という実験グループを指定した場合、デフォルトの実験名は *group*`.`*m*`.er` です。実験名を指定すると、これらのデフォルト名がオーバーライドされます。詳細は、59 ページの「MPI プログラムからのデータの収集」を参照してください。

派生プロセスの実験は、次のとおり、その系統に基づいて命名されます。派生プロセスの実験名は、その親の実験名に下線、コード文字、数字を追加して作成されます。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 の索引です (成功したかどうか)。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には、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を念頭に置いてください。

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する必要があります。

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、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

アナライザ GUI から、または `setpath` および `addpath` 指令を使って、検索順序を変更したり、他の検索ディレクトリを追加できます。

プログラムの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を確実に調査対象とするには、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には、プログラムを `-xs` とリンクし、ソース行とファイル位置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` オプションまたは `dbx collector archive` コマンドを使用して、実験にロードオブジェクトを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

この節では、実験の記録に必要な空きディスク容量を概算するにあたって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、イベント固有データとプログラム構造 (呼び出しスタック) に依存するデータとが入っています。データ型に依存するデータのサイズは、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 SPARC® アーキテクチャーでは 8 バイト) です。データパケットは、実験の LWP ごとに記録されます。ここで、Java™ プログラムの場合は、対象の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ため、ディスクに書き込まれるデータが増えることに注意してください。

プロファイルデータパケットが記録される速度は、時間データのプロファイル間隔とハードウェアカウンタデータのオーバーフロー値によって制御されます。ただし、これらのパラメータによってデータ収集のオーバーヘッドが変わるため、データの品質やプログラムパフォーマンスの歪みにも影響があります。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。また、値が小さければ、データ量が多くなります。

プロファイル間隔が 10 ミリ秒で呼び出しスタックが小さく、パケットサイズが 100 バイトの時間ベースのプロファイル実験の場合、データは LWP 1 つあたり毎秒 10K バイトで記録されます。オーバーフロー値を 1000000、パケットサイズを 100 バイトとして、750MHz のプロセッサで実行された CPU サイクルと命令のデータを収集する、ハードウェアカウンタオーバーフローのプロファイル実験の場合は、LWP 1 つ

あたり毎秒 150K バイトの速度です。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。

実験サイズの概算では、アーカイブファイルに使用す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)。必要なディスク領域のサイズを確定できない場合は、実験を短時間だけ行ってみてください。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調整することによって、実験全体のサイズの概算を求めることができます。

コレクタは、ディスク領域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。現在のところ、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は、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で必要となると思われる容量の方が大きい場合には、全体ではなく一部だけのデータを収集してもよいでしょう。それには、`collect` コマンドや `dbx collector` サブコマンドを使用するか、コレクタ API の呼び出しをプログラムに挿入します。`collect` コマンドや `dbx collector` サブコマンドを使用して、収集するプロファイルデータやトレースデータの総量を制限することもできます。

注 – パフォーマンスアナライザが読み込めるパフォーマンスデータは、2 G バイトまでです。

データの収集

パフォーマンスデータの収集は、スタンドアロンのパフォーマンスアナライザまたはアナライザモジュールを使用して、いくつかの方法で行うことができます。

- コマンド行から `collect` コマンドを使用する (41 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 コマンド行ツールのデータ収集時のオーバーヘッドは、`dbx` や IDE にあるデバッガの「コレクタ」ダイアログよりも小さいため、他の方法より、この方法の方が優れていることがあります。
- パフォーマンスアナライザでパフォーマンスツールにある「収集」ダイアログを使用する (パフォーマンスアナライザのオンラインヘルプの「パフォーマンスツールの「収集」ウィンドウからのパフォーマンスデータの収集」を参照)。
- デバッガの「コレクタ」ダイアログを使用する (パフォーマンスアナライザのオンラインヘルプの「デバッガ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 (50 ページの「dbx の collector サブコマンドによるデータの収集」および IDE にある「デバッグ」オンラインヘルプの「collector コマンド」を参照)。

次のデータ収集機能は、パフォーマンスツールの「収集」ダイアログと collect コマンドでのみ利用できます。

- Java™ プログラム上でのデータの収集。IDE にあるデバッガの「コレクタ」ダイアログまたは dbx の collector コマンドを使って Java™ プログラムに関するデータを収集した場合、実際に収集される情報は、Java プログラムではなく Java™ 仮想マシンに関する情報です。
- 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
```

collect-options は collect コマンドのオプション、*program* はデータの収集対象のプログラム名、*program-arguments* はそのプログラムに対する引数です。

コマンド引数を何も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 10 ミリ秒になります。

コマンドオプションの一覧とプロファイルに使用可能なハードウェアカウンタ名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。

```
% collect
```

ハードウェアカウンタの一覧については、8 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。34 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。

データ収集関連のオプション

データ収集のオプションは、どのような種類のデータを収集するのかを制御します。データの種類については、5 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何も指定しなかった場合は、デフォルトで -p on となり、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が明示的に無効にされ、同期待ちのトレース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のどちらも有効でない場合、collect コマンドはエラー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* には次のいずれかの値を指定できます。

- 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 (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 (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 (1 ミリ秒) で時間ベースのプロファイルを有効にします。Solaris 7 のオペレーティングシステムと Solaris 8 の初期のバージョンのオペレーティングシステムでは、高分解能プロファイル化を明示的に有効にする必要があります。高分解能のプロファイルについては、32 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* のデフォルト単位はミリ秒です。*value* を整数または浮動小数点数として指定することができます。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したり、u を付けてマイクロ秒を選択することができます。プロファイル間隔は、システム時間の分解能の倍数である必要があります。システム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。システム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。

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*[, *value*[, *counter2*[, *value2*]]]

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。カウンタ名の *counter* および *counter2* は次のいずれかです。

- カウンタ名の別名

- cputrack(1) によって使用されるような内部名。イベントレジスタのいずれかをカウンタに使用可能な場合は、内部名に /0 または /1 を付加することによって指定できます。

2 つのカウンタを指定する場合は、それぞれ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。どちらのレジスタでもカウントできるカウンタもあります。

使用可能なカウンタの一覧を表示するには、引数を指定せずに collect コマンドを端末ウィンドウに入力します。10 ページの「ハードウェアカウンタのリスト」に、カウンタの一覧があります。

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はイベントデータパケットに保存されます。

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オーバーフローイベントが記録された時点で数えられた、イベントの数です。*value* と *value2* を使用すれば、オーバーフロー値を指定できます。次のいずれかの値を設定します。

- hi[gh] - 指定したカウンタの高分解能値を有効にします。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] - 指定したカウンタの低分解能値を有効にします。
- *number* - オーバーフロー値。正の整数を指定します。
- on または NULL 文字列 - デフォルトのオーバーフロー値を有効にします。

デフォルトでは、事前に各カウンタに定義されている通常のしきい値が使用され、これらの値はカウンタの一覧に表示されます。34 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しないで -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* には次のいずれかの値を指定できます。

- all - しきい値ゼロで同期待ちの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 - 同期待ちの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 と同等です。
- off - 同期待ちトレースを無効します。

- `on` - デフォルトのしきい値 (実行時の測定で値を決定) で同期待ちのトレースを有効にします。`calibrate` と同等です。
- *value* - しきい値を指定した**値**に設定します。ミリ秒数を示す正の整数を指定します。

同期待ちのトレースデータは、Java™ モニタについては記録されません。

`-H` *option*

ヒープトレースデータを収集します。*option* には次のいずれかの値を指定できます。

- `on` - ヒープの割り当て要求と割り当て解除要求のトレースを有効にします。
- `off` - ヒープのトレースを無効にします。

デフォルトの場合、ヒープのトレースは無効となっています。

`-m` *option*

MPI トレースデータを収集します。*option* には次のいずれかの値を指定できます。

- `on` - MPI 呼び出しのトレースを有効にします。
- `off` - MPI 呼び出しのトレースを無効にします。

デフォルトの場合、MPI のトレースは無効となっています。

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の詳細については、14 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* には次のいずれかの値を指定できます。

- `off` - 定期的標本収集を無効にします。
- `on` - デフォルトの標本収集間隔 (1 秒) による定期的な標本収集を有効にします。
- *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。

デフォルトの場合、1 秒間隔による定期的標本収集が有効になっています。

実験制御関連のオプション

-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* には次のいずれかの値を指定できます。

- on - 関数 fork、exec、およびそのバリアントが作成す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 - 派生プロセスに関する実験を記録しません。

-F on オプションを指定すると、コレクタは、fork(2)、fork1(2)、fork(3F)、vfork(2)、および exec(2) の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たどります。vfork の呼び出しは、fork1 の呼び出しと内的に置換されます。

-F all オプションを指定すると、コレクタは、system(3C)、system(3F)、sh(3F)、および popen(3C)、その他類似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、さらには関係する派生プロセスをたどります。

-F on または -F all 引数が指定された場合、コレクタは、親の実験内の派生プロセスごとに新しい実験を 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。

- 英字「f」は fork、英字「x」は exec、英字「c」は他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) または他の呼び出しの索引です。

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 呼び出しを使っ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。

アナライザおよび er_print は、親の実験が読み取られたときに自動的の派生プロセスの実験を読み取りますが、派生プロセスの実験は、データ表示の対象として選択されていません。

コマンド行から表示するデータを選択するには、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 回目のフォークのデータを見るには、以下のように指定します。

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

```
analyzertest.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。

アナライザの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。親の実験のみ選択された実験の一覧が表示されます。これを選択解除して、対象の派生実験を選択します。

-j *option*

非標準 Java™ インストレーションの Java™ プロファイルを有効にします。または、Java HotSpot™ 仮想マシンによってコンパイルされたメソッドに関するデータを収集するかどうかを選択します。*option* には次のいずれかの値を指定できます。

- `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™ スタックを記録しようとします。
- `off` - Java HotSpot™ 仮想マシンによってコンパイルされたメソッドを認識しようとしません。

`.class` ファイルまたは `.jar` ファイルに関するデータを収集する場合には、このオプションは不要です。ただし、`java` 実行可能ファイルのパスが `JDK_1_4_HOME`、`JDK_HOME`、`JAVA_PATH`、`PATH` の環境変数のいずれかに含まれていることが条件です。それから *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。

これらの変数のどれにも `java` 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に、このオプションを使用するとよいでしょう。このオプションを使用する場合、*program* は 1.4.2_02 以上のバージョンの Java™ 仮想マシンにする必要があります。ただし、`collect` コマンドは *program* が JVM マシンであるかどうかと ELF 実行可能ファイルであるかどうかを確認し、そうでない場合にはエラーメッセージを出力します。

64 ビット JVM マシンを使用してデータを収集する場合、32 ビット JVM マシン用の `java` 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。*program* またはこの項で説明している環境変数の 1 つに 64 ビット JVM マシンのパスを指定してください。

-l *signal*

signal というシグナルがプロセスに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、シグナル番号のど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、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。シグナルは、kill(1) 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。

プログラムに専用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には、-l によって指定するシグナルがインターセプト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。

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

-x

デバッガがそのプロセスに接続できるように、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には、ignore PROF と ignore EMT の dbx コマンドを使用して、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 というシグナルを使用して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、シグナル番号のど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、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。シグナルは、kill(1) 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。それ以外の場合は、一時停止状態でコレクタが起動します。-y オプションが指定されなかった場合は、記録状態で起動します。

プログラムに専用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には、-l によって指定するシグナルがインターセプト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。

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

出力関連のオプション

-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* 文字列の末尾が「.er」でない場合、collect コマンドはエラーメッセージを出力して終了します。

-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。指定したディレクトリが存在しない場合、collect コマンドはエラーメッセージを出力して終了します。

-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 コマンド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、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したとすれば、実験グループはディレクトリ *directory-name* に設定され、それ以外は現在のディレクトリに設定されます。

-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しなければならないかどうかを管理します。option には次のいずれかの値を指定できます。

- 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をコピーして実験に保管します。

実験データが記録された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りたい場合は、-A copy を指定する必要があります。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は実験にコピーされません。実験データをコピーする先のマシン上でこれらのファイルにアクセスできるかどうか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* メガ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。この限界値は概数にすぎず、この値を超えることは可能です。

限界値に達するとプロファイルデータの記録は停止されますが、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。

記録データ量のデフォルト限界値は、2000M バイトです。この限界値が選択されたのは、2G バイトを超えるデータの実験をパフォーマンスアナライザが処理することができないためです。制限を外すには、*size* を `unlimited` または `none` に設定します。

-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

-C *comment*

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。これは「ドライラン」オプションです。

-R

パフォーマンスツール 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。

dbx の collector サブコマンドによるデータの収集

dbx からコレクタを実行するには、以下の操作を行います。

1. **次のコマンドを使用し、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、データ型を選択し、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。以下に、collector の全サブコマンドをまとめます。

データ収集関連のサブコマンド

ここでは、コレクタが収集するデータの種類を制御するサブコマンドをまとめています。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を収集するかどうかを制御します。*option* には次のいずれかの値を指定できます。

- `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* には次のいずれかの値を指定できます。
  - `on` - デフォルトのプロファイル間隔 (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 (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 (1 ミリ秒) を使用します。Solaris 8 の初期のバージョンのオペレーティングシステムでは、高分解能プロファイル化を明示的に有効にする必要があります。高分解能のプロファイルについては、32 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* のデフォルト単位はミリ秒です。*value* を整数または浮動小数点数として指定することができます。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したり、`u` を付けてマイクロ秒を選択することができます。プロファイル間隔は、システム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。また、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 10 ミリ秒です。

デフォルトの場合、`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を収集するかどうかを制御します。ハードウェアカウンタオーバーフローのプロファイル機能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* には次のいずれかの値を指定できます。

- 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。この一覧の形式については、10 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 *name value* [*name2 value2*] - ハードウェアカウンタ名として *name* を指定し、オーバーフロー値として *value* を設定します。*name2* と *value2* に、2 つめのハードウェアカウンタ名とそのオーバーフロー値を指定できます。オーバーフロー値は次のいずれかです。
  - hi[gh] - 指定したカウンタの高分解能値を有効にします。省略形 h もサポートされています。
  - lo[w] - 指定したカウンタの低分解能値を有効にします。
  - *number* - オーバーフロー値。正の整数を指定します。
  - on - デフォルトのオーバーフロー値が使用されます。

  各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はイベントデータパケットに保存されます。

デフォルトの場合、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。

34 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*

同期待ちのトレースデータを収集するかどうかを制御します。*option* には次のいずれかの値を指定できます。

- `on` - デフォルトのしきい値で同期待ちのトレースを有効にします。
- `off` - 同期待ちトレースを無効します。
- `threshold` *value* - 記録する最小同期遅延のしきい値を設定します。*value* には次のいずれかの値を指定できます。
  - `all` - しきい値ゼロを使用します。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` - 実行時に測定を行うことによってしきい値を設定します。`on` と同等です。
  - `off` - 同期待ちトレースを無効にします。
  - `on` - デフォルトのしきい値 (実行時の測定で値を決定) を設定します。`calibrate` と同等です。
  - *number* - しきい値を指定した値に設定します。ミリ秒数を示す正の整数を指定します。`value` が 0 であれば、すべてのイベントがトレースされます。

デフォルトの場合、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を収集するかどうかを制御します。*option* には次のいずれかの値を指定できます。

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

デフォルトの場合、コレクタはヒープのトレースデータを収集しません。

`mpitrace` *option*

MPI トレースデータを収集するかどうかを制御します。*option* には次のいずれかの値を指定できます。

- `on` - MPI 呼び出しのトレースを有効にします。
- `off` - MPI 呼び出しのトレースを無効にします。

デフォルトの場合、コレクタは MPI トレースデータを収集しません。

`sample` *option*

標本収集モードを制御します。*option* には次のいずれかの値を指定できます。

- `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、依然として有効のままです。
- `period` *value* - 標本収集の間隔を *value* に設定します (秒単位)。

デフォルトの場合、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は標本を記録しません。

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

実験制御関連のサブコマンド

disable

データの収集を無効にします。プロセスが動作中でデータを収集中の場合は、その実験が終了し、データ収集が無効になります。プロセスが動作中でデータ収集がすでに有効の場合、このサブコマンドは無視され警告が出されます。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

データの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すでに有効の場合、このサブコマンドは無視され警告が出されます。プロセスが動作していない場合は、以降の実行について、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。標本ポイントは引き続き記録されます。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

一時停止されていたデータの収集を再開します。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*

標本パケットはラベル名を付けて記録します。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。

出力関連のサブコマンド

次のサブコマンドは、実験の格納オプションを指定します。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* には次のいずれかの値を指定できます。

- on - ロードオブジェクトの通常の保管に設定します。
- off - ロードオブジェクトの保管なしに設定します。
- copy - 通常の保管のほかに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、このコマンドは無視されて警告が出されます。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* メガ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。この限界値は概数にすぎず、この値を超えることは可能です。

限界値に達するとプロファイルデータの記録は停止されますが、実験はオープン状態となり、標本ポイントは引き続き記録されます。

記録データ量のデフォルト限界値は、2000M バイトです。この限界値が選択されたのは、2G バイトを超えるデータの実験をパフォーマンスアナライザが処理することができないためです。制限を外すには、*value* を unlimited または none に設定します。

store *option*

実験ファイルの格納先を指定します。実験がアクティブな場合、このコマンドは無視されて警告が出されます。*option* には次のいずれかの値を指定できます。

- directory *directory-name* - 実験ファイルと実験グループの格納先のディレクトリを指定します。指定したディレクトリが存在しない場合、このサブコマンドは無視されて警告が出されます。
- experiment *experiment-name* - 実験名を指定します。指定した実験名の末尾が .er でない場合、このサブコマンドは無視され、警告が出されます。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、37 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* - 実験グループ名を指定します。指定されたグループ名の末尾が .erg でない場合、このサブコマンドは無視され、警告が出されます。グループが存在する場合は、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。

情報関連のサブコマンド

show

コレクタを制御するすべてのオプションの現在値を表示します。

status

開かれている実験の状態を報告します。

動作中のプロセスからのデータの収集

コレクタでは、動作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 (コマンド行バージョンと IDE のどちらでも可) 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。

注 – IDE のパフォーマンスアナライザの起動については、パフォーマンスアナライザの Readme を参照してください。これはファイル `/opt/SUNWspro/docs/ja/index.html` のマニュアルの索引で探すことができます。Sun Studio 9 ソフトウェアが `/opt` ディレクトリにインストールされていない場合は、システム管理者に実際のパスをお尋ねください。

プロセスが dbx の制御下にない場合は、プロセスに dbx を接続してから、パフォーマンスデータを収集し、収集を終えたらプロセスから切り離します。この後、プロセスはそのまま動作を継続します。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 の制御下にない動作中のプロセスからデータを収集するには、以下の操作を行います。

1. **プログラムのプロセス ID (PID) を調べます。**

   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。その他の場合は、次のコマンドを使用し、プログラムの PID を調べることができます。

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

2. **プロセスに接続します。**

   - IDE の「デバッグ」メニューから「デバッグ」→「接続」を選択し、ダイアログボックスでプロセスを選択します。この方法についての詳細は、オンラインヘルプを参照してください。
   - dbx から次のコマンドを入力します。以下とおり

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

   dbx をまだ実行していない場合は、次のコマンドを入力します。

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

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

3. **データの収集を開始します。**

   - IDE の「デバッグ」メニューから「パフォーマンスツールキット」→「コレクタを有効に」を選択し、データ収集パラメータをダイアログボックスで設定します。次に、「デバッグ」→「継続」を選択してプロセスの実行を再開します。

- dbx からの場合は、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。

- IDE では、「デバッガ」ウィンドウの「セッション」ビューに表示されているプロセスのセッションを右クリックし、コンテキストメニューから「完了」を選択します。「セッション」ビューが表示されていない場合には、「デバッガ」ウィンドウ上部にある「セッション」ボタンをクリックします。
- dbx からの場合は、次のコマンドを入力します。

```
(dbx) detach
```

トレースデータを収集する場合は、プログラムを実行する前に、コレクタライブラリの libcollector.so を事前に読み込んでおく必要があります。これは、このライブラリによって、データの収集を可能にする本当の関数にラッパーが提供されるためです。また、コレクタは、他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。コレクタライブラリを事前に読み込まなかった場合、ラッパー関数は挿入できません。コレクタがシステムライブラリ関数上で割り込み処理を行う方法の詳細については、24 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 を事前に読み込むには、環境変数を使用してライブラリ名とライブラリパスの両方を設定する必要があります。ライブラリ名を設定するには、環境変数 LD_PRELOAD を使用します。ライブラリをパスに設定するには、環境変数 LD_LIBRARY_PATH、LD_LIBRARY_PATH_32、LD_LIBRARY_PATH_64 を使用します (LD_LIBRARY_PATH は、_32 と _64 バリアントが定義されていない場合に使用します)。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 ライブラリを事前に読み込むための環境変数の設定

環境変数	値
LD_PRELOAD	libcollector.so
LD_LIBRARY_PATH	/opt/SUNWspro/lib
LD_LIBRARY_PATH_32	/opt/SUNWspro/lib
LD_LIBRARY_PATH_64	/opt/SUNWspro/lib/v9

/opt/SUNWspro 以外のディレクトリに Sun のコンパイラとツールがインストールされている場合は、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 および LD_LIBRARY_PATH の設定を削除し、同じシェルから起動される他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。

すでに実行中の MPI プログラムからデータを収集する場合は、プロセスごとに 1 つの dbx インスタンスを接続し、それらのプロセスごとにコレクタを有効にする必要があります。MPI ジョブのプロセスに dbx を接続すると、各プロセスが停止され別々の時間に再起動されます。この時間差によって MPI プロセス間のインタラクションに変化が生じ、収集するパフォーマンスデータに影響を及ぼす可能性があります。この問題の影響を抑える 1 つの方法は、pstop(1) を使用してすべてのプロセスを停止することです。ただし、すべてのプロセスを dbx に接続した場合は、dbx からそれらのプロセスを再開する必要があり、そのときに時間的な遅延が発生して、MPI プロセスの同期に影響が出ることがあります。59 ページの「MPI プログラムからのデータの収集」も参照してください。

MPI プログラムからのデータの収集

コレクタは、Sun MPI (Message Passing Interface) ライブラリ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。MPI ライブラリは、Sun HPC ClusterTools™ ソフトウェアに付属しています。可能であれば、最新である 4.0 バージョンの ClusterTools™ ソフトウェアを使用してください。可能でなければ、3.1 バージョンまたは互換バージョンを使用してもかまいません。並列ジョブを起動するには、Sun CRE (Cluster Runtime Environment) コマンドの mprun を使用します。詳細については、Sun HPC ClusterTools™ のマニュアルを参照してください。また、MPI や 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。

MPI とコレクタの実装方法により、1 つの MPI プロセスに 1 つの実験ファイルが作成されます。これらの実験は、それぞれ一意の名前を持つ必要があります。実験ファイルの格納場所と格納方法は、MPI ジョブから利用可能なファイルシステムの種類に依存します。実験ファイルの格納については、次の節を参照してください。

MPI ジョブからデータを収集する方法としては、MPI の下で collect コマンドを実行する方法と、MPI の下で dbx を起動し、dbx の collector サブコマンドを使用する方法があります。以下では、これらのオプションのそれぞれについて説明します。

MPI 実験ファイルの格納

マルチプロセス環境は複雑になることがあり、MPI プログラムから収集されたパフォーマンスデータを記録する MPI 実験ファイルの格納にあたっては、注意すべきいくつかの問題があります。これらは、データ収集と記憶領域の効率性、実験の命名に関係している問題です。MPI 実験をはじめとする実験の実験名については、37 ページの「収集データの格納場所」を参照してください。

パフォーマンスデータを収集する MPI プロセスは、それぞれ専用の実験ファイルを作成します。実験を作成するとき、MPI プロセスは実験ディレクトリをロックします。このため、他の MPI プロセスがそのディレクトリを使用するには、ロックが解除されるのを待つ必要があります。つまり、あらゆる MPI プロセスからアクセス可能なファイルシステムに実験を格納した場合、実験ファイルは順次に作成されますが、各 MPI プロセスにローカルのファイルシステムに格納した場合は、すべての実験ファイルが同時に作成されます。

実験名の標準形式である *experiment*`.`*n*`.er` を使用し、共通ファイルシステムの 1 つに実験ファイルを格納した場合、それらの実験ファイルには一意の名前が割り当てられます。この場合の *n* の値は、MPI プロセスが実験ディレクトリに対するロックを取得した順序によって決まり、必ずしもプロセスの MPI ランクに対応しません。動作中の MPI ジョブ内の MPI プロセスに `dbx` を接続した場合、*n* は接続した順序によって決まります。

実験名の標準形式を使用してローカルファイルシステムに実験ファイルを格納した場合、それらの実験ファイルの名前は一意ではありません。たとえば `node0`、`node1`、`node2`、`node3` の 4 つのシングルプロセッサノードを持つマシン上で MPI ジョブを実行したとします。各ノードには `/scratch` という名前のローカルディスクがあり、このディスク上のディレクトリ *username* に実験を格納します。MPI ジョブによって作成された実験は、次のフルパス名を持ちます。

```
node0:/scratch/username/test.1.er
node1:/scratch/username/test.1.er
node2:/scratch/username/test.1.er
node3:/scratch/username/test.1.er
```

ノード名を含むフルパス名は一意ですが、実験ディレクトリ内の実験名はすべて `test.1.er` です。MPI ジョブの完了後に実験ファイルを共通の場所に移動する場合は、名前が重複しないようにする必要があります。たとえば、自分のホームディレクトリ (すべてのノードに共通と仮定) に実験を移動して、実験名を変更するには、以下のコマンドを使用します。

```
rsh node0 'er_mv /scratch/username/test.1.er test.0.er'
rsh node1 'er_mv /scratch/username/test.1.er test.1.er'
rsh node2 'er_mv /scratch/username/test.1.er test.2.er'
rsh node3 'er_mv /scratch/username/test.1.er test.3.er'
```

大規模な MPI ジョブの場合は、スクリプトを使用して共通の場所に実験ファイルを移動することもできます。ただし、その場合は、UNIX® コマンドの `cp` や `mv` を使用しないでください。実験ファイルのコピーと移動の方法については、173 ページの「実験の操作」を参照してください。

実験名を指定しなかった場合、標本コレクタは MPI ランクに基づき、標準形式の *experiment*.*n*`.er` で実験名を作成します。この場合の *n* は MPI ランクです。また、*experiment* は、実験グループが指定された場合の実験グループ名で、それ以外の場合は `test` になります。実験名は、共通またはローカルのファイルシステムのどちらが使用されるかに関係なく一意です。つまり、ローカルファイルシステムを使用して実験ファイルを記録し、それらのファイルを共通のファイルシステムにコピーする場合、実験名を変更する必要はありません。

利用できるローカルファイルシステムがわからない場合は、`df -lk` コマンドを使用するか、システム管理者に確認してください。実験ファイルは、必ず、一意に定義され、他の実験に使用されていない既存のディレクトリに格納してください。また、ファイルシステムに、実験ファイルを格納するための十分な空き領域があることを確認してください。必要なディスク容量の概算方法については、39 ページの「必要なディスク容量の概算」を参照してください。

注 – コンピュータ間やノード間で実験ファイルだけをコピーまたは移動すると、注釈付きのソースコードや注釈付きの逆アセンブリコード内のソース行を表示できなくなります。これらのコードを表示するには、実験に使用されたロードオブジェクトとソースファイル (または同じパスとタイムスタンプを持つコピー) にアクセスする必要があります。

MPI の制御下での `collect` コマンドの実行

MPI の制御下で `collect` コマンド を使用してデータを収集するには、次の構文を使用します。

```
% mprun -np n collect [collect-arguments] program-name [program-arguments]
```

ここで、*n* は MPI で作成されるプロセス数です。この手順では、`collect` の *n* 個の個別のインスタンスを作成しますが、それぞれのインスタンスは実験を記録します。実験を保存する場所と方法についての詳細は、37 ページの「収集データの格納場所」の節をお読みください。

さまざまな MPI ランから集めた実験結果が別々に保存されるようにするために、MPI ランごとに `-g` オプションで実験グループを作成することができます。実験グループはすべての MPI プロセスからアクセスできるファイルシステムに保存してください。実験グループを作成すると、1 つの MPI ランに関する実験データをパ

フォーマンスアナライザに容易に読み込むこともできます。グループを作成する方法として、-d オプションで各 MPI ランに個別のディレクトリを指定することもできます。

MPI の制御下で dbx を起動することによるデータ収集

MPI の制御下で dbx を起動し、データを収集するには、次の構文を使用します。

```
% mprun -np n dbx program-name < collection-script
```

ここで、*n* は MPI で作成されるプロセス数であり、*collection-script* はデータ収集をセットアップして起動するのに必要なコマンドを含む dbx スクリプトです。この手順では、dbx の *n* 個の個別のインスタンスを作成しますが、それぞれのインスタンスは MPI プロセスのうちの 1 つに実験を記録します。実験名を定義しないと、実験には MPI ランクのラベルが付けられます。実験を保存する場所と方法についての詳細は、60 ページの「MPI 実験ファイルの格納」の節をお読みください。

収集スクリプトとプログラム内の MPI_Comm_rank() への呼び出しを使用して、MPI ランクで実験に名前を付けることができます。たとえば、C プログラムには次の行を挿入します。

```
ier = MPI_Comm_rank(MPI_COMM_WORLD,&me);
```

Fortran プログラムには次の行を挿入します。

```
call MPI_Comm_rank(MPI_COMM_WORLD, me, ier)
```

この呼び出しを行 17 に挿入してあれば、次のようにスクリプトを使用することができます。

```
stop at 18
run program-arguments
rank=$[me]
collector enable
collector store filename experiment.$rank.er
cont
quit
```

ppgsz での collect の使用

ppgsz(1) で collect を使用することができます。このためには、ppgsz コマンド上で -F on または -F all フラグを指定して collect を実行します。親の実験は ppgsz 実行可能ファイル上にあり、無関係です。パスに 32 ビット版の ppgsz があり、64 ビットプロセス対応のシステムで実験を実行すると、ppgsz によって最初にその 64 ビット版が exec されて、_x1.er が作成されます。この実行可能ファイルはフォークし、_x1_f1.er が作成されます。

子プロセスは、exec が成功するまで、パス上の最初のディレクトリ、次に 2 つ目のディレクトリというように、指定された名前のターゲットの exec を試みます。たとえば 3 回目の試みが成功した場合、最初の 2 つの派生の実験にはそれぞれ _x1_f1_x1.er、_x1_f1_x2.er という名前が付けられ、両者は完全に空です。ターゲット上の実験は成功した exec (この例では 3 番目) から得られたものであり、_x1_f1_x3.er という名前で、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 を呼び出すことによって直接処理することができます。

上記の例と同じパスで、初期プロセスが 64 ビットの ppgsz であるか、32 ビットカーネル上で 32 ビットの ppgsz を呼び出した場合、本当のターゲットを exec するフォークの子のデータは _f1.er に格納され、本当のターゲットの実験は _f1_x3.er に格納されます。

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、collect コマンドか IDE、または dbx 上でコレクタコマンドを使用したときに、コレクタによって収集した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。第 3 章で説明されているように、プロセスの実行中、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、表やグラフの形式にデータを表示します。アナライザは er_print としてコマンド行からも利用できます。これについては第 5 章を参照してください。

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、コマンド行に以下を入力します。

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

または、IDE の「エクスプローラ」を使って、実験に移動し、これを開きます。experiment-list コマンド引数には、実験名、実験グループ名、またはその両方を空白で区切って指定します。

コマンド行には、複数の実験や実験グループを指定できます。その中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が、派生実験のデータは表示されません。個々の派生実験を読み込むには、それぞれの実験を明示的に指定するか、実験グループを作成する必要があります。実験グループを作成するには、最初の行が以下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
```

この後の行に実験の名前を追加します。ファイル拡張子は、.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。

「実験を開く」ダイアログまたは「実験を追加」ダイアログ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、以下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]
```

アナライザが初期化を行い、「収集」ダイアログに、指定のターゲットとその引数、実験を収集するための設定が表示されます。詳細は 79 ページの「実験の記録」を参照してください。

アナライザオプション

これらのオプションはアナライザの動作を制御し、以下の 3 つのオプショングループに分類されます。

- 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

-j | --jdkhome *jvm-path*

アナライザを実行する Java™ 仮想マシン (JVM™) へのパスを指定します。デフォルトのパスは `/usr/j2se` です。(「Java 仮想マシン (JVM™)」という用語は、Java™ プラットフォーム用仮想マシンを意味します。)

-J *jvm-options*

JVM™ オプションを指定します。

制御オプション

-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。以下のそれぞれのオプションはスタンドアロンのオプションで、他の `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

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。

-?| --h| --help

使用情報を出力して終了します。

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」メニューが入っています。

「ファイル」メニューでは、実験と実験グループを開いたり、追加、および解除を行うことができます。「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ための方法については、79 ページの「実験の記録」を参照してください。
「ファイル」メニューではマップファイルも作成できます。マップファイルを使って、実行可能ファイルのサイズを最適化したり、その実効キャッシュ動作を最適化できます。マップファイルの詳細は、79 ページの「マップファイルの生成と関数の順序の変更」を参照してください。

「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、以下の「アナライザデータ表示」の節で説明されているように、タイムライン表示の操作を支援します。

「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、クイックリファレンスとショートカットの説明、および障害追跡の説明が含まれます。

ツールバー

ツールバーは、メニューショートカットの一連のアイコンを提供します。これには、以下に説明するデータ表示を使用する際に関数を見つけるための検索機能が含まれています。検索機能の詳細については、77 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。それぞれの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含まれます。これらを以下に示します。

- 「関数」タブ
- 「呼び出し元 - 呼び出し先」タブ
- 「ソース」タブ
- 「行」タブ
- 「逆アセンブリ」タブ
- 「PC」タブ
- 「タイムライン」タブ
- 「リーク一覧」タブ
- 「統計」タブ
- 「実験」タブ

デフォルトでは、「関数」タブが表示されます。ターゲットを指定しないでアナライザを起動すると、実験を開くように指示されます。実験を開くと、「実験」タブが表示されます。

読み取り中の実験にデータ空間プロファイリングのデータが記録されている場合は、前述のタブの他に「データレイアウト」タブと「データオブジェクト」タブも表示されます。

「関数」タブ

「関数」タブには、関数およびそのメトリックのリストが表示されます。メトリックは、実験で収集されたデータから得られます。メトリックは、排他的または包括的になります。排他的メトリックは、関数自体の中での使用を表します。包括的メトリックは、関数とその関数が呼び出したすべての関数内での使用を表します。

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、メトリックがゼロ以外である関数だけです。

時間メトリックは秒数として表示され、ミリ秒の精度で示されます。百分率は 0.1% の精度で示されます。メトリック値が正確にゼロの場合は、その時間と百分率は「0」と示されます。ゼロではないのだが、精度より小さい場合、値は「0.000」、百分率では「0.0」と示されます。カウントメトリック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は、収集されたデータ、および各種 .rc ファイルから読み取られる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、以下のようになります。

- 時間ベースのプロファイルでは、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ルでは、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 (他のカウンタの場合) 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。

2 種類以上のデータを収集した場合は、各種類のデフォルトメトリックが示されます。

表示されるメトリックは、変更または再編成できます。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、ツールバーの「検索」ツールを使用します。「検索」ツールの詳細は、77 ページの「テキストとデータの検索」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」タブの中央区画には選択された関数が表示され、上の区画にはその関数の呼び出し元が、下の区画には呼び出し先が表示されます。

このタブには、各関数の排他的メトリック値と包括的メトリック値のほか、属性メトリックも表示されます。選択された関数の属性メトリックは、その関数の排他的メトリックを表します。関数の呼び出し先の属性メトリックは、呼び出し先の包括的メトリックのうちの、中央の関数からの呼び出しに帰させることが可能な部分を示します。呼び出し先の属性メトリックと選択された関数の合計が、選択された関数の包括的メトリックになります。

関数の呼び出し元の属性メトリックは、選択された関数の包括的メトリックのうちの、呼び出し元からの呼び出しに帰させることが可能な部分を示します。すべての呼び出し元の属性メトリックの合計が加算されて、選択された関数の包括的メトリックになります。

「呼び出し元 - 呼び出し先」タブに表示されるメトリックは、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。

呼び出し元または呼び出し先の区画の関数を 1 回クリックすると、その関数が選択され、ウィンドウの内容が再描画されて、選択された関数が中央区画に表示されます。

「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、各ソース行に対応するパフォーマンスメトリックの注釈付きで表示されます。ソースファイル、対応するオブジェクトファイル、およびロードオブジェクトの完全な名前が、ソースコードの見出しに示されます。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絶対パス名のもとで、選択された関数を含むファイルを探します。そこにない場合は、現在の作業ディレクトリで、同じベース名を持つファイルを探します。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、カレントディレクトリからそのソースの実際の場所を指すシンボリックリンクを設定できます。

ソースコードは、表示するように選択されたコンパイラのコメントとともに交互表示されます。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、デフォルト値ファイルで設定することができます。

「ソース」タブに表示されるメトリックは、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。

ソースファイル内の任意の行のメトリックがその最大値のしきい百分率と等しいかそれを超える行は、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、デフォルト値ファイルで設定することができます。ソースファイル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照合マークが表示されます。たとえば、ソースファイルの末尾近くにしきい値を超えている行が 2 行ある場合は、ソースウィンドウの下部近くのスクロールバーの横に 2 つの照合マークが表示されます。照合マークの横にスクロールバーを移動すると、それに対応するしきい値を超えた行が表示されるように、ソースウィンドウに表示されるソース行が位置付けられます。

「行」タブ

「行」タブには、ソース行およびそのメトリックのリストが表示されます。ソース行には、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が示されます。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 PC が 1 つのエントリとしてまとめられて表示されます。関数が非表示の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」表示に 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、選択した関数が入っ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ます。各命令のパフォーマンスメトリック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は、逆アセンブリリストには、そのソースコード、表示するように選択されたコンパイラのコメントが交互表示されます。「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。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、デフォルト値ファイルで設定することができます。

メトリックがその固有のしきい値と等しいかそれを超える行は、アナライザによって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、デフォルト値ファイルで設定することができます。ソースウィンドウの場合と同様に、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、スクロールバーの隣りに照合マークが表示されます。

「PC」タブ

「PC」タブには、PC およびそのメトリックのリストが表示されます。PC には、そのラベルとして、呼び出し元の関数、およびその関数内のオフセットが示されます。関数名が公開されていないロードオブジェクトから呼び出された関数の PC は、ロードオブジェクトの 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、その行の全メトリックが「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。

「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のリストが表示されます。実験にデータ空間データが記録されている場合は、デフォルトでこのタブが表示されます。また、このタブは、「データ表示方法を設定」ダイアログの「書式」タブで「データ領域表示」を「有効化」にするか、アナライザ起動時に読み取られる `er.rc` ファイルの 1 つで「datamode on」コマンドを設定することによっても表示できます。このタブは、積極的なバックトラッキングオプションを有効にしたハードウェアカウンタの実験と、C コンパイラで `-xhwcprof` オプションを使ってコンパイルされたソースファイルにのみ適用できます。

有効にすると、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が示されます。

「データレイアウト」タブ

「データレイアウト」タブは、データ誘導メトリックデータを持つ、プログラムのすべての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を表示します。レイアウトは、実験のロードオブジェクトに定義されている順序に表示されます。このタブには、集合体データオブジェクトごとに、そのオブジェクトの属性となる合計メトリックが表示され、その後に、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、そのメトリックと、32 バイトブロック単位のそのサイズと位置を示す情報が表示されます。

「データオブジェクト」タブの場合と同様に、実験にデータ空間データが記録されている場合は、デフォルトで「データレイアウト」タブが表示されます。また、このタブは、「データ表示方法を設定」ダイアログの「書式」タブで「データ領域表示」を「有効化」にするか、アナライザ起動時に読み取られる `er.rc` ファイルの 1 つで「datamode on」コマンドを設定することによっても表示できます。

「タイムライン」タブ

「タイムライン」タブには、イベントのグラフとコレクタによって記録された標本ポイントが、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。それぞれの LWP について、記録される各データ型 (時間ベースのプロファイル、ハードウェアカウンタプロファイル、同期トレース、ヒープトレース、および MPI トレース) ごとに 1 つのバーが表示されます。

標本データを含むバーは、各標本の個々の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シングデータのバーには、記録される各イベントのイベントマーカが表示されます。イベントマーカは、イベントで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。選択物は「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。

ある種のデータのイベントは、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ある場合は、常に 1 つだけが描画されます。1 つか 2 つのピクセル内に複数のイベントがある場合は、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灰色の小さな照合マークが表示されて、その重なりが示されます。

選択された関数にマップされる色と同様に、「タイムライン」タブに表示されるイベント固有のデータの種類は変更できます。「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、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。

「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。合計値の後には、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、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を示すツリーが表示されます。情報には、実験や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が含まれます。下のパネルには、アナライザセッションで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。

データ表示、右の区画

右の区画には、「概要」タブ、「イベント」タブ、および「凡例」タブが表示されます。デフォルトでは、「概要」タブが表示されます。「タイムライン」タブが選択されている場合を除き、他の 2 つのタブは選択不可になります。

「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 (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、スレッド、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に関する詳細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て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。

このタブを利用できるのは、「タイムライン」タブが選択されたときだけです。

「凡例」タブ

「凡例」タブには、「タイムライン」タブの関数とマイクロステートに使用する色のマップに関する凡例が表示されます。このタブを利用できるのは、左区画で「タイムライン」タブが選択されたときだけです。項目にマップされた色は、凡例で項目を選択して「タイムライン」メニューから「関数のカラーチューザ」を選択するか、色ボックスをダブルクリックすることで、変更することができます。

「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。

データ表示オプションの設定

データの表示方法は、「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、ツールバーで「データ表示方法の設定」ボタンをクリックするか、「表示」メニューから「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 6 つのタブを持つ区画が含まれています。

- メトリック
- ソート
- ソース/逆アセンブリ
- 書式
- タイムライン
- 検索

「メトリック」タブには、使用できるすべてのメトリックが表示されます。各メトリックの 1 つ以上の欄には、メトリックの種類に応じて「*Time*」、「*Value*」、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。

「ソート」タブには、メトリックが表示される順序と、ソートに選択できるメトリックが表示されます。

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

- 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交互表示
- 逆アセンブリリストにおけるソース行のメトリック
- 逆アセンブリリストにおける 16 進での命令の表示

「書式」タブには、C++ 関数名と Java メソッド名について、長短どちらの形式の関数名を使用するかが表示されます。また、このタブには、選択できる Java モード (「オン」、「上級」、または「オフ」) と、データ領域表示オプション (「有効」または「無効」) も示されます。

「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。

「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示し、他のすべての名前はファイルシステムのパスを示します。

「データ表示方法の設定」ダイアログボックスの「保存」ボタンを使用して、現在の設定を格納できます。

テキストとデータの検索

アナライザのツールバーには、コンボボックスで指定した検索ターゲットのための 2 つのオプションを備えた「検索」ツールが装備されています。「関数」タブや「呼び出し元 - 呼び出し先」タブの「名前」欄のテキストや、「ソース」タブや「逆アセンブリ」タブの「コード」欄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」タブには、各ロードオブジェクトの全関数が表示されます。「関数の表示/非表示」ダイアログボックスを使って、ロードオブジェクトの全関数を非表示に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の関数を非表示にすると、「関数」タブと「呼び出し元 - 呼び出し先」タブに、ロードオブジェクトの全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全関数の全 PC を集合体化した 1 つのエントリが表示されます。

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、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。

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は、実験によってフィルタリングできます。実験は個々に読み込むことも、実験グループを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て指定するか、または 15 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て指定するか、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て指定するか、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 は同じものを意味します。

CPU の選択

CPU 情報が記録されている場合は (Solaris™ 9)、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て指定するか、または範囲を指定します。

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、「収集」ダイアログボックスが表示されます。これによって、指定したターゲットの実験を記録することができます。引数なしで、または *experiment-list* を指定してアナライザを起動した時には、「収集」ダイアログボックスを開くことによって、新しい実験が記録します。

「収集」ダイアログボックスの上のパネルでは、実験の実行に使用するターゲット、引数、および各種パラメータを指定します。これらは、第 3 章で説明されている `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。このボタンを押すと、「実行」ボタンを押したときに使用される `collect` コマンドがテキストフィールドに表示されます。

ここには、コレクタ自体の出力を取得するパネルと、プロセスの出力を取得するパネルの 2 つがあります。

一連のボタンを使って、以下の操作を実行できます。

- 実行
- 実行の「一時停止」、「再開」、および「標本」シグナルのプロセスへの送信 (対応するシグナルが指定されている場合に有効)
- 実行
- パネルを閉じる

実験の実行中にパネルを閉じても、実験は続行されます。パネルを再度起動すると、実験の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。

マップファイルの生成と関数の順序の変更

アナライザでは、データの解析の他、関数の順序を変更できます。アナライザでは、実験のデータを使用してマップファイルを生成できます。このマップファイルを静的リンカー (ld) で利用してアプリケーションを再リンクすることによって、作成する実行可能ファイルのワーキングセットサイズの縮小や I キャッシュ動作の効率化を図ることができます。

マップファイルに記録される関数の順序と、実行可能ファイルにおける関数の順序の変更に使用される関数の順序は、関数リストのソートに使用されるメトリックで決まります。マップファイルの生成には、通常、排他的ユーザー CPU 時間または排他的 CPU サイクル時間が使用されます。同期遅延やヒープトレースなどのメトリック、または名前やアドレスを使っても、マップファイルに意味ある順序付けを行うことはできません。

デフォルト

アナライザは、カレントディレクトリに `.er.rc` ファイルがある場合はその指令を、ユーザーのホームディレクトリに `.er.rc` ファイルがある場合はその指令を、そしてシステム全体の `.er.rc` ファイルの指令を処理します。これらのファイルには、メトリック、ソート、そしてコンパイラのコメントオプションを指定してソースおよび逆アセンブリ出力のしきい値を強調表示するためのデフォルト設定が含まれている場合があります。また、「タイムライン」タブ、名前形式、Java™ モードの設定 (`javamode`)、および「データ領域表示モード」(`datamode`) のデフォルト設定も指定されています。さらに、ソースおよびオブジェクトファイルの検索パスを制御する指令も含まれている場合があります。

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限り、標準出力に出力されます。er_print 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 ユーティリティを使用して、関数のパフォーマンスメトリックや呼び出し元と呼び出し先、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、実行統計情報を表示することができます。

この章では、以下について説明します。

- er_print の構文
- メトリックリスト
- 関数リストを管理するコマンド
- 呼び出し元 - 呼び出し先リストを管理するコマンド
- リークリストと割り当てリストを管理するコマンド
- 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
- データ領域リストを管理するコマンド
- 実験、標本、スレッド、および LWP を一覧するコマンド
- 選択を管理するコマンド
- ロードオブジェクトの選択を管理するコマンド
- メトリックを一覧するコマンド
- 出力を制御するコマンド
- 他の表示を出力するコマンド
- デフォルト値を設定するコマンド
- パフォーマンスアナライザに対するデフォルト値を設定するコマンド
- その他のコマンド
- 例

コレクタが収集するデータについては、第 2 章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に表示する方法については、第 4 章とオンラインヘルプを参照してください。

er_print の構文

er_print のコマンド行構文は以下のとおりです。

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

表 5-1 に、er_print のオプションをまとめます。

表 5-1 er_print コマンドのオプション

オプション	内容の説明
-	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* というファイルからコマンドを読み取ります。*script* ファイルは er_print コマンドからなるリストで、1 行に 1 つの割合で er_print コマンドを指定します。-script オプションを指定しなかった場合、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、デフォルトでは、er_print は対話モードになり、キーボードからコマンドを入力することができます。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。

er_print に指定可能なコマンドについては、以降の節で説明します。すべてのコマンドは、他のコマンドと重複しない限り、短縮することができます。

メトリックリスト

er_print コマンドの多くは、メトリックキーワードのリストを使用します。このリストの構文は以下のとおりです。

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

size、address、name キーワードを除き、メトリックキーワードは、メトリックタイプ文字列 (type)、メトリック表示形式文字列 (visibility)、およびメトリック名文字列 (name) 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。

```
<type><visibility><name>
```

メトリックタイプとメトリック表示形式文字列は、タイプ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タイプ文字を表 5-2 にまとめます。複数のタイプ文字からなるメトリックキーワードは展開され、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 5-2 メトリックタイプ文字

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み)

指定可能なメトリック表示形式文字を表 5-3 にまとめます。表示形式文字列を構成する文字の順序は重要ではありません。対応するメトリックの表示順序が、この指定順序の影響を受けることはありません。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他のキーワードで区切られている場合は、標準の順序で 2 つのメトリックの 1 つ目の位置にメトリックが表示されます。

表 5-3 メトリック表示形式文字

文字	内容の説明
.	時間形式でメトリックを表示します。この指定は、タイミングメトリックと循環型のハードウェアカウンタメトリックに有効です。これ以外のメトリックに指定された場合は、+ と解釈されます。

表 5-3　メトリック表示形式文字 (続き)

文字	内容の説明
%	プログラム全体のメトリックに占める割合 (百分率) でメトリックを表示します。呼び出し元 - 呼び出し先リストの属性メトリックの場合は、選択した関数の包括的メトリックに占める割合が表示されます。
+	絶対値の形式でメトリックを表示します。ハードウェアカウンタの場合、この値はイベント発生回数です。タイミングメトリックに指定された場合は、"." と解釈されます。
!	メトリック値を表示しません。他の表示形式文字と組み合わせることはできません。

タイプと表示形式文字列それぞれが複数の文字から構成されている場合は、タイプ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。

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。

表 5-4 に、タイミングメトリック、同期遅延メトリック、メモリー割り当てメトリック、MPI トレースメトリック、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に指定可能な `er_print` メトリック名文字列をまとめます。他のハードウェアカウンタメトリックの場合、メトリック名文字列はカウンタ名と同じです。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、8 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 5-4 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	user	ユーザー CPU 時間
	wall	時計時間
	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	他の待ち時間
同期遅延メトリック	sync	同期待ち時間
	syncn	同期待ち回数
MPI トレースメトリック	mpitime	MPI 呼び出しに費やされた時間
	mpisend	MPI 送信関数の数
	mpibytessent	MPI 送信関数で送信したバイト数
	mpireceive	MPI 受信関数の数
	mpibytesrecv	MPI 受信関数で受信したバイト数
	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	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のオーバーフローメトリック	cycles	CPU サイクル
	insts	発行された命令

表 5-4 に示した名前文字列のほかに、デフォルトメトリックリストでのみ使用できる名前文字列が 2 つあります。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® と Intel に共通のものですが、アーキテクチャ固有の他のタイプのものも存在することに注意してください。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 コマンドを使用します。

関数リストを管理するコマンド

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とともに関数リストを出力します。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関数のすべて、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。

出力する行数は、limit コマンドを使用して制限できます (100 ページの「出力を制御するコマンド」を参照)。

デフォルトでは、秒数およびプログラム全体のメトリックに占める割合 (百分率) の形式で排他的および包括的ユーザー CPU 時間が出力されます。表示するメトリックは、metrics コマンドを使用し変更できます。この操作は、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、Java™ モードが on、expert、off のどれに設定されているかによって異なります。

- Java™ モードを on に設定すると、表示された関数情報に、Java™ メソッドと呼び出されたネイティブメソッドに対するメトリックが含められます。(ネイティブメソッドは、Java™ ターゲットによって呼び出される C/C++ または Fortran 関数です。) Java™ モードでは、解釈され HotSpot™ でコンパイルされたメソッドのデータが集合体として表示され、非ユーザー Java スレッドのデータは表示されません。
- Java™ モードを expert に設定すると、Java™ モードで表示されない JVM 内部要素の詳細が表示されます。タイムラインには、Java™ ユーザースレッドだけでなく、すべてのスレッドが表示されます。その他の点では、表示される関数情報は Java™ モードが on に設定されている場合と同じになります。
- Java™ モードを off に設定すると、HotSpot™ でコンパイルされたメソッドやネイティブメソッドとともに、JVM 自体でインタプリタされる Java™ アプリケーションにある関数でなく JVM 自体にある関数が表示されます。すべてのスレッドが表示されます。

metrics *metric_spec*

関数リストに表示するメトリックを指定します。*metric_spec* には、キーワードの default (一群のデフォルトのメトリックが復元される) 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。下記は、メトリックリストの指定例です。

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

このコマンドが入力すると、er_print は以下を表示します。

- 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)

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。上記の例では、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、82 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、前回の設定が引き続き有効になります。

sort *metric_spec*

指定したメトリックを基準に関数リストの内容をソートします。文字列 *metric-spec* は、82 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、包括的ユーザー CPU 時間を基準に関数リストの内容をソートします。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て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

関数リスト内のすべての関数について、それぞれに概要メトリック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 (100 ページの「出力を制御するコマンド」を参照)。

概要メトリックパネルには、関数や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と包括的メトリック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メトリックパネルを書き込みます。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、概要メトリックパネルが書き込まれます。コマンド行でコマンドを使用する場合には *N* が必要です。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使用すると、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

すべての関数のそれぞれについて、内容をソートした順序で呼び出し元 - 呼び出し先パネルを表示します。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 (100 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、以下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。

cmetrics *metric_spec*

呼び出し元 - 呼び出し先に関するメトリックを指定します。*metric_spec* は、次の例のようにコロンで区切ったメトリックキーワードのリストです。

```
% cmetrics i.user:i%user:a.user:a%user
```

このコマンドを入力すると、`er_print` は以下を表示します。

- 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)
- 属性ユーザー CPU 時間 (秒単位)
- 属性ユーザー CPU 時間 (百分率)

cmetrics コマンドが終了すると、現在有効な一群のメトリックを示すメッセージが表示されます。上記の例では、メッセージは次のようになります。

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

メトリックリストの構文については、82 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の一覧を表示するには、cmetric_list コマンドを使用します。

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書き込みます。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書き込まれます。コマンド行でコマンドを使用する場合には *N* が必要です。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使用すると、対応する *N* 値を持つ関数のリストが出力されます。

csort *metric_spec*

指定したメトリックを基準に呼び出し元 - 呼び出し先の内容をソートします。文字列 *metric-spec* は、82 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。

```
% csort a.user
```

このコマンドが入力されると、er_print は属性ユーザー CPU 時間を基準に呼び出し元 - 呼び出し先の内容をソートします。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。

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る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る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 (PC) と、そのメトリックのリストを書き込みます。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集計されたメトリックを示す行が含まれています。

psummary

PC リストに各 PC の概要メトリックパネルを現在のソートメトリックで指定された順序で書き込みます。

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のリストを書き込みます。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集計されたメトリックを示す行と、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集計されたメトリックを示す行が含まれています。

lsummary

行リストに各行の概要メトリックパネルを現在のソートメトリックで指定された順序で書き込みます。

source { *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。

オプションのパラメータの *N* (正の整数) は、ファイルまたは関数名が一意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 (*N*) のないあいまいな名前が指定された場合、er_print はオブジェクトファイル名の候補を表示します。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付けられ、そのオブジェクトファイルの *N* の値を表す番号も表示されます。

関数名は *function'file'* としても指定できます。この場合、file を使って、関数の代替ソースコンテキストを指定します。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属するソースファイルと定義されます。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になります。代替ソースコンテキストは、関数に属する命令を含む他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。

disasm { *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。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。

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 5-5 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]	フロントエンドメッセージを示します。
all	すべてのメッセージを示します。
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 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の強調表示に使用するしきい値百分率を指定します。任意のメトリック値が、ファイル内のソース行の該当メトリック値の最大値の *value* % と同じかそれ以上である場合、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*

注釈付きソースコード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、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 5-6 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]	ソースリストを注釈付き逆アセンブリリストにインタリーブします。
nos[rc]	ソースリストを注釈付き逆アセンブリリストにインタリーブしません。
as[rc]	注釈付きソースコードを注釈付き逆アセンブリリストに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の強調表示に使用するしきい値の百分率を指定します。任意のメトリック値が、ファイル内の命令行の該当メトリック値の最大値の *value*% と同じかそれ以上である場合、メトリックが発生する行の先頭に ## が挿入されます。

setpath *path_list*

ソース、オブジェクトなどのファイルを検索するためのパスを設定します。*path_list* は、ディレクトリのコロンで区切られたリストです。ディレクトリ内にコロンがある場合は、バックスラッシュでコロンをエスケープする必要があります。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。

デフォルトの設定は次のとおりです。`$expts:.` 現在のパスの設定の検索でファイルが見つからない場合は、内部でコンパイルされた完全名が使用されます。

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。

データ領域リストを管理するコマンド

`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とともに出力します。積極的なバックトラッキングを指定した HW カウンタの実験と、`-xhwcprof` でコンパイルされたファイル内のオブジェクトにのみ適用できます (C のみ SPARC® で適用できます)。詳細は、『C ユーザーズガイド』または `cc`(1) のマニュアルページを参照してください。

`data_osingle name [`*N*`]`

指定されたデータオブジェクトの概要メトリック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。積極的なバックトラッキングを指定した HW カウンタの実験と、`-xhwcprof` でコンパイルされたファイル内のオブジェクトにのみ適用できます (C のみ SPARC® で適用できます)。詳細は、『C ユーザーズガイド』または `cc`(1) のマニュアルページを参照してください。

`data_olayout`

データ誘導メトリックデータを持つすべてのプログラムデータオブジェクトについて、それらが実験のロードオブジェクトで定義されている順序で、注釈付きのデータオブジェクトレイアウトを作成します。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が表示され、その後に、そのオブジェクトのすべての要素が表示されます。各要素には、そのメトリックと、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。

実験、標本、スレッド、および LWP を一覧するコマンド

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

読み込まれているすべての実験をその ID 番号とともに一覧表示します。各実験は索引付きで表示されますが、この索引は標本、スレッド、LWP を選択するときに使用します。

以下は、実験リストの表示例です。

```
(er_print) experiment_list
ID 実験ファイル
== ==========
1 test.1.er
2 test.6.er
```

sample_list

解析対象として選択されている標本を一覧表示します。

以下は、標本リストの表示例です。

```
(er_print) sample_list
Exp Sel      合計
=== ======= =====
  1 1-6         31
  2 7-10,15     31
```

lwp_list

解析対象として選択されている LWP を一覧表示します。

thread_list

解析対象として選択されているスレッドを一覧表示します。

cpu_list

解析対象として選択されている CPU を一覧表示します。

選択を管理するコマンド

選択リスト

選択リストの構文は、以下の例に示すとおりです。この節では、この構文を使用してコマンドを説明しています。

```
[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 + 符号でつなぐことによって、複数の選択リストを指定することもできます。

実験リストおよび選択リストの構文は同じで、all キーワード、または空白なしのコロンで区切った番号または番号範囲 (*n*-*m*) リストを指定できます。

```
2,4,9-11,23-32,38,40
```

実験番号は、exp_list コマンドを使用して調べることができます。

以下に選択リストの例を示します。

```
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、標本のいずれかです。

選択用のコマンド

LWP、標本、CPU、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、以下のようにして 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選択を管理するコマンド

object_list

ロードオブジェクトのリストを表示します。各ロードオブジェクト名の前には yes または no のいずれかが付きます。yes は、そのオブジェクトの関数が関数リストに現れることを示し、no は、そのオブジェクトの関数が関数リストに現れないことを示します。

以下は、ロードオブジェクトリストの表示例です。

```
(er_print) object_list
Sel Load Object
=== ================================================
yes /tmp/var/synprog/synprog
yes /opt/SUNWspro/lib/libcollector.so
yes /usr/lib/libdl.so.1
yes /usr/lib/libc.so.1
```

object_select *object1,object2,...*

ロードオブジェクト内の関数情報を表示するロードオブジェクトを選択します。*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。選択されていないロードオブジェクトの場合、ロードオブジェクト内の関数に関する情報ではなく、ロードオブジェクト全体に関する情報が表示されます。

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と、関数リスト内のさまざまな種類のメトリックを参照するときに、その他のコマンド (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

呼び出し元 - 呼び出し先リストで現在選択されているメトリックと、呼び出し元 - 呼び出し先リスト内のさまざまな種類のメトリックを参照するときに、その他のコマンド (cmetrics、csort 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。

注 – 属性メトリックは、cmetrics コマンドおよび callers-callees でのみ指定・表示できます。metrics コマンドや functions コマンドで指定・表示することはできません。

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* で指定したファイルを開きます。**ファイル名**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。

limit *n*

出力を レポートの最初の *n* 個のエントリだけに制限します。*n* は、符号なしの正の整数です。

name { long | short }

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) 。

javamode { on | expert | off }

Java™ 実験のモードを on (Java™ モデルを示す)、expert (Java™ モデルと内部 JVM の詳細を示す)、または off (マシンモデルを示す) に設定します。

他の表示を出力するコマンド

header *exp_id*

指定した実験に関する説明情報を表示します。*experiment-ID* は、`exp_id` コマンドを使用して調べることができます。*exp_id* として `all` を指定するか省略した場合は、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、各ヘッダーの後に表示されます。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は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。

objects

パフォーマンス解析を目的にロードオブジェクトを使用した結果として出力されたエラーメッセージや警告メッセージとともに、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 (100 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出力します。*experiment-ID* は、exp_id コマンドを使用して調べることができます。*exp-id* として all を指定するか省略した場合は、読み込まれた実験すべて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出力します。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。Solaris 7™ および 8 のオペレーティングシステムの標準スレッドライブラリは、プロファイル対象でないシステムスレッドを作成します。これらのスレッドはほとんどの時間をスリープ状態で消費し、その時間は「その他の待ち時間」として統計ディスプレイに表示されます。

exp_id は、experiment_list コマンドを使用して調べることができます。*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

次のコマンドを使用して、er_print およびアナライザに対するデフォルト値を設定することができます。これらのコマンドは、デフォルトの設定にしか使用できません。これらは、er_print の入力では使用できません。また、これらのコマンドは、.er.rc というデフォルト値ファイル内で使用することができます。コマンドの中には、パフォーマンスアナライザにしか適用できないものがあります。

デフォルト値ファイルは、ユーザーのホームディレクトリに置くことも、それ以外のディレクトリに置くこともできます。ホームディレクトリに置かれたデフォルト値ファイル内の設定は、すべての実験に対して適用され、それ以外のディレクトリに置かれたデフォルト値ファイル内の設定は、ローカルに適用されます。er_print、er_src、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デフォルト値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し、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します。

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` 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`、および `dthresh` コマンドを含めることもできます。`er.rc` ファイルには、複数の `dmetrics` および `dsort` コマンドを指定することができ、その場合、それらのコマンドは連結されます。

`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、82 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。実験が読み込まれている場合、ソート基準メトリックは、このリスト内の、その実験に存在するメトリックに最初に一致するメトリックになります。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* のエントリに他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、82 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。

`gdemangle` *library.so*

`gdemangle` コマンドは旧式で、無視されます。現在は、GNU demangler の独自の実装が `er_print` に組み込まれています。

パフォーマンスアナライザに対するデフォルト値を設定するコマンド

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、コロンで区切られたリストです。許容オプションを下表に示します。

表 5-7 タイムライン表示モードオプション

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

オプション lwp、thread、および cpu は、root および leaf と同様に相互排他です。一連の相互排他オプションのいくつかをリストに含める場合、最後のオプションが使用する唯一のオプションです。

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。許容タイプを下表に示します。

表 5-8 タイムライン表示データの種類

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

表 5-8　タイムライン表示データの種類 (続き)

種類	意味
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

datamode { *on* | *off* }

データ領域関連の画面を表示するモードを on (タブが見える) または off (タブが見えない) に設定します。

その他のコマンド

mapfile *load-object* { *mapfilename* | - }

指定したロードオブジェクトのマップファイルを、*mapfilename* で指定したファイルに書き込みます。マップファイル名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。

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 のバージョン情報を表示します。

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。

例

- ここでは、実験から gprof 形式に似た一覧を生成する例を紹介します。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と呼び出し先の一覧です。

  ```
  er_print -outfile er_print.out -metrics e.user:e%user\
  -sort e.user -limit  100 -functions -cmetrics a.user:a%user\
  -csort  a.user -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:e%user -sort e.user \
    -limit 100 -functions test.1.er
    ```
  - ```
    er_print -cmetrics a.user:a%user -csort a.user \
    -callers-callees test.1.er
    ```

- この例は、関数での時間の消費のされ方に関する概要情報を示します。

  ```
  er_print -functions test.*.er
  ```

- この例は、呼び出し元と呼び出し先の関係を示します。

  ```
  er_print -callers-callees test.*.er
  ```

- この例は、着目したいソース行を表示します。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。関数名の後の 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.*.er
  ```

- この例は、コンパイラのコメントのみ表示します。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
  ```

- 次の例では、時計時間プロファイルを使用して、関数および呼び出し元と呼び出し先を一覧表示しています。

  ```
  er_print -metrics ei.%wall -functions test.*.er
  ```

  ```
  er_print -cmetrics aei.%wall -callers-callees test.*.er
  ```

- 次の例は、高度な MPI 関数を表示します。MPI には多数の内部ソフトウェア層がありますが、これはエントリポイントだけを表示する方法です。多少、シンボルの重複が出てくることがありますが、そうした重複は無視してかまいません。

  ```
  er_print -functions test.*.er | grep PMPI_
  ```

第6章

パフォーマンスアナライザとそのデータの内容

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に変換します。メトリックは、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。収集されたあらゆるイベントについて、ヘッダーと次の 2 つの部分からなるデータが記録されます。

- メトリック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の関連付けに使用す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を関連付ける処理は、常に簡単にできるとは限りません。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、以下について説明します。

- データ収集の機能
- パフォーマンスメトリックの意味
- 呼び出しスタックとプログラムの実行
- プログラム構造へのアドレスのマップ
- 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プ

データ収集の機能

データ収集実行からの出力は実験であり、ファイルシステム内の各種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格納されます。

実験の形式

すべての実験には 3 つのファイルが必要です。

- ログファイル。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どのバージョンか、また、ターゲットが生きている間の各種イベントのレコードなどに関する情報が含まれている ASCII ファイル。
- マップファイル。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る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るか読み込み解除される時間を記録する ASCII 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生き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、113 ページの「パフォーマンスメトリックの意味」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。データファイルは、次のような名前が付いています。

表 6-1　データの種類と対応するファイル名

データの種類	ファイル名
時間プロファイル	profile
HWC プロファイル	hwcounters
同期トレース	synctrace
ヒープトレース	heaptrace
MPI-tracing	mpitrace

時間プロファイルまたは HW 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 clock-tick または counter-overflow で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、ヒープトレースまたは MPI 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` により割り込み処理される `libcollector.so` 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後、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、ブロック単位で記入されます。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したり作成できます。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。

アーカイブサブディレクトリ

各実験にはアーカイブサブディレクトリがあり、このサブディレクトリには、ロードオブジェクトファイルで参照された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、データ収集の終わりに実行される er_archive によって作成されます。プロセスが異常終了すると、er_archive 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で呼び出されると、er_print またはアナライザで書き込まれます。

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内のサブディレクトリに書き込みます。これらのサブディレクトリの名前にはアンダースコア、コード文字 (fork を示す f、exec を示す x、そして組み合わせを示す c)、および数字が付けられ、これらは直接の作成者の実験名に追加されて派生の系統を示します。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、動的な関数を記述するロードオブジェクトファイル内の追加レコードと、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™ 実験のロードオブジェクトファイル内には、その内部の目的のために JVM 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もあります。

また、Java™ 実験には、呼び出されたすべてのユーザーの Java™ クラスに関する情報を含む JAVA_CLASSES ファイルがあります。

Java™ ヒープと同期トレースデータは、libcollector.so の一部である JVMPI エージェントを使用して記録されます。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も受信します。

実験の記録

実験を記録する方法として、collect コマンドを使用する方法、プロセスを作成する dbx を使用する方法、実行中プロセスから実験を作成する dbx を使用する方法があります。アナライザデータコレクタ GUI は collect による実験、IDE は dbx による実験を実行します。

collect 実験

collect を使用して実験を記録する場合、collect プログラム自体は実験ディレクトリを作成し、libcollector.so 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 を設定します。このプログラムは次に、実験名に関して libcollector に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、ターゲットをそれ自体の一番上で実行します。

libcollector.so は、すべての実験ファイルの書き込みを行います。

dbx 実験、プロセスの作成

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、実験ディレクトリも作成され、libcollector.so の事前読み込みが保証されます。このコマンドは、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 libcollector 内の初期化ルーチンを呼び出してデータ収集を開始します。

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 JVMDI エージェントを使用し、そのエージェントがデータ収集に必要な JVMPI エージェントと共存できないからです。

dbx 実験、実行中プロセス上

dbx を使用して実行中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 を使用できません。dbx は対話関数呼び出しをターゲット内に行なって libcollector.so を dlopen し、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 実験の場合と同様に libcollector.so で書き込まれます。

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と思われるので、割り込み処理は行えません。さらに、派生プロセス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機能しません。

ユーザーが dbx でプロセスを開始する前か、dbx で実行中プロセスに接続する前に明示的に libcollector.so を LD_PRELOAD した場合は、トレースデータが収集されることがあります。

パフォーマンスメトリックの意味

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。これらの最初の 3 つのデータを使用すれば、時間、スレッド、または LWP によってパフォーマンスアナライザでメトリックのフィルタ処理が行えます。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の精度と、データ収集がメトリック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からなる配列で構成されています。Solaris™ 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 インクリメントされ、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。配列をリセットするには、ユーザー CPU 状態の配列要素を 1 に設定し、他の全状態の配列要素を 0 に設定します。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 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/スレッドが再びユーザーモードにならない限り、呼び出しスタックの内容が変わることはありません。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 6-2 に、各マイクロステートとメトリックの Solaris™ における対応関係をまとめます。

表 6-2 カーネルのマイクロステートとメトリックの対応関係

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	他の待ち時間
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	他の理由によるスリープ	他の待ち時間
LMS_STOPPED	停止 (/proc、ジョブ制御、lwp_stop のいずれか)	他の待ち時間
LMS_WAIT_CPU	CPU 待ち	CPU 待ち時間

タイミングメトリックの精度

タイミングデータは統計データとして収集されます。このため、どのような統計的な標本収集手法であっても、その手法が持つ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。タイミングメトリックでデータが不正確になる、つまり、ひずむ可能性があるのは、たとえば以下のような場合です。

- Solaris™ LWP または Linux スレッドを作成すると、少し時間が経過してから、最初のプロファイルパケットが記録されます。こ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たマイクロステートに帰せられます。多数の LWP/スレッドが作成される場合、誤差はその個数分のプロファイル間隔の大きさ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、少し時間が経過してから、最後のプロファイルパケットが記録されます。多数の LWP/スレッドが破壊される場合、誤差はその個数分のプロファイル間隔の大きさになることがあります。
- LWP/スレッドの再スケジューリングは、プロファイル間隔中に行われます。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 スレッドを実行するプロセッサの個数より実行する Solaris™ LWP または Linux スレッドが多いほど、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に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、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引き起こされる可能性があります。すなわち、マイクロステート状態が記録されたときに、LWP の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、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、このマイクロステート値に、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™ 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。
- スレッドライブラリの重大なセクションにあるときに、プロファイルシグナルが廃棄されることがあり、その場合は、タイミングメトリックが実際より小さくなることがあります。この問題は、Solaris™ のデフォルトスレッドライブラリにのみ発生します。
- システム時間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リング間隔を反映しませんが、システム時間に対して施された調整結果は組み込まれます。時間調整の結果、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、調整は一定のインクリメント単位で行われます。

これの不正確さのほかにも、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に反映されることはありません。これは、相関関係のもう 1 つの例です。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と、その他の方法で得られた時間を比較すると、以下の問題があることに気付きます。

シングルスレッドアプリケーションの場合、通常 1 つのプロセスについて記録された Solaris™ LWP または Linux スレッド合計時間は、同じプロセスについて gethrtime(3C) によって返された値と比較すると、数十分の 1 パーセントの精度になります。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、数パーセントほど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™ 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() が 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ためです。

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 <**合計**> 関数 (プログラム全体を表す) のメトリックを比較することによって、集計された時間メトリック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、<**合計**> の時間メトリック値に含まれないその他の寄与要素が含まれることがあります。これらの原因は、次のとおりです。

- プロファイル対象でないシステムによって作成されるスレッド。Solaris™ 8 のオペレーティングシステムの標準スレッドライブラリは、プロファイル対象でないシステムスレッドを作成します。これらのスレッドはほとんどの時間をスリープ状態で消費し、その時間は「その他の待ち時間」として「統計」タブに表示されます。
- データ収集が一時停止される期間。

同期待ちのトレース

同期待ちトレースは、Solaris™ プラットフォームでのみ利用できます。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、同期遅延イベントの

データを収集します。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。要求時刻と許可時刻の差が待ち時間です。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。

プログラムが結合スレッドを使用している場合は、その遅延の原因となったイベントが完了しない限り、待ちスレッドがスケジューリングされている LWP が他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。

プログラムが非結合スレッドを使用している場合、待ちスレッドがスケジューリングされている LWP は自身に他のスレッドをスケジューリングさせたり、ユーザーの作業を続行したりできます。「ユーザーロック時間」は、いくつかのスレッドが同期イベントを待っている間にすべての LWP がビジー状態であればゼロです。しかし、「同期待ち時間」がゼロでないのは、それが特定のスレッドに関連し、スレッドが動作している LWP に関連しないからです。

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た待ち時間の一部は、イベント記録しきい値を大きくすることによって抑えることができます。

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リングは、Solaris™ プラットフォームでのみ利用できます。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。このことは、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、ずっと頻繁にインクリメントされるサイクルカウンタや命令カウンタに当てはまります。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は、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。詳細については、161 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、122 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では、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に大きな違いがある場合があります。

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ます。
- カウント値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。こうしたカウント値は、コレクタ関数の `collector_record_counters` が原因とされます。この関数のカウント値が大きい場合は、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が原因であることがあります。この場合、プログラムに大量のキャッシュミスがあるように見えますが、データ収集を行わないと、キャッシュミスはごく少なくなることがあります。

この逆に、大きな値を選択すると、オーバーフローの発生が非常に少なくなり、良好な統計情報を得ることができます。最後のオーバーフローの発生後に生じたカウントは、コレクタ関数の `collector_final_counters` が原因とされます。この関数がカウント値のかなりの割合を占める場合は、オーバーフロー値が大きすぎます。

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` の上で割り込み処理を行うことによって、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延びる可能性があります。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ルはキャッシュミスなどのメモリー関係のイベントの報告データをまとめたもので、メモリー関係のイベントが発生する命令だけではなく、イベントを発生させるデータオブジェクト参照についても報告します。データ空間プロファイリングは、Linux システムでは利用できません。

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 -xdebugformat=dwarf -g` フラグを付けて SPARC® 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、カウンタ名の先頭にオプションの + 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**」と「**データレイアウト**」という 2 つのタブが装備されています。

MPI トレース

MPI トレースは、Solaris™ プラットフォームでのみ利用できます。MPI トレース機能は、MPI ライブラリ関数の呼び出しに関する情報を記録します。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、および送受信動作の数と送受信バイト数で構成されます。トレースは、MPI ライブラリの呼び出し上で割り込み処理を行うことによって実施します。割り込み関数は、データ送信の最適化に関する情報や送信エラーに関する情報を持たないので、提示される情報は、以降で説明する単純な形でのデータ送信を表しています。

受信バイト数は、MPI 関数の呼び出しで定義されるバッファサイズです。実際に受信したバイト数は、割り込み関数には利用できません。

一部の「大域通信」関数は、ルートと呼ばれる 1 つの受信プロセスまたは 1 つの起点を持ちます。こういった関数のアカウンティングは、次のように行われます。

- ルートが自分自身を含むすべてのプロセスにデータを送信する。
- ルートが自分自身を含むすべてのプロセスからデータを受信する。
- 各プロセスが自分自身を含む他の各プロセスと通信する。

次の例は、アカウンティングの手順を示しています。これらの例における G は、グループのサイズです。

`MPI_Bcast()` の呼び出しの場合、

- ルートは N バイトのパケット G 個を、自分自身を含む各プロセスに対して 1 個ずつ送信する。
- グループ内の G 個のプロセスすべてが (ルートを含む) N バイトを受信する。

`MPI_Allreduce()` の呼び出しの場合、

- 各プロセスが N バイトのパケットを G 個送信する。
- 各プロセスが N バイトのパケットを G 個受信する。

`MPI_Reduce_scatter()` の呼び出しの場合、

- 各プロセスが N/G バイトのパケットを G 個送信する。
- 各プロセスが N/G バイトのパケットを G 個受信する。

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 (PC) のアドレスです。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、次に実行す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 (「スタックの展開」と呼ぶ)。展開の失敗については、133 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を割り当てるときに使用されます。スタック上の各 PC (リーフ PC を含む) は、その PC が存在する関数に包括的メトリックを割り当てるときに使用されます。

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が紛らわしいことがあります。こうした事例については、135 ページの「プログラム構造へのアドレスのマップ」を参照してください。

シングルスレッド実行と関数の呼び出し

プログラムの実行でもっとも単純なものは、シングルスレッドのプログラムがそれ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、分岐命令 (たとえば、関数呼び出しや条件文を表すことがある) 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されるようにコミットされています。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の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他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。あらゆる共有オブジェクトには、それぞれにプログラムリンケージテーブル (PLT) が 1 つあり、その PLT には、そのオブジェクトが参照する関数で、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 命令の中の 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 命令ではない場合、ld.so のコードと PLT が起因する PC はすべて、包括的時間を集計する擬似的な関数 `@plt` の呼び出しと置き換えられます。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、さまざまなレジスタおよびスタック操作が発生し、シグナル送信時のリーフ PC が、システム関数 `sigacthandler()` への呼び出しの復帰アドレスを示していたかのようになります。`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、そして `sigacthandler()` が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他の関数の包括的メトリックは、割り込まれた関数の包括的メトリックとして表示されます。

コレクタは sigaction() 上で割り込み処理を行うことによって、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であり、ハードウェアカウンタデータ収集時には SIGEMT シグナルのプライマリハンドラであることを確保します。

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。たとえば、あらゆるシステムコールは、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™ 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、トラップを発行したユーザーのコードの命令またはシステムコールが原因とされます。

一部のシステムコールでは、こうした呼び出しをできる限り効率よ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は、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としては、gethrtime や 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えられません。

その他、トラップをできる限り効率よく処理することが重要とみなされる環境もあります。たとえば、マイクロステートが切り替えられていないレジスタウィンドウのスピルやフィル、および TLB (translation lookaside buffer) ミスなどです。

いずれの場合も、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、できれば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考慮することによって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、パフォーマンスアナライザでハンドラの呼び出しを表すことができますが、その時間は整数ロードまたはストア命令が原因とされます。

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後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。

テール呼び出しの最適化

特定の関数がその最後で他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® マシン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
```

B および C に対してテール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、テール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は、2 以上のレベルでテール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™ オペレーティングシステムでは、簡単なプログラムは、単一の LWP (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、その関数にターゲット関数が渡されます。ターゲットが存在する場合、スレッドはスレッドライブラリによって破壊されます。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()` という関数の位置で動作を開始します。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係する包括的メトリックは、`_thread_start()` と <**合計**> 関数に加算されるだけです。

スレッドライブラリは、スレッドを作成するほかに、スレッドを実行するための LWP も Solaris™ 上に作成します。スレッド化は、結合スレッド (特定の 1 つの LWP に結合されるスレッド)、または非結合スレッド (異なるタイミングで異なる LWP にスケジューリングすることが可能なスレッド) のどちらを使用しても行うことができます。

- 結合スレッドが使用された場合、スレッドライブラリは 1 つのスレッドに LWP を 1 つ作成します。

- 非結合スレッドが使用された場合、スレッドライブラリは、作成する LWP の個数 (効率的に動作する個数) とそれらスレッドのスケジューリング先の LWP を決定します。スレッドライブラリは、必要に応じて後で複数の LWP を作成できます。非結合スレッドは、Solaris™ 9 オペレーティングシステムの一部ではなく、Solaris 8™ オペレーティングシステムの代替スレッドライブラリの一部でもありません。

  非結合スレッドのスケジューリングの一例として、スレッドが `mutex_lock` などによって同期が阻まれているときに、スレッドライブラリは、最初のスレッドが動作していた LWP に別のスレッドをスケジューリングできます。同期を阻まれていたスレッドがロック待ちに費やした時間は、同期待ち時間メトリックに反映されますが、LWP がアイドルではないため、その時間はユーザーロック時間メトリックに加算されません。

Solaris™ 8 オペレーティングシステムの標準スレッドライブラリは、ユーザースレッド以外にも、シグナル処理やその他のタスクを行うためのスレッドを作成します。結合スレッドを使用するプログラムの場合、結合スレッド用の LWP も作成されます。これらの結合スレッドはほとんどの時間をスリープ状態で費やすので、そのパフォーマンスデータの収集や表示は行われません。ただし、プロセス統計、および標本データに記録される時間値には、これらのスレッドで消費される時間が含まれます。Solaris™ 9 オペレーティングシステムのスレッドライブラリと Solaris™ 8 オペレーティングシステムの代替スレッドライブラリは、こういった追加スレッドの作成を行いません。

Linux オペレーティングシステムは、明示的なマルチスレッド化のための P- スレッド (POSIX スレッド) を提供します。データ型の `pthread_attr_t` は、スレッドの動作属性を制御します。結合スレッドを作成するには、属性のスコープは、`pthread_attr_setscope()` 関数を使って `PTHREAD_SCOPE_SYSTEM` に設定する必要があります。デフォルトまたは属性スコープが `PTHREAD_SCOPE_PROCESS` に設定されている場合は、スレッドは非結合です。新しいスレッドを作成するために、アプリケーションは P- スレッド 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。新しいスレッドは実行を開始すると、Linux 固有のシステム関数 `clone()` で動作します。この関数は、別の内部初期化関数 `pthread_start_thread()` を呼び出し、これは、当初は `pthread_create()` に渡されるユーザー定義の起動ルーチンを呼び出します。コレクタで使用できる Linux メトリック収集関数は、スレッドが LWP に結合されているかどうかに関係なく、スレッドに固有です。したがって、`collect` ユーティリティを実行すると、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別のプログラムのように動作します。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他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 (純粋なものか、C/C++ が混合しているもの) で書かれたアプリケーションは Java™ 仮想マシン (JVM[1]) 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、メインなどを呼び出す _start から実行を開始します。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。

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。その他のメソッドは、Java HotSpot™ 仮想マシンによってコンパイルされ、コンパイルされたメソッドと呼ばれています。動的に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。特定のメソッドについては、インタプリタされたバージョンがあるほか、1 つ以上のコンパイルされたバージョンもあります。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++、Fortran、またはネイティブコンパイル (SBA SPARC® Bytecode Accelerator) Java™ 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。

JVM ソフトウェアは、従来の言語で書かれたアプリケーションでは一般に行われない多数のことを行います。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。これらのうちの 1 つは、アプリケーションのバイトコードメソッドの処理に使用する実際のインタプリタコードです。

インタプリタの実行中、Java HotSpot™ 仮想マシンはパフォーマンスをモニタし、インタプリタを行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ができます。生成されたマシンコードは、プロセスのデータ空間内にもあります。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、他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アプリケーションは本質的にマルチスレッド型であり、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

1. 「Java 仮想マシン (JVM)」という用語は、Java™ プラットフォーム用仮想マシンを意味します。

かあります。libthread.so のバージョンにより、スレッドと LWP 間に 1 対 1 の対応関係またはそれより複雑な関係があります。Solaris™ 8 上のデフォルトの libthread.so スレッドライブラリの場合は、いつでもスレッドをスケジュール解除したり、LWP にスケジュールすることができます。スレッドのデータは、そのスレッドが LWP にスケジュールされていない間は収集されません。Solaris™ 8 上の代替 libthread.so ライブラリや Solaris™ 9 スレッドを使用しても、スレッドはスケジュール解除できません。

Java とマシン呼び出しスタック

パフォーマンスツールは、イベント時に呼び出しスタックのほか、各 Solaris™ LWP または Linux スレッドの有効期間中にイベントを記録することによってデータを収集します。呼び出しスタックは、アプリケーション (Java™ とそうでないものがある) の実行中はいつでも、プログラムが実行中のどの場所にあり、どのようにしてそこまで到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は異なる 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。

時間およびハードウェアカウンタプロファイル

Java™ プログラムに対する時計およびハードウェアカウンタプロファイル機能は、Java™ と マシンの呼び出しスタック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。

同期トレース

Java™ プログラムの同期トレースは、スレッドが Java™ モニタを取得しようとしたときに生成されるイベントに基づいています。これらのイベントに関してはマシンと Java™ の呼び出しスタックがともに収集されますが、JVM 内で使用される内部ロックに関しては同期トレースデータが収集されません。

ヒープトレース

ヒープトレースデータは、オブジェクト割り当てイベント (ユーザーコードで生成される) とオブジェクト割り当て解除イベント (ガーベッジコレクションで生成される) を記録します。また、malloc、free 等の C/C++ メモリー管理関数を使用すると、記録されるイベントも生成されます。これらのイベントは、ネイティブコードから発生するか、JVM 自体から発生します。

Java 処理の表現

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™ 表現、上級 Java™ 表現、マシン表現があります。デフォルトでは、データが Java™ 表現をサポートする場合は、Java 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。

Java 表現

Java™ 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、特定の Java™ メソッドのインスタンスが、多数存在する場合があります。Java™ 表現では、すべてのメソッドが 1 つのメソッドとして集合された状態で表示されます。デフォルトでは、このモードがアナライザで選択されます。

Java™ メソッド (Java™ 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索引に対応し、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 が混ざり合っていることがあります。呼び出しスタックには、Java™ 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 は Java™ スタックを展開することができず、特別な関数 `<no Java™ callstack recorded>` 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。

ハウスキーピングスレッドの場合は、マシン呼び出しスタックのみが取得され、Java™ 表現内では、これらのスレッドのデータが特別な関数 `<JVM-Overhead>` が原因とされます。

Java™ 表現の関数リストは、Java™ メソッドと呼び出されたネイティブメソッドに対するメトリックを示します。また、このリストは、JVM オーバーヘッドスレッドの疑似関数も示します。呼び出し元と呼び出し先のパネルには、呼び出しの関係が Java™ 表現で示されます。

Java™ メソッドのソースはそのメソッドがコンパイルされた `.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とインタリーブされた Java™ ソースを示します。

Java™ 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™ プログラムは、クラスをインスタンス化してデータを格納するためのメモリーを割り当てますが、C および C++ アプリケーションとは異なり、メモリーの明示的な割り当て解除は行いません。メモリーは、いわゆるガベージコレクタによって管理されます。JVM の一部であるこのコードは、定期的にメモリーをスキャンして、もはやプログラムの他の部分でポイントされていない割り当て領域を見つけます。そし

て、メモリーを割り当て解除して他で利用できるようにすることで、これを再生します。Java™ 表現のヒープトレースは Java™ メモリー管理と JVMPI イベントに基づいており、通常のヒープトレースのデータは Java™ 表現にも示されます。

すべての Java™ プログラムは、通常は `monitor-enter` ルーチンを呼び出すことで明示的同期化を実行できます。

Java™ 表現の同期遅延トレースは、JVMPI 同期イベントをベースとします。通常の同期トレースのデータは、Java™ 表現では表示されません。

Java™ 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 Java 表現

上級 Java™ 表現は、JVM 内部要素の詳細のいくつかを除いては Java™ 表現に似ています。Java™ 表現では表示されない JVM 内部要素の詳細のいくつかが、上級 Java™ 表現に表されます。ネイティブフレームは、`<no Java Callstack recorded>` が Java™ 表現に表示された場合に使用されます。上級 Java™ 表現のタイムラインには、すべてのスレッドが示されます。

マシン表現

マシン表現には、JVM 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 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、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。

マシン表現では、一般に非常に大きいチャンクでメモリーが JVM で割り当てられ、割り当て解除されます。Java™ コードからのメモリー割り当てはすべて、メモリーをマップすることで JVM とそのガーベッジコレクションで処理されます。ヒープトレース時には、メモリーマップ操作はメモリー割り当てとして扱われるため、ヒープトレースは JVM 割り当てを示します。

マシン表現では、スレッド同期は _lwp_mutex_lock の呼び出しになります。これらの呼び出しはトレースされないため、同期データは示されません。

並列実行とコンパイラ生成の本体関数

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、並列実行用にコンパイルできます。OpenMP は、Sun のコンパイラやツールで利用できる機能です。『OpenMP API ユーザーズガイド』、『Fortran プログラミングガイド』および『C ユーザーズガイド』の関連箇所、または OpenMP 標準を規定している Web サイト http://www.openmp.org を参照してください。

ループまたは他の並列構造を並列実行用にコンパイルすると、マイクロタスクライブラリによる調整を受けながら、コンパイラ生成コードが複数のスレッドによって実行されるようになります。Sun のコンパイラによって行われる並列化処理の概略は、以下に示すとおりです。

本体関数の生成

コンパイラは並列構造を検出した場合、並列構造の本体を独立した本体関数にし、マイクロタスクライブラリの関数の呼び出しにその構造を置き換えることによって、並列実行用のコードを生成します。マイクロタスクライブラリ関数は、本体関数を実行するためにスレッドをディスパッチする作業をします。本体関数のアドレスは、引数としてマイクロタスクライブラリの関数に渡されます。

並列構造が次のリスト内の指令のいずれかによって区切られている場合、この並列構造は、マイクロタスクライブラリ関数 __mt_MasterFunction_() への呼び出しと置き換えられます。

- Sun Fortran の c$par doall 指令
- Cray Fortran の c$mic doall 指令
- Fortan の OpenMP 指令の c$omp PARALLEL、c$omp PARALLEL DO、c$omp PARALLEL SECTIONS のいずれか
- C または C++ の OpenMP 指令の #pragma omp parallel、#pragma omp parallel for、#pragma omp parallel sections のいずれか

また、コンパイラによって自動的に並列化されるループも、__mt_MasterFunction_() への呼び出しに置き換えられます。

1 つまたは複数の worksharing do、for、または sections 指令を含んでいる OpenMP 並列構造の場合、各 worksharing 構造はマイクロタスクライブラリ関数 __mt_Worksharing_() への呼び出しに置き換えられ、それぞれについて新しい本体関数が作成されます。

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、オリジナルソースにおける構造の先頭の行番号、および並列構造のシーケンス番号をエンコードする名前を本体関数に設定します。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。

並列実行シーケンス

プログラムの実行は、1 つのスレッド (メインスレッド) からのみ開始されます。プログラムが初めて `__mt_MasterFunction_()` を呼び出すと、この関数が、ワークスレッドを作成するために、Solaris™ スレッドライブラリ関数の `thr_create()` を呼び出します。各ワークスレッドは、`thr_create()` に引数として渡されていたマイクロタスクライブラリ関数の `__mt_SlaveFunction_()` を実行します。

Solaris™ 8 のオペレーティングシステムの標準スレッドライブラリは、ワークスレッド以外にも、シグナル処理や他のタスクを行うためのスレッドを作成します。これらのスレッドはほとんどの時間をスリープ状態で費やすので、そのパフォーマンスデータは収集されません。ただし、プロセス統計、および標本データに記録される時間値には、これらのスレッドで消費される時間が含まれます。Solaris™ 9 オペレーティングシステムのスレッドライブラリと Solaris™ 8 オペレーティングシステムの代替スレッドライブラリは、こういった追加スレッドの作成を行いません。

すべてのスレッドが作成されると、`__mt_MasterFunction_()` はメインスレッドとワークスレッド間の作業の配分を管理します。作業がない場合は、`__mt_SlaveFunction_()` が `__mt_WaitForWork_()` を呼び出し、そこで、ワークスレッドは作業を待ちます。作業が発生すると、ワークスレッドはすぐに `__mt_SlaveFunction_()` に制御を戻します。

作業がある場合は、各スレッドによって `__mt_run_my_job_()` が呼び出され、本体関数に関する情報が渡されます。ここからの実行シーケンスは、その本体関数が parallel sections、parallel do (または parallel for)、parallel のどの指令から生成されたかによって異なります。

- parallel sections の場合は、`__mt_run_my_job_()` が本体関数を直接呼び出します。
- parallel do または `parallel for` の場合は、`__mt_run_my_job_()` が別の複数の関数 (ループの性質によって異なる) を呼び出し、それらの関数によって本体関数が呼び出されます。
- parallel の場合は、`__mt_run_my_job_()` が本体関数を直接呼び出し、すべてのスレッドが、`__mt_WorkSharing_()` の呼び出しがあるまで本体関数内のコードを実行します。この関数には、`__mt_run_my_job_()` に対する呼び出しがもう 1 つあります。この呼び出しでは、worksharing section の場合は直接に、また worksharing do または for の場合は他のライブラリ関数を介して間接に、ワークシェアリング用の本体関数を呼び出します。worksharing 指令に `nowait` が指定されている場合、各スレッドは並列本体関数に制御を戻し、動作を続行します。

nowait が指定されていない場合は __mt_WorkSharing_() に制御を戻し、このルーチンが __mt_EndOfTaskBarrier_() を呼び出して、スレッド間の同期を取ります。

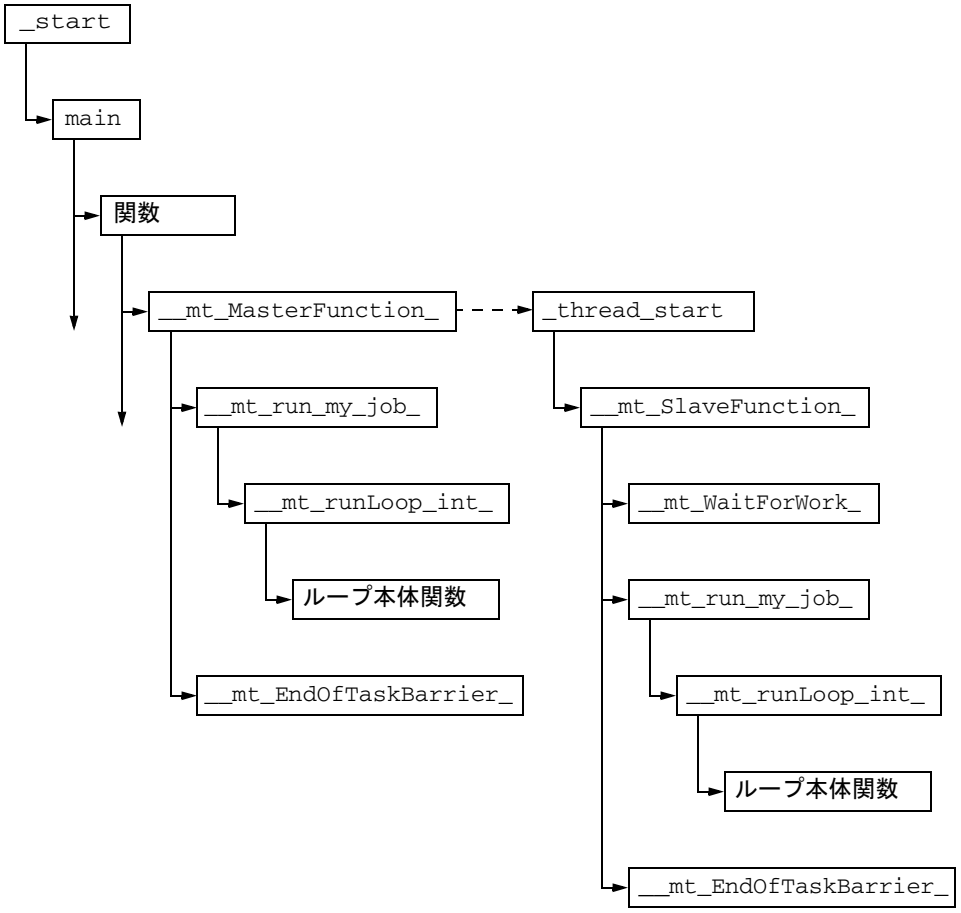

図 6-1 Parallel Do または Parallel For 構造を含むマルチスレッドプログラムの呼び出しツリー

すべての並列作業が完了すると、スレッドは __mt_MasterFunction_() または __mt_SlaveFunction_() に制御を戻し、__mt_EndOfTaskBarrier_() を呼び出して、並列構造の終了に関係する同期の作業を行います。すべてのワークスレッドは再び __mt_WaitForWork_() を呼び出し、メインスレッドはシリアル領域で引き続き動作します。

ここで説明した呼び出しシーケンスは、並列に動作するプログラムだけでなく、並列化用のコンパイルしたプログラムであっても、単一 CPU マシンまたは LWP を 1 つだけ使用するマルチプロセッサマシンで動作するプログラムに当てはまります。

図 6-1 は、簡単な parallel do 構造の呼び出しシーケンスを示しています。ワークスレッドの呼び出しスタックは、スレッドライブラリ関数の _thread_start() (実際にはこの関数は __mt_SlaveFunction_() を呼び出す) から始まります。点線の矢印は、__mt_MasterFunction_() から thr_create() への呼び出しの結果としてスレッドの実行が開始されることを示しています。終点のない矢印は、図には現れていない、その他の関数の呼び出しがある可能性があることを示しています。

図 6-2 は、worksharing do 構造を含む並列領域の呼び出しシーケンスを示しています。mt_run_my_job_() の呼び出し元は、__mt_MasterFunction_() と __mt_SlaveFunction_() のいずれかです。図 6-1 の __mt_run_my_job_() の呼び出しをこの図全体に置き換えることができます。

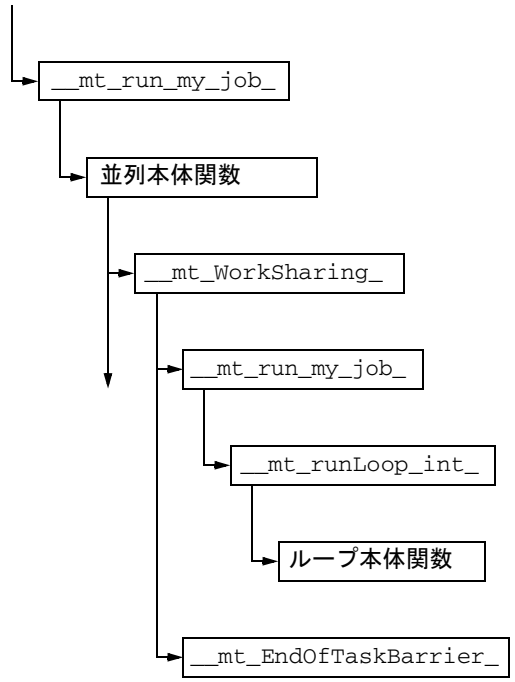

図 6-2 Worksharing Do または Worksharing For 構造を含む並列領域の呼び出しツリー

これらの呼び出しシーケンスでは、コンパイラによって生成されたすべての本体関数が、マイクロタスクライブラリ内の同じ関数 (複数のこともある) から呼び出されます。このため、本体関数のメトリックを元のユーザー関数に関連付けるのは困難になります。パフォーマンスアナライザは対応する呼び出しを元のユーザー関数から本体関数に挿入し、マイクロタスクライブラリは対応する呼び出しを本体関数からバリア関数 __mt_EndOfTaskBarrier_() に挿入します。このため、同期が原因のメトリックは本体関数に加算され、本体関数のメトリックは元の関数に加算されます。こうした挿入によって、本体関数の包括的メトリックは、マイクロタスクライブラリ関数ではなく、元の関数のメトリックに直接加算されます。このように呼び出しを付加

すると、その結果として、本体関数が元のユーザー関数とマイクロタスク関数の両方の呼び出し先として表示されます。さらには、ユーザー関数の呼び出し元がマイクロタスクライブラリ関数であるように、またユーザー関数が自分自身を呼び出すように見えます。包括的メトリックを二重にカウントすることは、再帰関数呼び出しに使用されている仕組みによって回避されています (20 ページの「関数レベルのメトリックに再帰が及ぼす影響」参照)。

一般に、ワークスレッドは、新しい作業が届くと (すなわち、メインスレッドが新しい並列構造に達すると)、待ち時間を短縮するために、`__mt_WaitForWork_()` にある間に CPU 時間を使用します。これは `busy-wait` として知られています。ただし、環境変数でスリープ待機を指定することもでき、この場合、パフォーマンスアナライザでは、この時間は ユーザー CPU 時間ではなくその他の待ち時間になります。一般に、ワークスレッドが作業待ちに時間を費やす状況としては、以下の 2 つの場合があります。このような場合は、プログラムを設計し直して、待ち時間を短縮することを推奨します。

- メインスレッドがシリアル領域で動作していて、ワークスレッドが行う作業がない。
- 作業負荷が不均衡で、作業を終了して待機しているスレッドと作業を続行しているスレッドが存在する。

デフォルトでは、マイクロタスクライブラリは LWP に結合されたスレッドを使用します。Solaris™ 8 のオペレーティングシステムにおけるこのデフォルトの設定は、`MT_BIND_LWP` 環境変数を `FALSE` に設定することによって変更できます。デフォルトの変更は推奨されません。

注 – 多重処理のディスパッチプロセス全体は実装状態に依存しません。このプロセスは、将来のリリースで変更される可能性があります。

不完全なスタック展開

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。

- スタックがユーザーコードで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がコアダンプを行う場合があります。
- 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。

  ほかのプラットフォームでも、手書きのアセンブラコードに規則違反がある場合があります。
- Intel プラットフォームで、C または Fortran コードで高レベルの最適化のもとにコンパイルが行われている場合。このことは展開するうえでのフレームポインタを持たないことを意味します。

- Intel プラットフォームで、C++ コードがその最適化のレベルにかかわらず、-noex または -features=no%except オプションを使用してコンパイルされている場合。
- リーフ PC が本体関数のスタックからフレームがポップされた後、関数が戻される前に関数内にある場合。
- 250 を超えるフレームが呼び出しスタックに含まれている場合、この呼び出しスタックを完全に展開するだけの容量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数え上げるために <Total> から呼び出されたものとして示されます。

中間ファイル

-E または -P の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、パフォーマンスアナライザはオリジナル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 #line 指令を生成すると、ソース行へのメトリックの割り当てで問題が発生する原因となります。

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、次の条件の下で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
- コンパイルの後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行番号がある場合。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ためのオブジェクトファイルが見つからない場合。
- ファイルが -g フラグを使用しないでコンパイルされた場合、またはファイルがストリップ済みの場合。
- 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プ

パフォーマンスアナライザは、呼び出しスタックの内容を処理して PC 値を生成した後に、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、逆アセンブリ行 (命令) にマップします。ここでは、これらのマップ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 アドレスは、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、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解除できるため、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後に、同じオブジェクトが別のアドレスに再度読み込まれた場合は、時間によって同じ関数に別の PC が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、さまざまなシンボルテーブルが含まれます。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、この ELF シンボルテーブルには、そのオブジェクトの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演算を表す一群の命令を説明するために使用されます。この用語は、Fortran で使用されているサブルーチン、C++ や Java™ などで使用されているメソッドなども表します。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。プログラムカウンタ値がアドレスセットに含まれているということは、プログラムの実行がその関数で起こっていることを意味します。

基本的に、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。呼び出しスタック上のリーフ PC および他のすべての PC について、まったく同じマップ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

通常、関数は大域関数と定義されます。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、実行可能な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、実行時リンカーはそのうちの 1 つに対するすべての参照を解決します。その他の関数は実行されず、このため、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。一般的な例として、たとえば、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などです。一般に、強い名前は対応する弱い名前と同じですが、最後に下線 (_) が付きます。スレッドライブラリ内の多くの関数にも、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™ スレッド用の別の名前があります。いずれの場合も、パフォーマンス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。使用されるのは、与えられたアドレス位置のアルファベット順で最後の名前です。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、選択されている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。この逆に、複数のコード部分に同一名が使用されている場合もあります。

- 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。このことは、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、アナライザでは、同じ名前の複数の関数がプログラムのまったく異なる部分を参照しているように表示されます。「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または中間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置き換えられていることがあります。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

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、パフォーマンスアナライザは、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、その関数の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。パフォーマンスアナライザは、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後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、パフォーマンス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 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に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、関数の最終的な復帰用のスタックが作成され、その後で、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、パフォーマンスアナライザに、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。最適化できる呼び出しを見つけると、コンパイラは、この関数のコピー (クローンと呼ばれる) を作成し、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の符号化を解除し、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。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 関数のあらゆるコピーについてデータが集計されます。

インライン化された関数

インライン化された関数は、コンパイルすると実際の呼び出しの代わりに関数の呼び出し位置に命令が挿入されます。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、パフォーマンス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) で行われた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、命令数が増え、そのためレジスタの利用や命令の実行スケジューリングの最適化に影響が出ることがあります。

いずれのインライン化も、メトリック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現れません。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とみなされるメトリック (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わす) が原因の排他的メトリックと報告されます。

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、関数をインライン化しても、いわゆる行の範囲外 (out-of-line) の関数が残ることがあります。ある呼び出し場所では、その行の範囲外の関数が呼び出され、別の場所では命令がインライン化されることがあります。このような場合は、関数リストに関数が表示されますが、その関数が原因のメトリックには、行の範囲外の呼び出しだけが反映されます。

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、129 ページの「並列実行とコンパイラ生成の本体関数」で詳しく説明しています。

パフォーマンスアナライザは、このような本体関数を通常の関数として表示し、コンパイラ生成名に加え、その関数が抽出された関数に基づいてその関数に名前を割り当てます。こうした関数の排他的および包括的メトリックは、本体関数で費やされた時間を表します。また、構造が抽出された関数は各本体関数の包括的メトリックになります。このことがどのように行われるかについては、130 ページの「並列実行シーケンス」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` でコンパイルされたモジュールに対してのみ復号化することができます。

アウトライン関数

フィードバックデータを利用した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。アウトライン関数は、通常は実行されないコードです。具体的には、最終的な最適化コンパイル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などです。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。パフォーマンスアナライザは、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ことはなく、ジャンプ先になります。同じ意味で、アウトライン関数が復帰することはなく、ジャンプ先から戻ることになります。動作をユーザーのソースコードモデルに近づけるために、パフォーマンスアナライザは、メイン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。

アウトライン関数には、通常の関数として、適切な包括的および排他的メトリックが表示されます。また、アウトライン関数のメトリックは、アウトライン化が行われた元の関数の包括的メトリック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 ユーザーガイド』の第 3 章で、`-xprofile` コンパイル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て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していません。API 関数については、31 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 **<未知>** 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。他のメソッドの場合、メソッドを実行する Java™ 仮想マシンに関する情報がパフォーマンスツールに表示されます。Java™ モードでは、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。マシンモードでは、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。

<未知> 関数

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 **<未知>** という特別な関数にマップされます。

PC が **<未知>** にマップされるのは、次のような場合です。

- C や 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、31 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™ メソッドは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。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 V9 マシン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。この問題について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 が復帰レジスタで、既知のロードオブジェクト内に存在しないようにみえる場合は、<**未知**> 関数に原因が帰せられて、無視されます。
- コレクタにシンボリック情報がない Java™ 仮想マシンの内部部分に PC がマップしている場合。

<**未知**> 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、正しく処理されます。

<JVM-Overhead> 関数

Java™ 表現では、`<JVM-Overhead>` 関数は、JVM が Java™ プログラムの実行以外のアクションを実行するために使用した時間を示します。JVM は、ガーベージコレクションや HotSpot™ コンパイルなどのタスクを、この時間間隔で実行します。デフォルトでは、`<JVM-Overhead>` は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 <**未知**> 関数に似ていますが、Java™ スレッドの場合は Java™ 表現でのみ表されます。コレクタが Java™ スレッドからイベントを受信すると、コレクタは Java™ 仮想マシン内へネイティブスタックと呼び出しを展開して対応する 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、疑似関数 `<no Java callstack recorded>` でアナライザ内にイベントが表示されます。JVM 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™ スタックを展開したために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を記録するためにアナライザが使用するバッファ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()` を始めとする最も重要度の低いスタック下部の関数のメトリックからドロップします。ドロップされた関数のメトリックは、1 つの疑似関数 `<Truncated-stack>` にまとめられます。`<Truncated-stack>` 関数は、Java プログラムでも使用できます。

<合計> 関数

<合計> 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造です。あらゆるパフォーマンスメトリックは、呼び出しスタック上の関数のメトリックとして加算されるほかに、**<合計>** という特別な関数のメトリックに加算されます。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他の関数のデータの概略を見ることができます。特別な関数の <合計> は、あらゆる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、**<合計>** 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。

HW カウンタプロファイルに関連する関数

次の関数は、HW カウンタ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_lost_hwc_overflow`: 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_lost_sigemt`: 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は失われたように見えます。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が失敗すると、ハードウェアカウンタの収集が終了します。
- `collector_final_counters`: 収集の中断または終了の直前に取られたカウンタの値で、直前のオーバーフロー以降のカウントです。このカウントが **<合計>** カウントの有効部分に対応する場合、それより小さいオーバーフロー間隔 (すなわち、さらに高い分解能の構成) を推奨します。

- collector_record_counters: ハードウェアカウンタイベントの処理および記録中に蓄積されたカウントで、部分的にハードウェアカウンタプロファイルのオーバーヘッドを計算します。このカウントが <**合計**> カウントの有効部分に対応する場合、それより大きいオーバーフロー間隔 (すなわち、さらに低い分解能の構成) を推奨します。

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プ

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うまくバックトラックするように処理されると、パフォーマンスアナライザは、ハードウェアプロファイルサポート情報内のコンパイラから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誘導します。

データオブジェクトという用語は、プログラム定数、変数、配列、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。ソース言語により、データオブジェクトのタイプとそのサイズは変わります。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、他の単純なデータオブジェクトから誘導または集められるものがあるため、データオブジェクトの集合はしばしば複雑なものになります。

各データオブジェクトは、関連する適用範囲、その適用範囲が定義されて参照できるソースプログラムの領域 (大域的なもの、たとえばロードオブジェクト)、特定のコンパイルユニット (すなわち、オブジェクトファイル)、または関数を持っています。同一のデータオブジェクトを異なる適用範囲で定義したり、特定のデータオブジェクトを異なる適用範囲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演算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は、関連するプログラムのデータオブジェクトタイプに加算され、そのデータオブジェクトとすべてのデータオブジェクト (<**未知**> と <**スカラー**>) を含む疑似の <**合計**> を含む集合体に伝搬します。<**未知**> のさまざまなサブタイプは、<**未知**> の集合体まで伝搬します。以降では、<**合計**>、<**スカラー**>、<**未知**> 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完全に記述できます。単純なスカラーデータオブジェクト「`{int i}`」は、型「`int`」の変数「`i`」を記述するのに対して、「`{const+pointer+int p}`」は「`p`」と呼ぶ型「`int`」への定数ポ

インタを記述します。型名のスペースは「_」(アンダースコア) と置き換えられ、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」の構造体について「{structure:foo_t}」と表されます。集合体の要素は、直前の「foo_t」型の構造体の型「int」のメンバ「i」に対してその要素のコンテナ、たとえば、「{structure:foo_t}.{int i}」と追加指定する必要があります。集合体はそれ自体、(さらに大きい) 集合体の要素である可能性もあり、それらの要素の対応する記述子は集合体記述子、最終的にはスカラー記述子を連結したものとして構成されています。

完全修飾された記述子は、必ずしもデータオブジェクトを明確にする必要はありませんが、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仕様を示します。

<合計> データオブジェクト

<**合計**> 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) のメトリックとして加算されるほかに、<**合計**> 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関数リストの先頭に表示され、そのデータを使用して他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。

<スカラー> 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は関連する集合体のメトリック値にさらに加算されますが、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は擬似的な <**スカラー**> 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。

<未知> データオブジェクトとその要素

さまざまな環境で、特定のデータオブジェクトにイベントデータをマップすることができません。このような場合、データは <**未知**> という特別なデータオブジェクトおよび次に示すその要素の 1 つにマップされます

- **確定不可**

  1 つ以上のコンパイルオブジェクトはハードウェアプロファイルサポートなしでコンパイルされたので、メモリー参照命令に関連するデータオブジェクトを確認したりバックトラッキングの妥当性を検査することはできません。

- **確認不可**

  コンパイルオブジェクトに提供されているハードウェアプロファイルサポート情報は、バックトラッキングの妥当性を検査するには不十分なものでした。

- **解決不可**

イベントバックトラッキングで制御転送ターゲットが発生し、また、制御転送が実際に発生したかどうかを確認できないために、原因と思われるメモリー参照命令 (およびそれに関連するデータオブジェクト) を判別できませんでした。

- **未指定**

  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。

- **未識別**

  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で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。

第7章

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、関数内の演算がパフォーマンス低下の原因になっているコードを解析するときに役立ち、コンパイラがコードに対して行った変換処理に関するコメントを表示できます。この章では、注釈の生成処理と、注釈付きコードを理解するにあたっての問題点をいくつか説明します。

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の「アナライザデータ表示」ウィンドウの左の区画の「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` ユーティリティを使って、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` を使った注釈付きソースコードの表示の詳細は、170 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。

パフォーマンスアナライザの注釈付きソースには、以下の情報が含まれます。

- 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
- 特定のしきい値を超えたメトリック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ウィンドウのレイアウト

ソースウィンドウには、いくつかの欄が表示されます。ウィンドウの左側には、個々のメトリックを表示する固定幅の欄が表示されます。ウィンドウ右側の残りの欄には、注釈付きソース用の欄が表示されます。

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。注釈付きソース欄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た、またはバイトコードにインタプリタされたファイルを示します。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。パフォーマンス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コード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。パフォーマンスアナライザは、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示します。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で、テキストが山括弧内に示されます。最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。以下の例は、C 関数 `icputime` のインデックス行を示しています。

```
                600. int
                601. icputime(int k)
0.      0.      602. {
0.      0.           <Function: icputime>
```

この例で分かるように、インデックス行は、最初の命令の後の行に表示されます。C ソースの場合は、最初の命令は、関数本体の先頭の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、以下の例に示されているように、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 02/04/2000 Sun Microsystems, Inc. All Rights Reserved
              2. !
              3. ! Synthetic f90 program, used for testing openmp directives and
              4. !       the analyzer
              5.
              6. !$PRAGMA C (gethrtime, gethrvtime)
              7.
0.  81.497[  8] 9000    format ('X', f7.3, 7x, f7.3, 4x, a)
                 <Function: main>
```

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、コードは #include されている場合や他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。

オブジェクトコードのソースが、オブジェクトコードに含まれている関数のデフォルトソースコンテキストでない場合は、オブジェクトコードに対応する注釈付きソースには、関数のデフォルトソースコンテキストを相互参照する特別なインデックス行が含まれます。たとえば、synprog デモをコンパイルすると、ソースファイル endcases.c に対応するオブジェクトモジュール endcases.o が作成されます。endcases.c 内のソースは、ヘッダー inc_func.h 内で定義されている関数 inc_func を宣言します。ヘッダーにはしばしば、このようなインライン関数定義が含まれます。endcases.c は関数 inc_func を宣言しますが、endcases.c 内のソース行で inc_func の命令に対応するものはないため、endcases.c の注釈付きソースファイルの先頭には、以下のような特別なインデックス行が示されます。

```
0.650  0.650   <Function: inc_func, instructions from source file inc_func.h>
```

このインデックス行のメトリックは、endcases.o オブジェクトモジュールのそのコード部分の行は (ソースファイル endcases.c 内の) マップが行われていないことを示します。

パフォーマンスアナライザはまた、関数 inc_func が定義されている inc_func.h の注釈付きソースに標準インデックス行も追加します。

```
                2.
                3. void
                4. inc_func(int n)
0.     0.       5. {
                   <Function: inc_func>
```

同様に、関数に代替ソースコンテキスト[1]がある場合は、そのコンテキストを相互参照するインデックス行が、デフォルトソースコンテキストの注釈付きソースに表示されます。

```
                  142. inc_body(int n)
0.650  0.650        <Function: inc_body, instructions from source file inc_body.h>
0.         0.     143. {
                          <Function: inc_body>
                  144. #include "inc_body.h"
0.         0.     145. }
```

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、そのソースファイルの内容がソースウィンドウに表示されます。

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、162 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。コンパイル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。

1. 代替ソースコンテキストは、関数に属する命令を含む他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 (前述の例を参照) 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。

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、ループの最適化や並列化、インライン化、パイプライン化があります。以下に、コンパイラのコメントの例を示します。

```
0.     0.   Function freegraph inlined from source file ptraliasstr.c into the
code for the following line
              47.       freegraph();
              48.     }
0.     0.     49.     for (j=0;j<ITER;j++) {

                Function initgraph inlined from source file ptraliasstr.c into
the code for the following line
0.     0.     50.       initgraph(rows);

                Function setvalsmod inlined from source file ptraliasstr.c into
the code for the following line
                Loop below fissioned into 2 loops
                Loop below fused with loop on line 51
                Loop below had iterations peeled off for better unrolling and/or
parallelization
                Loop below scheduled with steady-state cycle count = 3
                Loop below unrolled 8 times
                Loop below has 0 loads, 3 stores, 3 prefetches, 0 FPadds, 0
FPmuls, and 0 FPdivs per iteration
                     51.       setvalsmod();
```

関数 `setvalsmod()` にはループコードが含まれており、この関数はインライン化されているため、51 行目のコメントにはループコメントが含まれていることに注意してください。

上記の抜粋では、コンパイラのコメントが行の再度で折り返されています。パフォーマンスアナライザのソースウィンドウは幅の制限がないため、このようには表示されません。

ソースウィンドウ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、75 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。

共通部分式の除去

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し、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することができます。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、コンパイラは以前あった式の一方に基づいて、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、数種類のループ最適化を行うことができます。以下に一般的なものを示します。

- 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繰り返しを取り除き、これらをループの前か後に適宜移動します。

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って、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。

場合によっては、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、その後で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、以下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++) {
```

インライン化

インライン関数があると、コンパイラは、実際の関数呼び出しを行う代わりに、関数が呼び出された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。以下に、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);
```

パフォーマンスアナライザのソースウィンドウでは、コンパイラのコメントは折り返されず、2 行にまたがらないことに注意してください。

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とその理由を示します。以下に、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、129 ページの「並列実行とコンパイラ生成の本体関数」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他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、162 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

実行可能コードの各行のソースコードメトリックは、固定幅の欄に表示されます。メトリックは、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 102 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としきい値の強調表示も、「データ表示方法を設定」ダイアログボックスを使ってパフォーマンスアナライザで変更できます。詳細は、75 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 をソース行にマップ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にあたり、パフォーマンスアナライザは、最初に特定のオブジェクトモジュール (`.o` ファイル) 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 のデータを調べます。注釈付きソースを作成するには、パフォーマンスアナライザが、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プ状態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。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、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 `resetpath` 指令、あるいはアナライザ GUI によって変更できます。

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プ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の意味

命令のメトリックについては、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。こういった場合、ソース行のメトリックが異常に高く見えることがあります。コード内の他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付き止めてください。

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 種類のメトリックをまとめます。

表 7-1 注釈付きソースコードのメトリック

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、以下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.`	この行にあったことになっている PC がプログラムに存在しますが、その PC を参照するデータがありません。このことは、統計的に標本収集されたか、トレースされた呼び出しスタックに、そうした PC が存在しないことを意味します。`0.` メトリックは、ソース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です。このリストには、各命令のパフォーマンスメトリックが表示されます。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、行番号のマップ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はない場合は、単に指定された関数の命令が逆アセンブルされ、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は、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。
- ソースファイルが存在し、行番号データが記録されている場合は、パフォーマンスアナライザは表示方式によっては、ソースと逆アセンブリコードを交互に表示します。
- 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、それらのコメントも交互に表示されます。

逆アセンブリコードの各命令には、注釈として以下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、交互に表示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は、表 7-1 で示しているソースコードの注釈で説明しているとおりです。

複数の場所に #include で取り込まれているコードの逆アセンブリリストでは、コードの #include のたびに逆アセンブリ命令が 1 回繰り返されます。ソースコードは、逆アセンブリコードの繰り返しブロックがファイルに最初に示されたときのみ交互表示されます。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にのみ交互表示されます。ソース表示に切り替えると、逆アセンブリコードの毎回の繰り返しに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表示に表示される場合があります。ソース表示の場合とは異なり、これらのインデックス行を直接移動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。

他のファイルのコードを #include するファイルでは、ソースコードの交互表示なしで、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、交互表示の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交互表示できますが、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交互に表示されます。最適化されている場合は、後の命令が前の行よりも前に表示されることがあります。パフォーマンスアナライザの交互表示アルゴリズムでは、命令が行 *N* にあったと判断された場合は、常に、その行 *N* 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。ソースの行 *N* に対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コードを理解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ありません。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あたっては、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取り、実行する方法を理解しておいてください。

以下では、注釈付き逆アセンブリコードを理解するにあたってのいくつかの問題点を取り上げます。

命令発行時のグループ化

グループ単位で読み込まれて、命令は発行されます (命令発行グループ)。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、他の命令またはレジスタに対する依存関係によって異なります。このことは、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。またこのことは、呼び出しスタックが記録されたときに、「次」に実行する命令が複数存在する可能性があることも意味します。

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行うので、関連性がないと思える関数を変更すると命令の位置合わせが異なってく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 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。このことは、次のケースをはじめとして、いくつかの状況で起き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などのように、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。この種の遅延としては、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 プロセッサの場合、読み込み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項目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

TLB ミスは別として、オーバーフローで生成された割り込みの処理に時間を要する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の呼び出しスタックは、オーバーフローの発生時点ではなく、命令シーケンスの後の方で記録されます。サイクルおよび命令発行などのカウンタの場合、このこと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以外の命令がメトリックの原因とされます。イベントを引き起こした PC が記録対象 PC の少し前の命令に位置していることがよくあるため、こうした場合は、逆アセンブリリストで正しい命令を特定できます。ただし、この命令範囲内に分岐先がある場合、イベントを引き起こした PC に対応する命令を見分けるのは、ほとんど (または、まったく) 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の場合、コレクタはカウンタ名の前に「+」が付いている場合にイベントを発生させた PC を検索します。

「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、注釈付きソースウィンドウと逆アセンブリウィンドウで、特別なインデックス行として表示されます。ソースウィンドウでは、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
                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;
               62.              }
               63.         }
```

注釈付き逆アセンブリでは、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。

```
0.     0.          <Function: main -- outline code from line 85 [_$o1D85.main]>
0.     0.          [ 85] 100001034:  sethi       %hi(0x100000), %i5
0.     0.          [ 86] 100001038:  bset        4, %i3
0.     0.          [ 85] 10000103c:  or          %i5, 1, %l7
0.     0.          [ 85] 100001040:  sllx        %l7, 12, %l5
0.     0.          [ 85] 100001044:  call        printf ! 0x100101300
0.     0.          [ 85] 100001048:  add         %l5, 336, %o0
0.     0.          [ 90] 10000104c:  cmp         %i3, 0
0.     0.          [ 20] 100001050:  ba,a        0x1000010b4
0.     0.          <Function: main -- outline code from line 46 [_$o1A46.main]>
0.     0.          [ 46] 100001054:  mov         1, %i3
0.     0.          [ 47] 100001058:  ba          0x100001090
0.     0.          [ 56] 10000105c:  clr         [%i2]
0.     0.          <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。パフォーマンスアナライザは、読みやすい関数名を表示します。詳細は、139 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、パフォーマンス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を示します。詳細については、169 ページの「包括的メトリック」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、129 ページの「並列実行とコンパイラ生成の本体関数」で詳しく説明しています。

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、オリジナルソースにおける構造の先頭の行番号、および並列構造のシーケンス番号をエンコードする符号化名を本体関数に設定します。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、より完全な名前に復号化されて表示されます。

以下に、関数リストに表示される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には、特別なインデックス行が生成されます。

```
                <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
```

```
                  <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と相互に関連付けされません。このような場合は、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、以下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て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把握していません。パフォーマンスアナライザの「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、以下のように、`collector_func_load()` に渡される情報によって異なります。

- 情報が指定されていない場合、つまり `collector_func_load()` が呼び出されていない場合は、動的にコンパイルされて読み込まれた関数が、パフォーマンスアナライザの関数リストに `<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、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が関数リストに表示されます。注釈付きソースは利用できませんが、逆アセンブリ命令は表示することができます。ただし、行番号は、未知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て、行番号テーブルを提供しないと、パフォーマンスアナライザによって表示される情報は、ソースファイル名を指定しない場合と似ています。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。たとえば、次のような具合です。

```
1.121   1.121       <関数 func0、行番号なしの命令>
                 1. #include        <stdio.h>
```

- ソースファイル名と行番号のテーブルが提供されている場合は、関数とそのメトリックは、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、31 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。監視プロセス時、仮想マシンは、インタプリタを行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ができます。

以下の例に示されているように、Java の場合は、コレクタ API 関数を使用する必要はありません。パフォーマンスアナライザは、メソッドのインデックス行の下の特別な行を使って、注釈付き逆アセンブリリストにおける HotSpot™ がコンパイルしたコードの存在を示します。

```
                11.    public int add_int () {
                12.       int       x = 0;
                    <Function: Routine.add_int()>
2.832   2.832       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、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を表示します。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、以下のようになります。

```
                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、Java コードにより Java Native Interface (JNI) を介して呼び出される、元は C、C++、または Fortran で記述されたコンパイル済みコードです。以下の例は、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。こういった呼び出しの一例としては、渡される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。最適化できる呼び出しを見つけると、コンパイラは、この関数のコピー (クローンと呼ばれる) を作成し、最適化コードを生成します。

注釈付きソースでは、コンパイルのコメントは、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。以下に示されているように、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
```

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。これらはソースファイルとは関連付けられていないため、命令はソース行番号と関連付けられていません。以下は、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ています。

```
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、パフォーマンスアナライザは、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、その関数の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。パフォーマンスアナライザは、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がまとめて表示されることがあります。静的関数の例は、以下に示された `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、前記の関数は、以下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「`<library.so> -- no functions found + 0x0000F870`」のように表示される場合もあります。

包括的メトリック

注釈付き逆アセンブリでは、スレーブスレッドと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。

以下に、デモプログラム omptest から抜粋して、*<inclusive metrics for slave threads>* の例を示します。

```
               3.         subroutine pardo(n,m,a,b,c)
                  <Function: pardo_>
0.     0.        [ 3]    1d200:  save        %sp, -240, %sp
0.     0.        [ 3]    1d204:  ld          [%i0], %i5
0.     0.        [ 3]    1d208:  st          %i5, [%fp - 24]
               4.
               5.         real*8 a(n,n), b(n,n), c(n,n)
               6.
               7.         call initarray(n,a,b,c)
               8.
0.     12.679     <inclusive metrics for slave threads>
               9. c$omp  parallel shared(a,b,c,n) private(i,j,k)
0.     3.653      <inclusive metrics for slave threads>
              10. c$omp  do
```

以下に、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。

```
              43.         else
              44.         {
              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示される疑似行の <branch target> (分岐先) は、その有効アドレスを見つけるためのバックトラッキングアルゴリズムが分岐先内で実行されたためにバックトラッキングに失敗した命令の PC に対応します。

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 ユーティリティを使用し、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表示できます。メトリックが表示されないことを除けば、この表示は、パフォーマンス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 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、オブジェクトファイル (.o ファイル) のいずれかのファイル名です。

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、ソースファイルまたはオブジェクトファイルのファイル名です。*item* は、*functon'file'* の形でも指定できます。この場合は、er_src が、指定されたファイルのソースコンテキストに、指定の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、参照する関数を決定するためのインデックスです。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 tag の all -1 は、オブジェクトのすべての関数について注釈付きソースか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。

以下に、er_src ユーティリティに使用可能なオプションについて説明します。

-func

所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 *item tag*

出力リストに逆アセンブリコードを含めます。デフォルトでは、逆アセンブリコードは含まれません。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コードリストが生成されます。

-{c,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 オプションが使用されている場合はソースのコンパイラのコメントに、-dcc オプションが使用されている場合は逆アセンブリのコメントに、-c が使用されている場合はソースと逆アセンブリのコメントの両方に適用されます。これらのクラスについては、91 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。デフォルト値ファイルとしては、システム全体の er.rc ファイル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し、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します。これらのファイルは、パフォーマンスアナライザと er_print によっても使用されますが、er_src が使用するのは、ソースおよび逆アセンブリコード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については、102 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 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*

リストの出力先として、*filename* に指定したファイルを開きます。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 er_src のバージョン情報を表示します。

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リティについて説明します。

この章では、以下について説明します。

- 実験の操作
- その他のユーティリティ

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 などの通常の 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。このことは、Forte Developer 7 (Sun™ ONE Studio 7, Enterprise Edition for Solaris™ Operating System) 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、削除用のコマンドが用意されています。以下に、これらのユーティリティ er_cp(1)、er_mv(1)、er_rm(1) を説明します。

実験には、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 ユーティリティを使った実験のコピー

次の 2 つの形の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。コピー先に *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 つ目の形式の er-cp 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含まれている場合、er_cp 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 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コピーしません。

er_mv ユーティリティを使った実験の移動

次の 2 つの形の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。コピー先に *experiment2* が存在する場合、er_mv 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 er-mv コマンドは、リスト中の空白で区切られた一群の実験を指定された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含まれている場合、er_mv 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 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移動しません。

er_rm ユーティリティを使った実験の削除

リストに指定された実験または実験グループを削除します。実験グループを削除すると、そのグループに含まれるすべての実験が削除されてから、グループファイルも削除されます。

er_rm コマンドの構文は以下のとおりです。

```
er_rm [-f] [-V] experiment-list
```

-f オプションは、エラーメッセージの出力を禁止し、実験が見つかったかどうかに関係なく、コマンドが確実に正常終了するようにします。-V オプションは、er_rm 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。

その他のユーティリティ

ここでは、通常は使用する必要のないその他のユーティリティについて説明します。これらのユーティリティを使用する必要がある環境を示しながら、説明を行います。

er_archive ユーティリティ

er_archive コマンドの構文は以下のとおりです。

```
er_archive [-qAF] experiment
er_archive -V
```

er_archive は、実験が正常終了したとき、または実験に対してパフォーマンスアナライザや er_print コマンドを起動したときに、自動的に実行されます。このユーティリティは、実験で参照されている共有オブジェクトの一覧を読み取り、それぞれにアーカイブファイルを 1 つ作成します。これらの出力ファイルには、必ず、接頭辞 .archive が付き、その共有オブジェクトの関数とモジュールのマッピング情報が含まれます。

ターゲットプログラムが異常終了した場合、コレクタによって er_archive が実行されないことがあります。実験データが記録されたのとは別のマシン上で異常終了した実行セッションで得られた実験を調べるには、その実験に対し、データが記録されたマシン上で er_archive を実行する必要があります。実験データのコピー先のマシンでロードオブジェクトを使用できるようにするには、-A オプションを使用します。

この実行によって、実験で参照されているすべての共有オブジェクトに対するアーカイブファイルが作成されます。これらのアーカイブには、オブジェクトファイルとそのロードオブジェクト内のあらゆる関数のアドレス、サイズ、名前、ロードオブジェクトの絶対パス、その最終変更日時を示すタイムスタンプが含まれます。

er_archive を実行したときに共有オブジェクトが見つからないか、その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タイムスタンプと異なるか、または実験が記録されたのとは異なるマシンで er_archive が実行された場合、アーカイブファイルには警告メッセージが書き込まれます。er_archive が手動で実行された場合、警告は stderr にも出力されます (-q フラグが指定されていない場合)。

以下に、er_archive ユーティリティに使用可能なオプションについて説明します。

-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取り込まれ、パフォーマンスアナライザまたは er_print で表示されます。

-A

実験へのすべてのロードオブジェクトの書き込みを要求します。この引数を使用して、実験が記録されたマシン以外のマシンにさらに容易にコピーできる実験を作成することができます。

-F

アーカイブファイルを強制的に作成または再作成します。この引数を使用し、警告のあったファイルを作成し直すことができます。

-V

er_archive のバージョン番号情報を表示し、終了します。

er_export ユーティリティ

er_export コマンドの構文は以下のとおりです。

```
er_export [-V] experiment
```

er_export ユーティリティは、実験ファイル内の raw データを ASCII テキストに変換します。このファイルの形式と内容は変更されることがあるため、特定の目的にのみ利用できます。このファイルは、アナライザが実験ファイルを読み取れないときにだけ使用されることを意図しています。出力を見ることによって、ツールの開発者は raw データを理解し、問題を解析できます。V オプションは、バージョン番号を表示します。

付録 A

prof、gprof、tcov によるプログラムのプロファイル

この付録では、プログラムの実行時間を測定したり、解析対象となるパフォーマンスデータを取得したりするための標準的なユーティリティについて説明します。このマニュアルでは、これらのユーティリティを「従来のプロファイルツール」と呼びます。プロファイリングツール `prof` および `gprof` は、Solaris™ オペレーティングシステムに付属しています。`tcov` は、Sun のコンパイラおよびツールに付属しているコードカバレッジツールです。

注 – 実行回数 (関数の呼び出し回数、ソースコード行の実行回数) の追跡には、従来のプロファイルツールを利用してください。これに対し、コレクタおよびパフォーマンスアナライザを使用すると、プログラムが時間を消費する部分に関するより詳細で正確な情報を得ることができます。これらのツールの使用方法については、第 3 章およびオンラインヘルプを参照してください。

表 A-1 に、標準的なパフォーマンスプロファイルツールで得られる情報をまとめます。

表 A-1 パフォーマンスプロファイルツール

コマンド	出力
`prof` `prof`	各関数に制御が渡される正確な回数とともに、プログラムが使用する CPU 時間の統計プロファイルを生成します。
`gprof`	各関数に制御が渡される正確な回数と、プログラムの呼び出しグラフ内の、個々の呼び出し元と呼び出し先の間で制御が受け渡しされる回数とともに、プログラムが使用する CPU 時間の統計プロファイルを生成します。
`tcov` `tcov`	プログラム内の各文の正確な実行回数情報を生成します。

従来のプロファイルツールには、C 以外のプログラミング言語で記述されたモジュールに使用できないものがあります。言語に関する詳細は、各ツールに関する節を参照してください。

この付録では、以下の内容について説明します。

- prof によるプロファイルの生成
- gprof による呼び出しグラフプロファイルの生成
- tcov による文レベルの解析
- 拡張 tcov による文レベルの解析

prof によるプロファイルの生成

prof は、プログラムが使用する CPU 時間の統計プロファイルを生成し、プログラム内の各関数に制御が渡される回数をカウントします。gprof 呼び出しグラフプロファイルおよび tcov コードカバレージツールは、これとは別の種類またはより詳細な情報を提供するツールです。

prof を使用してプロファイルレポートを生成するには、以下の操作を行います。

1. **-p コンパイラオプションを指定してプログラムをコンパイルします。**
2. **プログラムを実行します。**

   プロファイルデータが mon.out というプロファイルファイルに書き込まれます。このファイルは、プログラムを実行するたびに上書きされます。
3. **prof を実行してプロファイルレポートを作成します。**

   prof コマンドの構文は以下のとおりです。

```
% prof program-name
```

program-name は実行可能ファイルの名前です。プロファイルレポートは stdout に出力されます。このレポートには、各関数に関する情報が次の見出しで 1 行に 1 つ表示されます。

- %Time 総 CPU 時間に対して、この関数が消費する時間の割合
- Seconds この関数が占める総 CPU 時間
- Cumsecs この関数およびその前に示されている関数が占める秒数の総計
- #Calls この関数が呼び出される回数
- msecs/call この関数が呼び出されたときに消費される平均時間 (ミリ秒単位)
- Name 関数の名前

以下は、prof の使用例です。

```
% cc -p -o index.assist index.assist.c
% index.assist
% prof index.assist
```

以下は、prof のプロファイルレポート例です。

%Time	Seconds	Cumsecs	#Calls	msecs/call	名前
19.4	3.28	3.28	11962	0.27	compare_strings
15.6	2.64	5.92	32731	0.08	_strlen
12.6	2.14	8.06	4579	0.47	__doprnt
10.5	1.78	9.84			mcount
9.9	1.68	11.52	6849	0.25	_get_field
5.3	0.90	12.42	762	1.18	_fgets
4.7	0.80	13.22	19715	0.04	_strcmp
4.0	0.67	13.89	5329	0.13	_malloc
3.4	0.57	14.46	11152	0.05	_insert_index_entry
3.1	0.53	14.99	11152	0.05	_compare_entry
2.5	0.42	15.41	1289	0.33	lmodt
0.9	0.16	15.57	761	0.21	_get_index_terms
0.9	0.16	15.73	3805	0.04	_strcpy
0.8	0.14	15.87	6849	0.02	_skip_space
0.7	0.12	15.99	13	9.23	_read
0.7	0.12	16.11	1289	0.09	ldivt
0.6	0.10	16.21	1405	0.07	_print_index

.
.
.

（以降の出力は重要ではありません）

このプロファイルレポートでは、compare_strings() 関数にもっとも実行時間が費やされていることがわかります。2 番目に多く費やしているのが _strlen() です。このプログラムの実行効率を高めるには、総 CPU 時間の 20% 近くを費やしている compare_strings() に注目し、アルゴリズムを改良するか呼び出し回数を減らします。

prof のプロファイルレポートからは、compare_strings() が頻繁に再帰を繰り返す関数であることはわかりませんが、180 ページの「gprof による呼び出しグラフプロファイルの生成」で説明する呼び出しグラフプロファイルを利用することで、再帰回数を減らすことができます。また、この例の場合は、アルゴリズムを改良することによって呼び出し回数を減らすこともできます。

注 – Solaris™ 7 および 8 オペレーティングシステムでは、複数の CPU を使用するプログラムについても、正確な CPU 時間のプロファイルを得られますが、呼び出し回数がロックされないため、関数の呼び出し回数の精度に影響が出ることがあります。

---

gprof による呼び出しグラフプロファイルの生成

prof の表形式のプロファイルによっても、パフォーマンス向上のための有用な情報を得ることができますが、呼び出しグラフプロファイルを利用すると、さらに詳細な解析情報を得ることができます。呼び出しグラフプロファイルは、モジュール間の呼び出し関係を示すリストです。場合によっては、呼び出しを完全に削除することで、パフォーマンスが向上することもあります。

注 – gprof では、呼び出し元と呼び出し先の間で制御が受け渡された回数に比例して、関数内で費やされた時間が呼び出し元に帰せられます。ただし、あらゆる呼び出しがパフォーマンス的に等価であるわけではないため、こうした動作は誤った前提になる可能性があります。developers.sun.com にあるパフォーマンス解析のチュートリアルの例を参照してください。

---

prof と同様に、gprof も、プログラムが使用する CPU 時間の統計プロファイルを生成し、関数に制御が渡される回数をカウントします。gprof はまた、プログラムの呼び出しグラフ内の、個々のアークの間で制御が受け渡される回数もカウントします。**アーク**は、呼び出し元と呼び出し先の組です。

注 – Solaris™ 7 および 8 オペレーティングシステムでは、複数の CPU を使用するプログラムについても、正確な CPU 時間のプロファイルを得られますが、呼び出し回数がロックされないため、関数の呼び出し回数の精度に影響が出ることがあります。

---

gprof を使用してプロファイルレポートを生成するには、以下のようにします。

1. **適切なコンパイラオプションを指定してプログラムをコンパイルします。**
   - C プログラムの場合は、-xpg オプションを使用します。
   - Fortran プログラムの場合は、-pg オプションを使用します。
2. **プログラムを実行します。**

   プロファイルデータは、gmon.out というプロファイルファイルに書き込まれます。このファイルは、プログラムを実行するたびに上書きされます。
3. **gprof を実行してプロファイルレポートを作成します。**

   prof コマンドの構文は以下のとおりです。

```
% gprof program-name
```

program-name は実行可能ファイルの名前です。プロファイルレポートは標準出力に出力されます (このレポートは大きくなることがあります)。このレポートは、次の 2 つの項目から構成されます。

- 全体の呼び出しグラフプロファイル - プログラム内のすべての関数の呼び出し元と呼び出し先に関する情報です。この形式については、この後の例を参照してください。
- 「表」形式のプロファイル - prof コマンドの概要情報に似た形式のプロファイルです。

gprof のプロファイルレポートには、概要の各部の意味に関する説明が含まれています。また、次の例に示すように、標本収集の精度が示されます。

```
granularity: each sample hit covers 4 byte(s) for 0.07% of 14.74
seconds
```

上記の「4 byte(s)」は、1 つの命令に対する精度を意味しています。この例では「0.07% of 14.74 seconds」は、CPU の 10 ミリ秒単位で表現されていて、実行の 0.07% を占めることを意味します。

以下は gprof の使用例です。

```
% cc -xpg -o index.assist index.assist.c
% index.assist
% gprof index.assist > g.output
```

一部ですが、gprof によって作成される呼び出しグラフプロファイルは以下のようになります。

```
                                called/total
                                parents
index   %time    self    descendants   called+self       name                    index
                                called/total
                                children
-----------------------------------------------
                0.00    14.47         1/1               start                   [1]
[2]     98.2    0.00    14.47         1                 _main                   [2]
                0.59    5.70          760/760           _insert_index_entry     [3]
                0.02    3.16          1/1               _print_index            [6]
                0.20    1.91          761/761           _get_index_terms        [11]
                0.94    0.06          762/762           _fgets                  [13]
                0.06    0.62          761/761           _get_page_number        [18]
                0.10    0.46          761/761           _get_page_type          [22]
                0.09    0.23          761/761           _skip_start             [24]
                0.04    0.23          761/761           _get_index_type         [26]
                0.07    0.00          761/820           _insert_page_entry      [34]
-----------------------------------------------
                                      10392             _insert_index_entry     [3]
                0.59    5.70          760/760           _main                   [2]
[3]     42.6    0.59    5.70          760+10392         _insert_index_entry     [3]
                0.53    5.13          11152/11152       _compare_entry          [4]
                0.02    0.01          59/112            _free                   [38]
                0.00    0.00          59/820            _insert_page_entry      [34]
                                      10392             _insert_index_entry     [3]
-----------------------------------------------
```

この例の index.assist プログラムに対する入力ファイルには、761 行のデータが含まれています。このため、次のように結論付けることができます。

- fgets() は 762 回呼び出されます。fgets() の最後の呼び出しでは、ファイルの終わりが返されます。
- insert_index_entry() 関数は、main() から 760 回呼び出されます。

- `insert_index_entry()` 関数は、`main()` からの 760 回の呼び出しのほかに、自身を 10,392 回呼び出します。`insert_index_entry()` は非常に再帰的なものです。
- `compare_entry()` は `insert_index_entry()` から呼び出されますが、11,152 回呼び出されます。この回数は、760+10,392 回と同じです。`insert_index_entry()` から呼び出される `compare_entry()` は 11,152 (760+10,392) 回呼び出されます。つまり、`insert_index_entry()` が呼び出されるたびに、`compare_entry()` が 1 回呼び出されるということで、これは正しい呼び出し回数です。呼び出し回数に矛盾がある場合は、プログラム論理に何らかの問題があると考えられます。
- `insert_page_entry()` は、合計で 820 回呼び出されます。820 回の内訳は、プログラムがインデックスノードを構築している間の `main()` からの呼び出しが 761 回、`insert_index_entry()` からの呼び出しが 59 回です。この呼び出し回数は、重複するインデックスエントリが 59 個あることを示しており、このため、それらのページ番号エントリはインデックスノードと連結されて、1 つのチェーンになります。重複しているインデックスエントリはその後解放され、`free()` に対する呼び出し 59 回が発生します。

tcov による文レベルの解析

tcov ユーティリティは、プログラムがコードセグメントを実行する頻度に関する情報を出力します。このユーティリティは、実行頻度が注釈として付いた、ソースファイルのコピーを出力します。コードの注釈には、基本ブロックレベルとソース行レベルの 2 種類があります。基本ブロックは、分岐のない、ソースコードの線形セグメントです。基本ブロック内の文は同じ回数だけ実行されるので、基本ブロックの実行回数がわかれば、基本ブロック内の各文の実行回数がわかります。tcov ユーティリティは、時間ベースのデータを出力しません。

注 – tcov は、C および C++ プログラムで使用できますが、`#line` または `#file` 指令を含むファイルには使用できません。また、`#include` ヘッダーファイル内のコードのテストカバレージ解析もサポートしていません。

---

tcov を使用して注釈付きソースコードを作成するには、以下のようにします。

1. **適切なコンパイラオプションを指定してプログラムをコンパイルします。**
   - C プログラムの場合は、`-xa` オプションを使用します。
   - Fortran または C++ プログラムの場合は、`-a` オプションを使用します。

-a または -xa オプションを使用してコンパイルを行った場合は、リンクでもそのオプションを使用する必要があります。コンパイラは、オブジェクトファイルごとに .d という接尾辞を持つカバレージデータファイルを作成します。これらのコードカバレージファイルは、環境変数 TCOVDIR の示すディレクトリに作成されます。TCOVDIR が設定されていない場合は、現在のディレクトリに作成されます。

注 – -xa (C コンパイラの場合) や -a (C 以外のコンパイラの場合) オプションを指定したコンパイルで作成されたプログラムは、通常よりも実行速度が遅くなります。これは、実行のたびに .d ファイルが更新され、このためにかなりの時間を要するためです。

---

2. **プログラムを実行します。**

   プログラムが終了すると、カバレージデータファイルが更新されます。

3. **tcov を実行して注釈付きのソースコードを生成します。**

   tcov コマンドの構文は以下のとおりです。

```
% tcov options source-file-list
```

   source-file-list はソースファイル名のリストです。tcov のオプションについては、tcov(1) のマニュアルページを参照してください。tcov は、一群のファイルを出力します。これらのファイルの接尾辞はデフォルトでは .tcov ですが、-o *filename* オプションを使用して変更できます。

   コードカバレージ解析用のコンパイルで作成されたプログラムは、入力を変更しながら、繰り返し実行できます。つまり、プログラムに tcov を繰り返し使用し、動作を比較できます。

   以下は tcov の使用例です。

```
% cc -xa -o index.assist index.assist.c
% index.assist
% tcov index.assist.c
```

次に示す C コードのリストは、index.assist を構成するあるモジュールからの抜粋です。この部分は、再帰的に呼び出される insert_index_entry 関数を表しています。C コードの左側の数値は、各文が実行された回数を示しています。insert_index_entry() 関数は、main() から 11,152 回呼び出されています。

```
                struct index_entry *
11152       ->  insert_index_entry(node, entry)
                struct index_entry *node;
                struct index_entry *entry;
                {
                    int result;
                    int level;

                    result = compare_entry(node, entry);
                    if (result == 0) {     /* exact match */
                                           /* Place the page entry for the
duplicate */
                                           /* into the list of pages for this node
*/
59          ->              insert_page_entry(node, entry->page_entry);
                            free(entry);
                            return(node);
                    }

11093       ->      if (result > 0)        /* node greater than new entry -- */
                                               /* move to lesser nodes */
3956        ->              if (node->lesser != NULL)
3626        ->                      insert_index_entry(node->lesser, entry);
                            else {
330         ->                      node->lesser = entry;
                                    return (node->lesser);
                            }
                        else               /* node less than new entry -- */
                                           /* move to greater nodes */
7137        ->              if (node->greater != NULL)
6766        ->                      insert_index_entry(node->greater, entry);
                            else {
371         ->                      node->greater = entry;
                                    return (node->greater);
                            }
                }
```

tcov は、注釈付きコードリストの末尾に以下のような概要情報を追加します。もっとも頻繁に実行される基本ブロックの統計が、実行頻度の順に表示されます。行番号は、ブロックの先頭行の番号です。

以下は、index.assist プログラムの概要です。

```
              Top 10 Blocks

           Line        Count
           240         21563
           241         21563
           245         21563
           251         21563
           250         21400
           244         21299
           255         20612
           257         16805
           123         12021
           124         11962

77         Basic blocks in this file
55         Basic blocks executed
71.43      Percent of the file executed

           439144     Total basic block executions
           5703.17    Average executions per basic block
```

tcov プロファイル用の共有ライブラリの作成

tcov によるプロファイル用に共有可能なライブラリを生成し、バイナリファイルにすでにリンクされているライブラリの代わりに使用することができます。共有可能なライブラリを生成するときは、次の例に示すように、-xa オプション (C コンパイラの場合) か -a オプション (C 以外のコンパイラの場合) を使用します。

```
% cc -G -xa -o foo.so.1 foo.o
```

このコマンドによって、共有可能なライブラリに tcov プロファイル関数のコピーが取り込まれるため、ライブラリのクライアントの再リンクが不要になります。ライブラリのクライアントをプロファイル用にリンクした場合は、共有可能なライブラリのプロファイルに、そのクライアントが使用するバージョンの tcov 関数が使用されます。

ファイルのロック

tcov は、.d ファイルのブロックカバレージデータベースを更新するときに、簡単なファイルロックメカニズムを使用します。具体的には、tcov.lock という 1 つのファイルを使用してファイルをロックします。このファイルロックによって、-xa (C の場合) または -a (C 以外のコンパイラの場合) を使用したコンパイルで作成された実行可能ファイルは、同じシステムで一度に 1 つしか動作しないようになります。-xa または -a オプションを使用したコンパイルで作成されたプログラムを手動で終了した場合は、tcov.lock ファイルを手動で削除する必要があります。

-xa または -a オプションを使用してコンパイルされたファイルは、プログラムが tcov によるプロファイル用にリンクされると、自動的にプロファイルツール関数を呼び出します。プログラムの終了時にこれらの関数は、たとえばファイル xyz.f に関して実行時に収集された情報と、ファイル xyz.d に格納されていた既存のプロファイル情報を結合します。プロファイル済みのバイナリを複数のユーザーが同時に実行することによって、このファイルが壊れないようにするために、更新期間中、xyz.d に xyz.d.lock というロックファイルが作成されます。xyz.d またはそのロックファイルを開くか、読み取るときにエラーが発生するか、実行時の情報と既存の情報の間に矛盾がある場合、xyz.d に格納されているデータは変更されません。

xyz.f を編集して再コンパイルすると、xyz.d 内のカウンタの個数が変わることがあります。これは、プロファイル済みのバイナリを実行したときに検出されます。

プロファイル済みのバイナリを実行するユーザーが多すぎると、一部のユーザーがロックを取得できないことがあります。数秒の遅延があると、エラーメッセージが表示されます。格納されている情報は更新されません。このロックは、ネットワーク全体に機能します。また、ロックはファイル単位に行われるため、ほかのファイルが更新されなくなることはありません。

プロファイル関数は、アクセス不可能となっていた自動マウントファイルシステムにアクセスしようと試みます。ただし、カバレージデータファイルを含むファイルシステムがマシンごとに異なる名前でマウントされていたり、プロファイル済みのバイナリを実行しているユーザーがカバレージデータファイルや、そのファイルが含まれるディレクトリに対する書き込み権を持っていない場合、この試みは失敗します。関係するすべてのディレクトリ名を統一し、バイナリを実行する可能性のあるユーザー全員が、それらのディレクトリに書き込みできるようにしてください。

tcov 実行時関数によって報告されるエラー

ここでは、tcov 実行時関数が報告するエラーメッセージをまとめます。

- カバレージデータファイルに対する読み取りまたは書き込み権がありません。この問題は、カバレージデータファイルを含むディレクトリが削除されている場合にも発生します。

```
tcov_exit: Could not open coverage data file 'coverage-data-file-name'
because 'system-error-message-string'.
```

- カバレージデータファイルを含むディレクトリに対する書き込み権がありません。この問題は、バイナリを実行するマシンに、カバレージデータファイルを含むディレクトリがマウントされていない場合にも発生します。

```
tcov_exit: Could not write coverage data file 'coverage-data-file-name'
because 'system-error-message-string'.
```

- 多くのユーザーが同時にカバレージデータファイルを更新しようとしています。この問題は、カバレージデータファイルの更新中にマシンがクラッシュした場合にも発生します。この場合、ロックファイルは削除されずに残ります。クラッシュが発生した場合は、2 つのファイルのうちのサイズの大きい方を、クラッシュ後のカバレージデータファイルとして使用してください。ロックファイルは手動で削除してください。

```
tcov_exit: Failed to create lock file 'lock-file-name' for coverage
data file 'coverage-data-file-name' after 5 tries. Is someone else
running this executable?
```

- 利用可能なメモリーがなく、標準入出力パッケージが動作できません。この場合は、カバレージデータファイルを更新できません。

```
tcov_exit: Stdio failure, probably no memory left.
```

- ロックファイル名の長さがカバレージデータファイル名より 6 文字長くなっています。生成されたロックファイル名が無効である可能性があります。

```
tcov_exit: Coverage data file path name too long (length
characters) 'coverage-data-file-name'.
```

- tcov によるプロファイルが有効なライブラリまたはバイナリが、同時に実行、編集、再コンパイルされようとしています。古いバイナリは、カバレージデータファイルが特定の決まったサイズであると予測しますが、編集することによってそのサイズがしばしば変わることがあります。古いバイナリが、古いカバレージデータファイルを更新しようとしているときに、コンパイラが新しいカバレージデータファイルを作成すると、バイナリによって、カバレージファイルは空白または壊れていると報告されることがあります。

```
tcov_exit: Coverage data file 'coverage-data-file-name' is too short.
Is it out of date?
```

拡張 tcov による文レベルの解析

オリジナルの tcov 同様、拡張 tcov は、プログラムの動作に関する行単位の情報を提供します。具体的には、ソースファイルのコピーを作成し、使用される行とその行が使用されている回数を示す注釈を付加します。拡張 tcov は、基本的なブロックに関する概要情報も提供し、C および C++ 両方のソースファイルで使用することができます。

拡張 tcov では、オリジナル tcov にあった欠点の一部が解消されています。拡張 tcov で改善された機能は、以下のとおりです。

- C++ に対するサポートの強化
- #include ヘッダーファイルに含まれるコードのサポートと、テンプレートクラスおよび関数のカバレージ番号があいまいになっていた問題の修正
- オリジナルの tcov の実行時ルーチンからの実行効率の向上
- コンパイラがサポートしているすべてのプラットフォームのサポート

拡張 tcov を使用して注釈付きソースコードを作成するには、以下のようにします。

1. **-xprofile=tcov コンパイラオプションを指定し、プログラムをコンパイルします。**

   tcov と異なり、拡張 tcov はコンパイル時にファイルを生成しません。

2. **プログラムを実行します。**

   プロファイルデータを格納するためのディレクトリが作成され、そのディレクトリに tcovd というカバレージデータファイルが作成されます。デフォルトでは、このディレクトリは、プログラム (*program-name*) が実行されたディレクトリ内に作成され、*program-name*.profile という名前が付けられます。このディレクトリは、プロファイルバケツとも呼ばれます。これらのデフォルト値は、環境変数を使用して変更できます (191 ページの「tcov 関係のディレクトリと環境変数」を参照)。

3. tcov を実行して注釈付きのソースコードを生成します。

tcov コマンドの構文は以下のとおりです。

```
% tcov option-list source-file-list
```

source-file-list はソースコードファイル名のリスト、*option-list* はオプションのリストです (tcov のオプションについては、tcov(1) のマニュアルページを参照)。拡張 tcov による処理を有効にするには、必ず -x オプションを指定する必要があります。

拡張 tcov は、一群の注釈付きソースファイルを出力します。デフォルトでは、それらファイルには、対応するソースファイル命名に .tcov を付加した名前が割り当てられます。

以下に、拡張 tcov の使用例を示します。

```
% cc -xprofile=tcov -o index.assist index.assist.c
% index.assist
% tcov -x index.assist.profile index.assist.c
```

拡張 tcov の出力は、オリジナルの tcov の出力と同じです。

拡張 tcov プロファイル用の共有ライブラリの作成

拡張 tcov プロファイル用の共有ライブラリは、次の例に示すように、-xprofile=tcov コンパイラオプションを使用することによって作成できます。

```
% cc -G -xprofile=tcov -o foo.so.1 foo.o
```

ファイルのロック

拡張 tcov は、ブロックカバレージデータファイルを更新するときに、簡単なファイルロックメカニズムを使用します。具体的には、tcovd ファイルと同じディレクトリに作成された 1 つのファイルを使用してファイルをロックします。このファイル名は tcovd.temp.lock です。カバレージ解析用にコンパイルしたプログラムを手動で終了した場合は、ロックファイルを手動で削除する必要があります。

このロック方法では、ロックの競合がある場合、指数的バックオフが行われます。tcov 実行時ルーチンがロックの取得を試み、続けて 5 回失敗した場合、tcov は終了し、その実行用のデータは失われます。この場合は、以下のメッセージが表示されます。

```
tcov_exit: temp file exists, is someone else running this
executable?
```

tcov 関係のディレクトリと環境変数

tcov 用にプログラムをコンパイルして実行すると、そのプログラムによってプロファイルバケツが作成されます。すでにプロファイルバケツが存在する場合は、そのプロファイルバケツが使用されます。プロファイルバケツが存在しない場合は、新しく作成されます。

プロファイルバケツは、プロファイル出力が生成されるディレクトリを示します。プロファイル出力の名前と格納場所はデフォルト値によって制御されますが、環境変数で変更できます。

注 – tcov は、プロファイルフィードバック情報の収集に使用されるコンパイラオプション -xprofile=collect と -xprofile=use が使用するのと同じデフォルト値と環境変数を使用します。これらのコンパイラオプションについての詳細は、ご使用のコンパイラのマニュアルを参照してください。

プログラムが生成するデフォルトのプロファイルバケツには、実行可能ファイル名に拡張子 .profile を付加した名前が付けられ、実行可能ファイルが実行されたディレクトリに作成されます。このため、たとえば /home/userdir から、/usr/bin/xyz というプログラムを実行した場合、デフォルトでは、/home/userdir 内に xyz.profile という名前のプロファイルバケツが生成されます。

UNIX® プロセスは、プログラムの実行中に現在の作業用ディレクトリを変更できます。このため、プロファイルバケツの生成に使用される現在の作業用ディレクトリは、プログラム終了時の現在の作業用ディレクトリになります。ごくまれに、プログラムがその動作中に現在の作業用ディレクトリを変更することがありますが、その場合は、環境変数を使用し、プロファイルバケツが生成される場所を制御することができます。

デフォルト値は、以下の環境変数を設定することで変更できます。

- SUN_PROFDATA

実行時のプロファイルバケツの名前を指定します。SUN_PROFDATA_DIR も設定されている場合は、常にこの変数の値が SUN_PROFDATA_DIR の値に付加されます。この設定は、実行可能ファイル名が argv[0] の値と等しくない場合などに役立ちます (たとえば、異なる名前のシンボリックリンクから実行可能ファイルを起動した場合など)。

- SUN_PROFDATA_DIR

  プロファイルバケツがあるディレクトリの名前を指定します。この変数は、実行時および tcov コマンドによって使用されます。

- TCOVDIR

  下位互換性を維持するための、SUN_PROFDATA_DIR と同じ働きをする環境変数です。TCOVDIR と SUN_PROFDATA_DIR の両方が設定されている場合、TCOVDIR の設定は無視されます。また、この場合は、プロファイルバケツが生成されるときに、警告が表示されます。

  TCOVDIR は、実行時および tcov コマンドによって使用されます。

索引

E

F

G

J

L

M

N

O

P

S

T

X

あ

い

さ

し

す

せ

と

に

ね

は

ひ

ふ

へ

ほ

ら

り

る

ろ