プログラムのパフォーマンス解析

Forte Developer 7

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 U.S.A.
650-960-1300

Part No. 816-4917-10
2002 年 6 月 , Revision A

原典： *Program Performance Analysis Tools*
Part No: 816-2458-10
Revision A

目次

図目次

表目次

はじめに

このマニュアルでは、Forte™ Developer 7 製品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™ オペレーティング環境、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 章は、コレクタとパフォーマンスアナライザによって、4 つのサンプルプログラムのパフォーマンスを評価する方法を、具体例を使用してチュートリアル形式で説明しています。

第 3 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 4 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 5 章では、パフォーマンスアナライザのグラフィカルインタフェースの機能を説明しています。 注： パフォーマンスアナライザを使用するには、ライセンスが必要です。

第 6 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 7 章では、標本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 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)	#

Forte Developer の開発ツールとマニュアルページへのアクセス

Forte Developer の製品コンポーネントとマニュアルページは、標準の /usr/bin/ と /usr/share/man の各ディレクトリにインストールされません。Forte Developer のコンパイラとツールにアクセスするには、PATH 環境変数に Forte Developer コンポーネントディレクトリを必要とします。Forte Developer マニュアルページにアクセスするには、PATH 環境変数に Forte Developer マニュアルページディレクトリが必要です。

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 Forte Developer 製品が /opt ディレクトリにインストールされていることを想定しています。Forte Developer 製品が /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。

Forte Developer 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 Forte Developer 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には

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. 出力内容から /opt/SUNWspro/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 Forte Developer 開発ツールにアクセスできるように設定されています。パスがない場合は、次の指示に従って、PATH 環境変数を設定してください。

▼ PATH 環境変数を設定して Forte Developer の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. 次のパスを PATH 環境変数に追加します。

   /opt/SUNWspro/bin

Forte Developer 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 Forte Developer 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。

▼ MANPATH 環境変数を設定する必要があるかどうか判断するには

1. 次のように入力して、dbx マニュアルページを表示します。

```
% man dbx
```

2. 出力された場合、内容を確認します。

   dbx(1) 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、次の節の指示に従って MANPATH 環境変数を設定してください。

▼ MANPATH 変数を設定して Forte Developer 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. 次のパスを PATH 環境変数に追加します。

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

Forte Developer マニュアルへのアクセス

Forte Developer の製品マニュアルには、以下からアクセスできます。

- 製品マニュアルは、以下の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。

  ```
  /opt/SUNWspro/docs/ja/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』

インターネットの docs.sun.com Web サイト (http://docs.sun.com) から、サンのマニュアルを読んだり、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。

アクセスできる製品マニュアル

Forte Developer 7 製品マニュアルは、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。アクセス可能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。製品のソフトウェアが /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	アクセス可能な形式と格納場所
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 インストール製品について 場所：`file:/opt/SUNWspro/docs/ja/index.html` のマニュアル索引
Readme およびマニュアルページ	形式：HTML インストール製品について 場所： `file:/opt/SUNWspro/docs/ja/index.html` のマニュアル索引
リリースノート	製品 CD 内の HTML ファイル

関連する Forte Developer マニュアル

以下の表は、`file:/opt/SUNWspro/docs/ja/index.html` 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
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ます。
Fortran プログラミングガイド	Solaris オペレーティング環境での効果的な Fortran コードの記述方法について説明します (入出力、ライブラリ、パフォーマンス、プログラム解析、並列処理など)。
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ます。
(言語ごとのユーザーズガイド)	コンパイルとコンパイラオプションについて説明します。

関連する Solaris マニュアル

ここでは、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	内容の説明
Solaris 8 Reference Manual Collection	マニュアルページの節を参照。	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 8 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 8 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。

ご意見の送付先

米国 Sun Microsystems, Inc. では、マニュアルの向上に力を注いでおり、ユーザーのご意見やご提案をお待ちしております。ご意見などがありましたら、次のアドレスまで電子メールをお送りください。

`docfeedback@sun.com`

第1章

プログラムパフォーマンス解析ツールの概要

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。

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。

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、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™ プログラムに関するプロファイリング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 3 章、コレクタの詳細については第 4 章を参照してください。 コレクタは、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。 メトリックは、タイミングメトリック、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の 5 つのグループに分類されます。 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も行

います。 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改善するために「マップファイル」を作成することもできます。 パフォーマンスアナライザの詳細については第 5 章、コマンド行解析ツール `er_print` については第 6 章を参照してください。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)。

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。

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、LWP、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。 この他、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、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の操作は、マウスとキーボードのどちらを使用しても行えます。

`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。

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環境 (SPARC™ プラットフォーム版) に組み込まれています。

- `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。

注 - パフォーマンスアナライザの GUI と IDE は、バージョン 8 および 9 の Solaris オペレーティング環境用 Forte™ for Java™ 4, Enterprise Edition の一部です。

第2章

パフォーマンスツールの使用法の習得

この章では、コレクタとパフォーマンスアナライザの使用方法をチュートリアルを利用して説明します。チュートリアルの主な目的は、次の 3 つです。

- パフォーマンス問題の簡単な例とその確認方法を紹介する。
- パフォーマンスアナライザの機能について説明する。
- パフォーマンスアナライザがパフォーマンスデータをどのように表示し、各種のコード要素をどのように取り扱うかを説明する。

注 - パフォーマンスアナライザの GUI と IDE は、バージョン 8 および 9 の Solaris™ オペレーティング環境用 Forte™ for Java™ 4, Enterprise Edition の一部です。

以下の 4 つのプログラム例を通じて、いくつかの異なる状況におけるパフォーマンスアナライザの機能を具体的に紹介します。

- 例 1 :基本的なパフォーマンス解析。この例では、タイミングデータによってパフォーマンス問題を確認する方法を紹介し、関数、ソース行、および命令が時間とどのように対応しているか、また再帰呼び出し、オブジェクトモジュールの動的読み込み、および派生プロセスをパフォーマンスアナライザがどのように取り扱うかについて説明します。 この例では、アナライザのメインディスプレイである「関数」タブ、「呼び出し元-呼び出し先」タブ、「ソース」タブ、「逆アセンブリ」タブ、および「タイムライン」タブを使用します。 このサンプルプログラム `synprog` は、C で書かれています。
- 例 2 :OpenMP による並列化戦略。この例では、Fortran プログラム `omptest` を Open MP 指令によって並列化するさまざまな方法の効率を紹介します。
- 例 3 :マルチスレッドプログラムにおけるロック戦略。この例では、複数のスレッドに対して仕事をスケジューリングするためのさまざまな方法の効率や、データ管理がキャッシュパフォーマンスに及ぼす効果を、同期遅延データを使用して紹介し

ます。 この例では、明示的にマルチスレッド化されている C プログラム mttest を使用しています。このプログラムは、クライアント/サーバアプリケーションモデルです。

- 例 4 :キャッシュの動作と最適化。この例では、Fortran 90 プログラム cachetest において、メモリーアクセスとコンパイラの最適化が実行速度に及ぼす効果を紹介します。この例では、パフォーマンス解析におけるハードウェアカウンタデータおよびコンパイラのコメントの使用例についても取り上げます。

注 - この章で示すデータは、実際のサンプルプログラムを実行したときに表示されるデータと異なることがあります。

このチュートリアルで扱うパフォーマンスデータについては、コマンド行で使用する収集方法だけを示します。ほとんどの例の場合、IDE を使用してパフォーマンスデータを使用することもできます。 IDE からデータを収集するには、dbx デバッガと「デバッグ」メニューの「パフォーマンスツールキット」サブメニューを使用します。

サンプルプログラムの実行準備

この節では、Forte™ Developer 7 製品が /opt ディレクトリにインストールされていることを前提にしています。製品が/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。

各サンプルプログラムのソースコードとメークファイルは、パフォーマンスアナライザの次のサンプル用ディレクトリに格納されています。

```
/opt/SUNWspro/examples/analyzer
```

このディレクトリには、サンプルごとにサブディレクトリが存在し、それぞれ synprog、omptest、mttest、cachetest という名前になっています。

以下の手順に従い、デフォルトのオプションを使用してサンプルプログラムをコンパイルしてください。

1. Forte Developer ソフトウェアのディレクトリ /opt/SUNWspro/bin がパスに含まれていることを確認します。

2. 次のコマンドを使用し、使用するサンプルがあるサンプル用サブディレクトリを自分の作業用ディレクトリにコピーします。

```
% mkdir work-directory
% cp -r /opt/SUNWspro/examples/analyzer/example  work-directory/example
```

example には、上記の実際のサンプル用サブディレクトリ名のいずれかを指定してください。このチュートリアルでは、自分のディレクトリを上記のコードのように設定していると仮定しています。

3. `make` を使用し、サンプルプログラムをコンパイルしてリンクします。

```
% cd work-directory/example
% make
```

使用システム条件

サンプルプログラムを実行するには、それぞれ次の条件が満たされている必要があります。

- `synprog` - 1 つの CPU 上で実行すること。
- `omptest` - 少なくとも 4 つの CPU を搭載した SPARC™ ハードウェアで実行すること。
- `mttest` - 少なくとも 4 つの CPU を搭載したマシンで実行すること。 Solaris 7 または 8 のオペレーティング環境と標準スレッドライブラリを使用してテストを行ってうください。 Solaris 8 オペレーティング環境の代替スレッドライブラリや Solaris 9 オペレーティング環境のスレッドライブラリを使用した場合には、例の細部が多少異なってきます。
- `cachetest` - 少なくとも 160M バイトのメモリーを搭載した UltraSPARC™ III ハードウェアで実行すること。

デフォルトのコンパイラオプションの変更

サンプルプログラムに特定の動作をさせるために、コンパイラオプションはデフォルトの設定になっています。命令セットアーキテクチャーを選択する `-xarch` など一部のオプションは、プログラムのパフォーマンスに影響を及ぼす可能性があります。 使

用するコンピュータに最適な命令セットが使用されるようにするには、-xarch オプションを native に設定します。 別の設定にする場合は、メークファイル内の ARCH 環境変数の設定を変更してください。

デフォルトの V7 アーキテクチャーの SPARC プラットフォームの場合、コンパイラは、整数の乗算命令や除算命令を使用するのではなく、libc.so から .mul ルーチンと .div ルーチンを呼び出すコードを生成します。 これらの算術演算に要した時間は、<未知>関数に示されます。詳細は、172 ページの「<未知>関数」 を参照してください。

これら 3 つのサンプルプログラム用の各メークファイルには、コメントの形式で OFLAGS 環境変数の別のコンパイラオプションの設定の組み合わせが含まれています。デフォルトの設定でサンプルプログラムを実行した場合は、別の設定の組み合わせの 1 つを使用してプログラムをコンパイル、リンクし、コンパイラによるコードの最適化と並列化にどのような影響があるのかを調べてみてください。 OFLAGS のコンパイラオプションについては、『C ユーザーズガイド』または『Fortran ユーザーズガイド』を参照してください。

パフォーマンスアナライザの基本機能

この節では、パフォーマンスアナライザの基本機能について説明します。

パフォーマンスアナライザを起動すると、「関数」タブが表示されます。 コレクタでデフォルトのデータオプションが使用されていた場合は、このとき、「関数」タブにデフォルトの時間ベースのプロファイルメトリックの入った関数リストが表示されます。

- 排他的ユーザー CPU 時間 (Exclusive user CPU time) - 関数自体に費やされた秒単位の時間
- 包括的ユーザー CPU 時間 (Inclusive user CPU time) - 関数自体とその関数が呼び出した別の関数に費やされた秒単位の時間

デフォルトでは、関数一覧は、排他的ユーザー CPU 時間でソートされます。 メトリックについての詳細は、59 ページの「プログラム構造へのメトリックの対応付け」を参照してください。

「関数」タブで関数を選択して「呼び出し元-呼び出し先」タブをクリックすると、その関数の呼び出し元と呼び出し先に関する情報が表示されます。 このタブには、横長の次の 3 つの区画があります。

- 中央の区画 - 選択された関数のデータを表示します。
- 上の区画 - 選択された関数を呼び出すすべての関数のデータを表示します。
- 下の区画 - 選択された関数が呼び出すすべての関数のデータを表示します。

この「呼び出し元-呼び出し先」タブには、排他的メトリックと包括的メトリックの他に、呼び出し元と呼び出し先の属性メトリックも表示されます。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、呼び出し元から呼び出し先への呼び出しに関係する部分です。

「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、そのソースコードが各コード行に対応するパフォーマンスメトリックとともに表示されます。「逆アセンブリ」タブには、選択された関数の命令が、各命令に対応するパフォーマンスメトリックとともに表示されます。

「タイムライン」タブには、各実験の大域タイミングデータとコレクタによって記録された各イベントのデータが表示されます。データは、各実験のデータタイプと LWP ごとに表示されます。

例 1 :基本的なパフォーマンス解析

この例では、プログラミングに関係する次の 4 つの観点からパフォーマンスアナライザの主要機能の具体的な使用例を紹介します。

- 10 ページの「単純なメトリック解析」: 関数リスト、注釈付きソースコードリスト、および注釈付き逆アセンブリコードリストを使用し、2 つのルーチンの簡単なパフォーマンス解析を行うことによって、型変換のコストを調べます。
- 14 ページの「メトリックの対応と gprof の誤った推論」:「呼び出し元-呼び出し先」タブを使用し、下位レベルのルーチンで費やされる時間に呼び出し元がどのように関わっているのかを明らかにします。 `gprof` は、プログラムが CPU 時間の大

部分を費やしている関数を正しく発見する標準的な UNIX パフォーマンスツールですが、この例では、その CPU 時間の大半の原因になっている呼び出し元を誤って報告します。 gprof については、付録 A を参照してください。

- 17 ページの「再帰の効果」: 直接および間接両方の再帰関数呼び出しの再帰シーケンスにおいて、呼び出し元がどのように時間に関わっているのかを明らかにします。
- 21 ページの「動的にリンクされた共有オブジェクトの読み込み」: ロードオブジェクトの扱い方を示し、読み込まれる場所とタイミングが変わっても、関数が正しく特定される理由を明らかにします。

synprog に関するデータの収集

この節の手順に進む前に、6 ページの「サンプルプログラムの実行準備」 と 8 ページの「パフォーマンスアナライザの基本機能」 を参照してください。 この例を開始する前に、synprog をコンパイルします。

コマンド行から synprog のデータを収集し、パフォーマンスアナライザを起動するには、以下のようにコマンドを入力します。

```
% cd work-directory/synprog
% make collect
% analyzer test.1.er &
```

これで、以降の節の手順に従って synprog 実験データを解析する準備ができました。

単純なメトリック解析

この節では、cputime() および icputime() という 2 つの関数の CPU 時間を調べます。 どちらの関数にも、変数 x を 1 ずつインクリメントする for ループが含まれています。 ただし、x は、cputime() では浮動小数点型の変数ですが、icputime() では整数型の変数です。

1. 「関数」タブで `cputime()` および `icputime()` を見つけます。

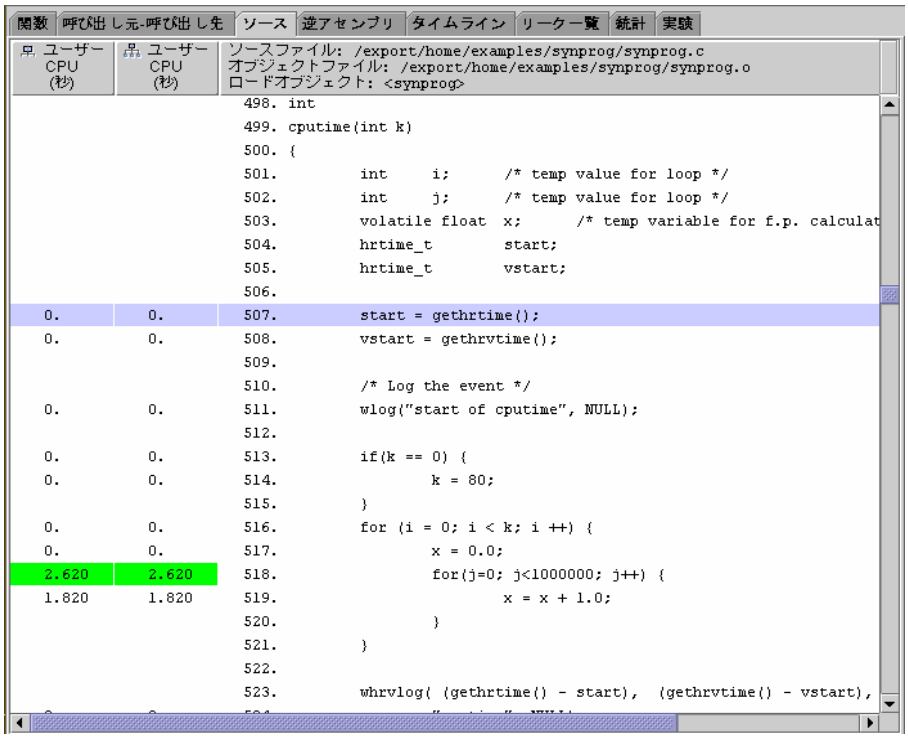
   表示をスクロールする代わりに「検索」ツールを使用して関数を見つけることもできます。

   これら 2 つの関数の包括的ユーザー CPU 時間を比較します。 `icputime()` よりもはるかに多くの時間が `cputime()` で消費されています。

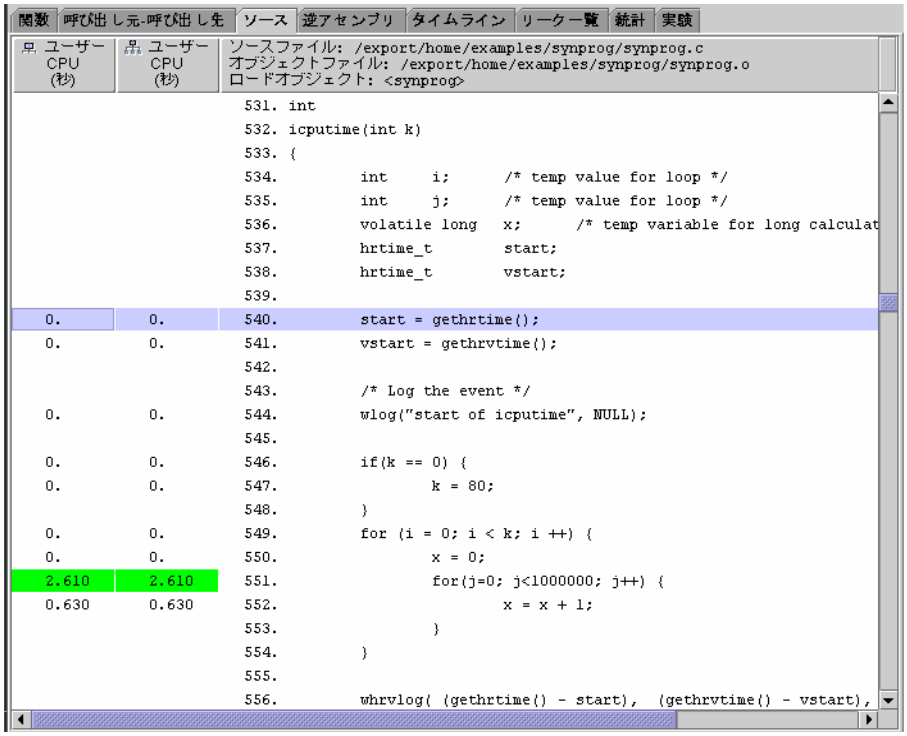
2. 「ファイル」 >「新規ウィンドウを作成」を選択します。

   同じデータを持つアナライザウィンドウが新たに表示されます。両方のウィンドウが見られるようにウィンドウを配置します。

3. 第 1 ウィンドウの「関数」タブで `cputime()` をクリックして選択し、「ソース」タブをクリックします。



4. 第 2 ウィンドウの「関数」タブで icputime() をクリックして選択し、「ソース」タブをクリックします。



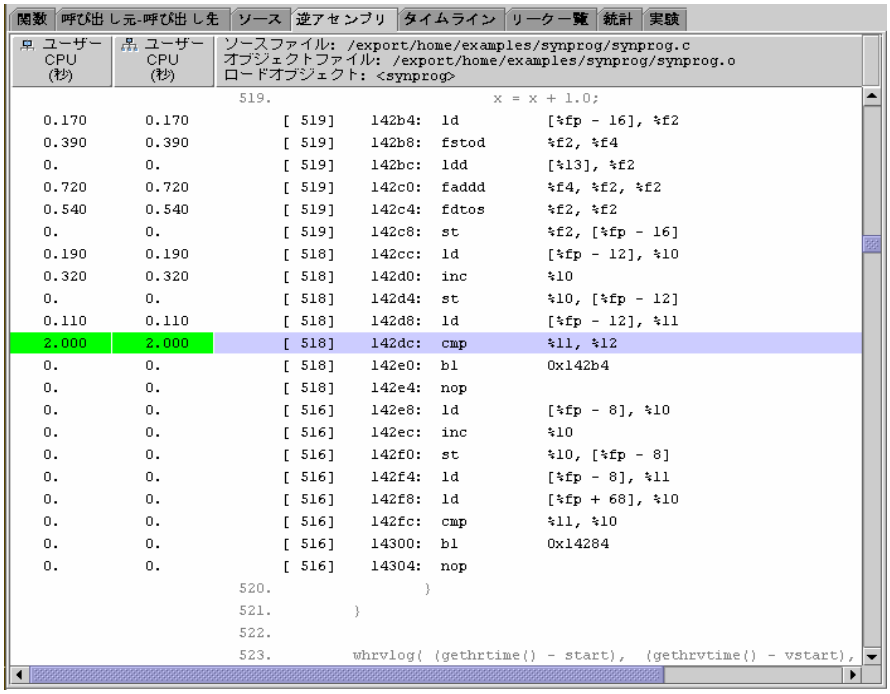
注釈付きソースコードリストを見ると、この CPU 時間の原因になっているコード行が分かります。 どちらの関数においても、実行時間の大部分がループ行と、x をインクリメントする行で費やされています。

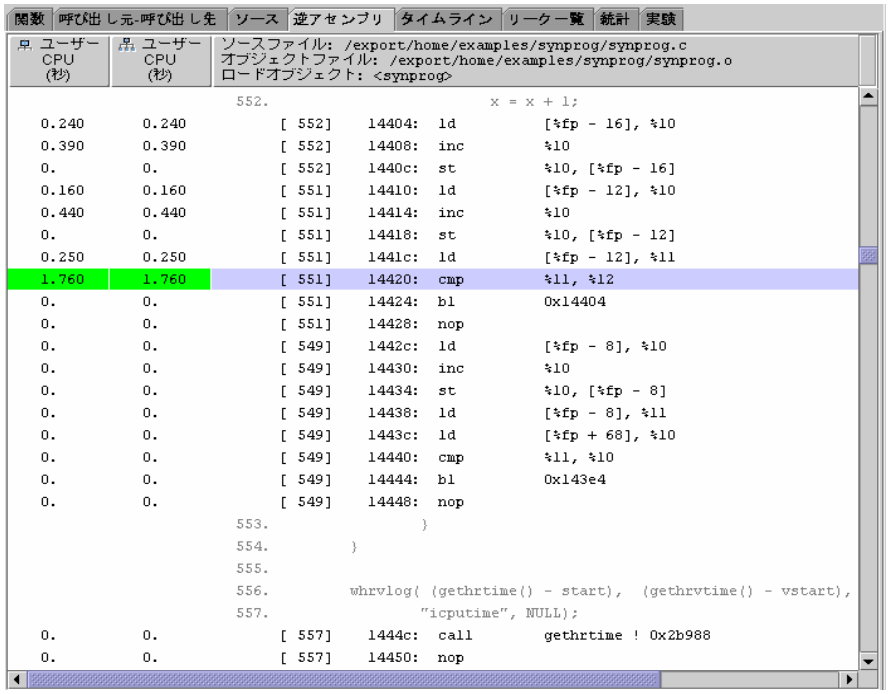
icputime() のループ行で費やされる時間は、cputime() のループ行で費やされる時間とほぼ同じですが、x をインクリメントする行の実行は、icputime() の方が cputime() よりもはるかに少ない時間で行われます。

5. 両方のウィンドウで「逆アセンブリ」タブをクリックし、x をインクリメントするソースコード行の命令を見つけます。

「検索」ツールのコンボボックスで「高メトリック値」を選択して検索すると、これらの命令を見つけることができます。

1 つの命令に対応する時間は、その命令が実行するまで待機していた時間であり、命令の実行に費やされた時間ではありません。


関数
呼び出し元-呼び出し先
ソース
逆アセンブリ
タイムライン
リーク一覧
統計
実験
ユーザー CPU (秒)
ユーザー CPU (秒)
ソースファイル: /export/home/examples/synprog/synprog.c
オブジェクトファイル: /export/home/examples/synprog/synprog.o
ロードオブジェクト: <synprog>
519. x = x + 1.0;
0.170 0.170 [519] 142b4: ld [%fp - 16], %f2
0.390 0.390 [519] 142b8: fstod %f2, %f4
0. 0. [519] 142bc: ldd [%l3], %f2
0.720 0.720 [519] 142c0: faddd %f4, %f2, %f2
0.540 0.540 [519] 142c4: fdtos %f2, %f2
0. 0. [519] 142c8: st %f2, [%fp - 16]
0.190 0.190 [518] 142cc: ld [%fp - 12], %l0
0.320 0.320 [518] 142d0: inc %l0
0. 0. [518] 142d4: st %l0, [%fp - 12]
0.110 0.110 [518] 142d8: ld [%fp - 12], %l1
2.000 2.000 [518] 142dc: cmp %l1, %l2
0. 0. [518] 142e0: bl 0x142b4
0. 0. [518] 142e4: nop
0. 0. [516] 142e8: ld [%fp - 8], %l0
0. 0. [516] 142ec: inc %l0
0. 0. [516] 142f0: st %l0, [%fp - 8]
0. 0. [516] 142f4: ld [%fp - 8], %l1
0. 0. [516] 142f8: ld [%fp + 68], %l0
0. 0. [516] 142fc: cmp %l1, %l0
0. 0. [516] 14300: bl 0x14284
0. 0. [516] 14304: nop
520. }
521. }
522.
523. whrvlog((gethrtime() - start), (gethrvtime() - vstart),


関数
呼び出し元-呼び出し先
ソース
逆アセンブリ
タイムライン
リーク一覧
統計
実験
ユーザー CPU (秒)
ユーザー CPU (秒)
ソースファイル: /export/home/examples/synprog/synprog.c
オブジェクトファイル: /export/home/examples/synprog/synprog.o
ロードオブジェクト: <synprog>
552. x = x + 1;
0.240 0.240 [552] 14404: ld [%fp - 16], %l0
0.390 0.390 [552] 14408: inc %l0
0. 0. [552] 1440c: st %l0, [%fp - 16]
0.160 0.160 [551] 14410: ld [%fp - 12], %l0
0.440 0.440 [551] 14414: inc %l0
0. 0. [551] 14418: st %l0, [%fp - 12]
0.250 0.250 [551] 1441c: ld [%fp - 12], %l1
1.760 1.760 [551] 14420: cmp %l1, %l2
0. 0. [551] 14424: bl 0x14404
0. 0. [551] 14428: nop
0. 0. [549] 1442c: ld [%fp - 8], %l0
0. 0. [549] 14430: inc %l0
0. 0. [549] 14434: st %l0, [%fp - 8]
0. 0. [549] 14438: ld [%fp - 8], %l1
0. 0. [549] 1443c: ld [%fp + 68], %l0
0. 0. [549] 14440: cmp %l1, %l0
0. 0. [549] 14444: bl 0x143e4
0. 0. [549] 14448: nop
553. }
554. }
555.
556. whrvlog((gethrtime() - start), (gethrvtime() - vstart),
557. "icputime", NULL);
0. 0. [557] 1444c: call gethrtime ! 0x2b988
0. 0. [557] 14450: nop

cputime() には、1 を x に追加するために実行しなければならない命令が 6 個あります。 倍精度浮動小数点定数である 1.0 を読み込んで x に追加するために、相当な時間が費やされます。 fdtos 命令と fstod 命令が x の値を単精度浮動小数点値から倍精度浮動小数点値に変換して再び変換しなおし、1.0 をfaddd 命令によって追加します。

一方、icputime() には読み込み、加算、格納の 3 つの命令しかありません。 変換が不要であるため、この 3 つの命令に要する時間は、cputime() 内の対応する命令セットに要する時間の約 3 分の 1 です。 ここでは、値 1 をレジスタにロードする必要はありません。 値 1 は、1 つの命令で直接 x に加算できます。

6. 演習を終了したら、アナライザウィンドウを閉じてください。

単純なメトリック解析の追加演習

テキストエディタで synprog のソースコードを開き、cputime() 内の x を double に変更してください。時間にどのような影響があるでしょうか。注釈付き逆アセンブリリストで違いを確認してください。

メトリックの対応と gprof の誤った推論

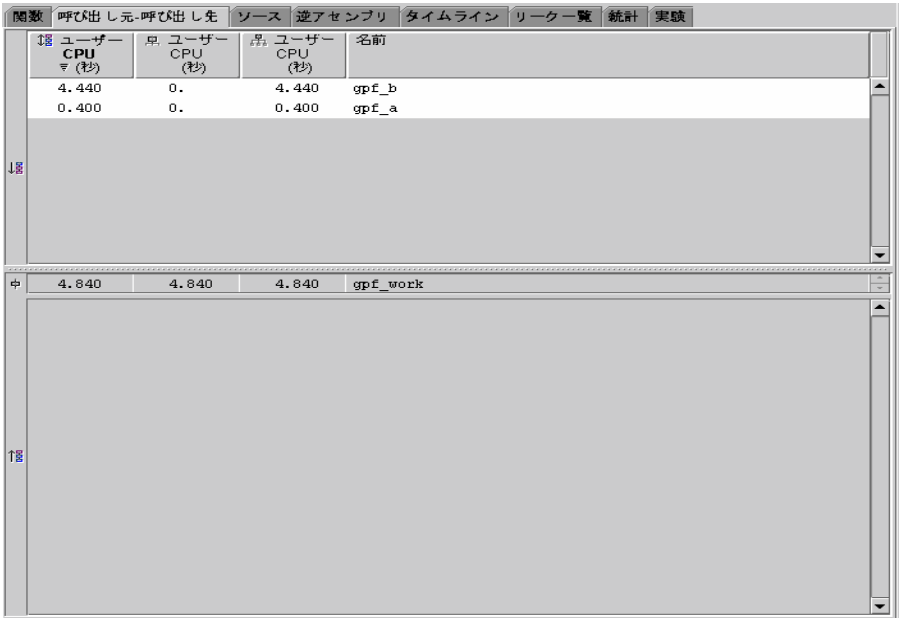
この節では、関数の呼び出し元がどのように実行時間に関わっているのかを調べ、パフォーマンスアナライザと gprof のその判定の仕方を比較します。

1. 「関数」タブで gpf_work() を選択し、「呼び出し元-呼び出し先」をクリックします。

「呼び出し元-呼び出し先」タブは、3 つの区画に分割されています。中央区画には、選択された関数が表示されます。上の区画には選択された関数の呼び出し元が表示され、下の区画には選択された関数によって呼び出される関数、つまり呼び出し先が表示されます。 このタブについては、109 ページの「「呼び出し元-呼び出し先」タブ」で説明します。 また、8 ページの「パフォーマンスアナライザの基本機能」 でも説明しています。

呼び出し元区画には、選択した関数を呼び出す 2 つの関数、gpf_b() および gpf_a() が示されます。 gpf_work() は他の関数を呼び出さないため、呼び出し先区画は空です。このような関数を「リーフ関数」と呼びます。

呼び出し元区画で属性ユーザー CPU 時間を見てみます。 gpf_work() に費やされている時間の大半は、gpf_b() からの呼び出しが原因であることが分かります。 gpf_a() からの呼び出しが原因の時間はわずかです。



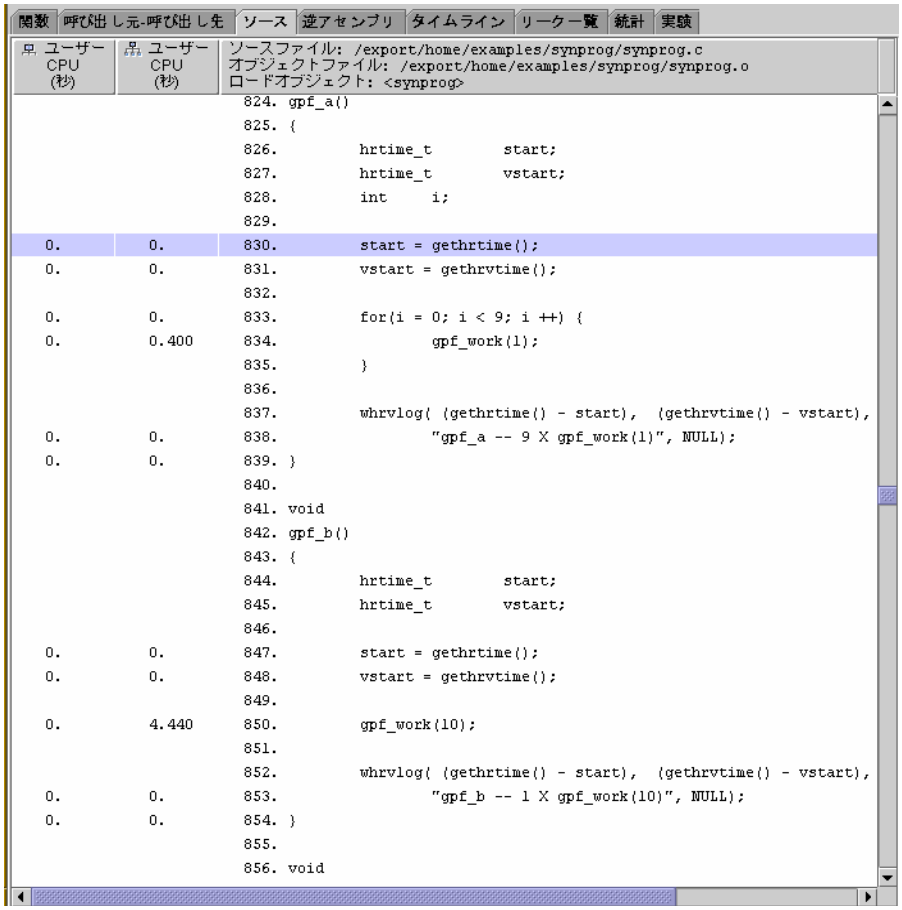
gpf_work において、gpf_b() からの呼び出しが、gpf_a() からの呼び出しよりも 10 倍以上も長い時間を要する理由を調べるには、これらの呼び出し元のソースコードを見る必要があります。

2. 呼び出し元区画の gpf_a() をクリックします。

   gpf_a() 関数が選択状態になり、中央の区画に移動します。 そして、gpf_a() の呼び出し元が呼び出し元区画、gpf_work() が呼び出し先区画に表示されます。

3. 「ソース」タブをクリックし、gpf_a() および gpf_b() の両方のコードが表示されるように下方向にスクロールします。

   gpf_a() が引数 1 で gpf_work() を 10 回呼び出しているのに対し、gpf_b() は、gpf_work() を 1 回しか呼び出していません。 ただし、引数は 10 です。 gpf_a() と gpf_b() の引数は、gpf_work() 内の仮引数 amt に渡されます。

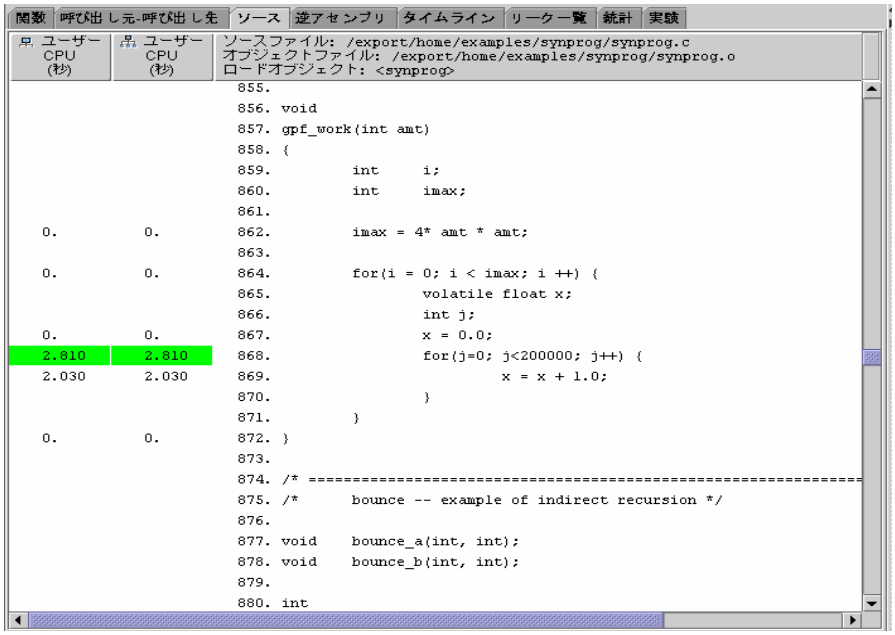


次に、gpf_work() のコードを表示し、gpf_work() の呼び出し方法によって違いが生じる理由を調べてみます。

4. 下方向にスクロールして gpf_work() のソースコードを見ます。

   変数 imax の計算式がある行を見てください。 imax は、その後の for ループに対する上限値を示します。 つまり、gpf_work() に費やされる時間は、amt 引数の 2 乗に依存することになります。このため、引数が 10 の関数からの 1 回の呼び出し (繰り返し回数が 400 回) は、引数が 1 の関数からの 10 回の呼び出し (4 回の繰り返しが 10 回) より約 10 倍の時間がかかります。

ただし、gprof では、関数に費やされる時間は、その時間が関数の引数、またはその関数がアクセスする他のデータにどのように依存しているかに関係なく、関数が呼び出される回数に基づいて概算されます。 このため、gprof を使用した synprog の解析では、gpf_a() からの呼び出しに gpf_b() からの呼び出しの 10 倍の時間がかかるという、誤った結論がもたらされます。 これが、gprof の誤った推論です。



再帰の効果

この節では、再帰シーケンスにおいてパフォーマンスアナライザが関数にメトリックを割り当てる方法を明らかにします。 コレクタによるデータの収集では、あらゆる関数呼び出しが記録されますが、解析では、特定の関数のすべてのインスタンスに関するメトリックが集計されます。 synprog プログラムには、2 つの再帰呼び出しシーケンス例が含まれています。

- recurse() 関数は、直接的な再帰の例です。 この関数は real_recurse() を呼び出します。real_recurse() は、テスト条件が満たされるまで自身を呼び出します。 そして、テスト条件が満たされると、ユーザー CPU 時間を必要とする処理を行います。 こうして、real_recurse() に対する呼び出しが繰り返され、最終的に recurse() に制御が戻されます。

- bounce() 関数は、間接的な再帰の例です。 この関数は、テスト条件が満たされているかどうかを検査する bounce_a() 関数を呼び出します。 条件が満たされていない場合、bounce() は bounce_b() を呼び出し、呼び出された bounce_b() は bounce_a() を呼び出します。 このシーケンスは、bounce_a() 内のテスト条件が満たされるまで繰り返され、 条件が満たされると、bounce_a() は ユーザー CPU 時間を必要とする処理を行います。こうして、bounce_b() および bounce_a() に対する呼び出しが繰り返された後、最終的に bounce() に制御が戻されます。

どちらの場合も、排他的メトリックは実際の仕事が行われた関数だけに属します。つまり、この例では、real_recurse() と bounce_a() です。これらのメトリックは、最終的な関数を呼び出すすべての関数に、包括的メトリックとして渡されます。

ここでは最初に、recurse() と real_recurse() のメトリックを見てみます。

1. 「関数」タブで recurse() を見つけて選択します。

   関数リストをスクロールする代わりに、「検索」ツールを使用して目的の関数を探すこともできます。

   recurse() 関数には、包括的ユーザー CPU 時間が示されますが、その排他的ユーザー CPU 時間はゼロになっています。 これは、recurse() が real_recurse() を 1 回呼び出すことしか行わないためです。

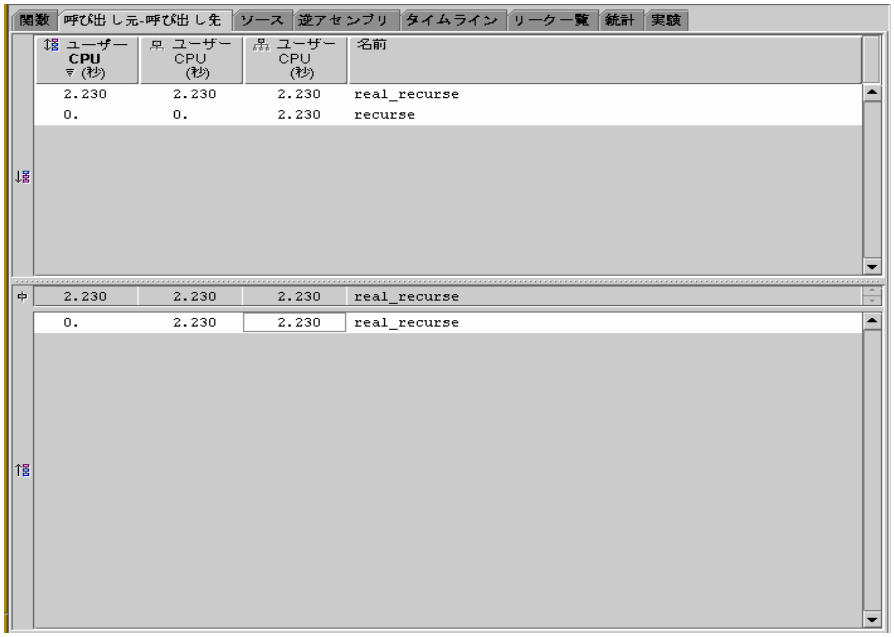
注 - プロファイルは統計的な性質をもつものであるため、synprog に対する実験で recurse() 関数のプロファイルイベントがいくつか記録され、その結果 recurse() の排他的 CPU 時間値が少なくなることがあります。しかし、こういったイベントが対応する排他的時間は、包括的な時間に比べるとごくわずかです。

---

2. 「呼び出し元-呼び出し先」タブをクリックします。

   選択された関数 recurse() が中央区画に表示されます。 recurse() によって呼び出された関数 real_recurse() が下の区画に表示されます。この区画のことを、「呼び出し先」区画と呼びます。

3. real_recurse() をクリックします。



real_recurse() に関する情報が「呼び出し元-呼び出し先」タブに表示されます。

- recurse() および real_recurse() はともに、real_recurse() の呼び出し元として呼び出し元区画 (上の区画) に表示されます。 このことは、recurse() が real_recurse() を呼び出した後に、real_recurse() が自身を再帰的に呼び出すことから見当がつきます。
- real_recurse() は自分自身を呼び出すので、呼び出し先区画に表示されます。
- real_recurse() には、排他的メトリックと包括的メトリックの両方が表示されます。 実際のユーザー CPU 時間が費やされるのは、この real_recurse() においてです。 この排他的メトリックは、その上の recurse() の包括的メトリックに加算されます。

次に、間接再帰シーケンスがどのようなものか見てみます。

1. 「関数」タブで bounce() を見つけて選択します。

   bounce() 関数には、包括的ユーザー CPU 時間が示されていますが、その排他的ユーザー CPU 時間はゼロになっています。 これは、bounce() が行う処理が bounce_a() を呼び出すことだけであるためです。

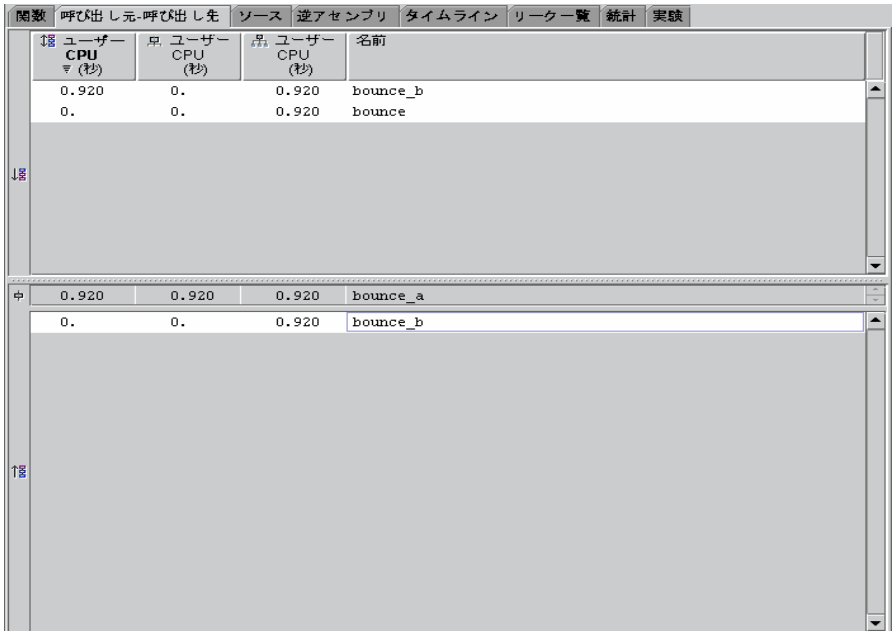
2. 「呼び出し元-呼び出し先」タブをクリックします。

   「呼び出し元-呼び出し先」タブが表示され、bounce() が 関数 bounce_a()だけを呼び出していることが分かります。

3. bounce_a() をクリックします。

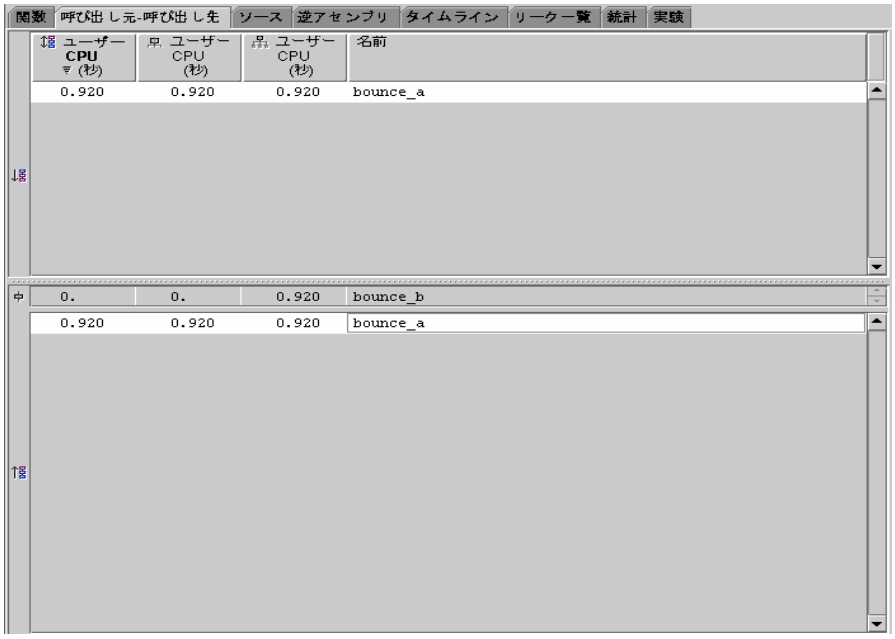
   「呼び出し元-呼び出し先」タブに bounce_a() に関する情報が表示されます。

   - bounce_a() の呼び出し元として、bounce() と bounce_b() の両方が呼び出し元区画に表示されます。
   - bounce_b() は bounce_a() によって呼び出されるので、呼び出し先区画にも表示されます。
   - bounce_a() には、排他的メトリックと包括的メトリックの両方が表示されます。実際のユーザー CPU 時間が費やされるのは、この bounce_a() においてです。これらのメトリックは、bounce_a() を呼び出す関数の包括的メトリックにも加算されます。



4. bounce_b() をクリックします。

   「呼び出し元-呼び出し先」ウィンドウに bounce_b() に関する情報が表示されます。 bounce_a() は、呼び出し元区画と呼び出し先区画の両方に表示されます。



動的にリンクされた共有オブジェクトの読み込み

この節では、共有オブジェクトに関する情報をパフォーマンスアナライザがどのように表示し、読み込まれる場所とタイミングが異なることがある、動的にリンクされた共有オブジェクトを構成する関数の呼び出しを、パフォーマンスアナライザがどのように処理するのかを明らかにします。

synprog ディレクトリには、動的にリンクされた共有オブジェクトが 2 つ 含まれています (so_syn.so と so_syx.so)。 実行中に、synprog は最初に so_syn.so を読み込み、そこに含まれる関数の 1 つである so_burncpu() を呼び出します。 そして so_syn.so の読み込みを解除し、同じアドレス位置に so_syx.so を読み込んで、so_syx.so に含まれる関数の 1 つである sx_burncpu() を呼び出します。 この so_syx.so は読み込み解除されず、再度 so_syn.so が別のアドレス位置に読み込まれ、so_burncpu() が呼び出されます。 so_syn.so が読み込まれるアドレス位置が異なるのは、最初に読み込まれたアドレス位置が別の共有オブジェクトによってまだ使用されているためです。

ソースコードを見ると分かるように、関数 so_burncpu() および sx_burncpu() はまったく同じ処理を行います。このため、これら 2 つの関数の実行に費やされるユーザー CPU 時間は同じであるはずです。

共有オブジェクトの読み込み先アドレスは、実行時に決定され、実行時ローダーがオブジェクトの読み込み先を選択します。

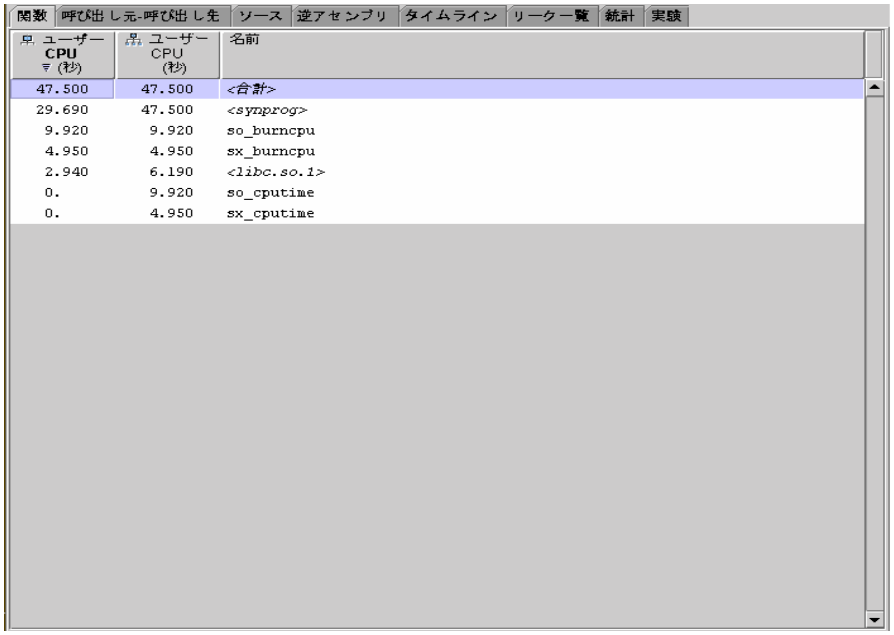
この例では、プログラムの実行中、同じ関数であっても、呼び出されるアドレスとタイミングが異なることがあること、異なる関数が同じアドレスに呼び出されることがあること、また、パフォーマンスアナライザがこのような動作を正しく処理し、関数がどのアドレスにあるかに関係なく、その関数に関するデータを集計することを明らかにします。

1. 「関数」タブをクリックします。
2. 「表示」>「関数の表示/非表示」を選択します。

   プログラムが実行時に使用したすべてのロードオブジェクトが「関数を表示/非表示」ダイアログに表示されます。
3. 「すべてを選択解除」をクリックし、`so_syx.so` と `so_syn.so` を選択して「適用」をクリックします。

   選択されたロードオブジェクト以外のオブジェクトの関数は、関数リストに表示されなくなります。そのエントリは、ロードオブジェクト全体を示す 1 つのエントリに置き換えられます。

   「関数」タブに表示されるロードオブジェクトリストには、メトリックが記録されるロードオブジェクトだけが入っているので、「関数を表示/非表示」ダイアログに表示されるリストよりも短い可能性があります。
4. 「関数」タブにおいて、`sx_burncpu()` と `so_burncpu()` のメトリックを調べます。



so_burncpu() は、sx_burncpu() と同じ処理を行います。 so_burncpu() は 2 回実行されたため、so_burncpu() のユーザー CPU 時間は sx_burncpu() のユーザー CPU 時間のほぼ 2 倍です。このように、パフォーマンスアナライザは、プログラムの実行中、現れるアドレスが異なっていても、同じ関数であることを認識し、その関数に関するデータを集計します。

例 2 :OpenMP による並列化戦略

Fortran プログラムの omptest は、OpenMP の並列化機能を使用し、次の 2 つの事例について並列化戦略の効率性をテストします。

- 最初の事例では、2 つの配列が別の配列から更新されるコード部分における PARALLEL SECTIONS 指令と PARALLEL DO 指令の使用を比較します。この事例では、複数のスレッドに対する作業負荷の平衡化を扱います。
- 第 2 の事例では、配列要素を合計してスカラー結果を求めるコード部分における CRITICAL SECTION 指令と REDUCTION 指令の使用を比較します。この事例では、メモリーへのアクセスをめぐるスレッド間の競合の影響を扱います。

並列化戦略と OpenMP 指令については、『Fortran プログラミングガイド』を参照してください。OpenMP 指令を検出した場合、コンパイラは特殊な関数とスレッドライブラリへの呼び出しを生成します。それらの関数は、パフォーマンスアナライザに表示されます。 詳細は、160 ページの「並列実行とコンパイラ生成の本体関数」 および 171 ページの「コンパイラ生成の本体関数」 を参照してください。注釈付きのソースおよび逆アセンブリコードのリストには、コンパイラが行った処理に関するメッセージが表示されます。

omptest に関するデータの収集

この節の手順に進む前に、6 ページの「サンプルプログラムの実行準備」 と 8 ページの「パフォーマンスアナライザの基本機能」 を参照してください。 この例を開始する前に、omptest をコンパイルします。

この例では、4 つの CPU での実行用の実験と、2 つの CPU での実行用の実験を作成します。 実験には、CPU の数を示すラベルが付きます。

C シェルのコマンド行から omptest のデータを収集するには、以下のようにコマンドを入力します。

```
% cd ~/work-directory/omptest
% setenv PARALLEL 4
% collect -o omptest.4.er omptest
% setenv PARALLEL 2
% collect -o omptest.2.er omptest
% unsetenv PARALLEL
```

Bourne シェルまたは Korn シェルを使用している場合は、以下のようにコマンドを入力します。

```
$ cd ~/work-directory/omptest
$ PARALLEL=4; export PARALLEL
$ collect -o omptest.4.er omptest
$ PARALLEL=2; export PARALLEL
$ collect -o omptest.2.er omptest
$ unset PARALLEL
```

両方の実験を対象としてパフォーマンスアナライザを起動するには、以下のように入力します。

```
$ analyzer omptest.4.er &
$ analyzer omptest.2.er &
```

これで、以降の節の手順に従って omptest 実験データを解析する準備ができました。

PARALLEL SECTIONS と PARALLEL DO 戦略の比較

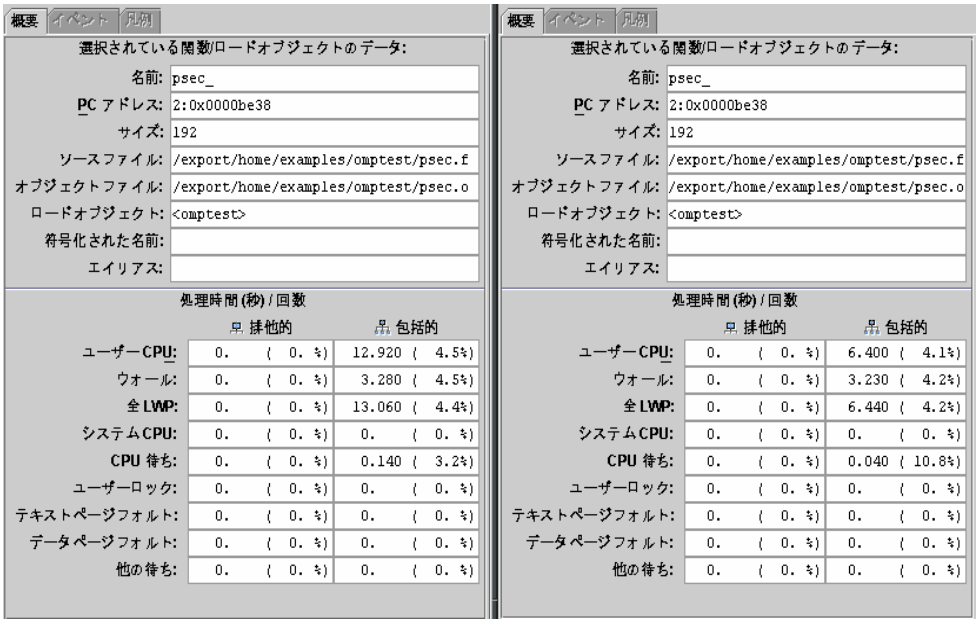
この節では、それぞれ PARALLEL SECTIONS 指令と PARALLEL DO 指令を使用する 、psec_() および pdo_() という 2 つのルーチンのパフォーマンスを比較します。これらルーチンのパフォーマンスは、CPU の個数の関数として比較されます。

4 つの CPU での実行と 2 つの CPU での実行を比較するには、2 つの「アナライザ」ウィンドウが必要です。 一方のウィンドウに omptest.4.er、もう一方のウィンドウに omptest.2.er を読み込みます。

1. 各パフォーマンスアナライザウィンドウの「関数」タブで、psec_ を見つけて選択します。

   この関数を見つけるには、「検索」ツールを利用できます。他にも、コンパイラによって生成された、psec_ から始まる関数があります。

2. 「概要」タブを比較できるようにウィンドウを左右に配置します。



この図では、左側のウィンドウが 4 つの CPU での実行データです。

3. ユーザー CPU 時間、ウォール時間、全 LWP 時間の包括的メトリックを比較します。

   2 つの CPU での実行の場合、ユーザー CPU 時間または全 LWP に対するウォール時間の比率は約 2 対 1 です。 これは、並列化の効率が比較的良いことを示しています。

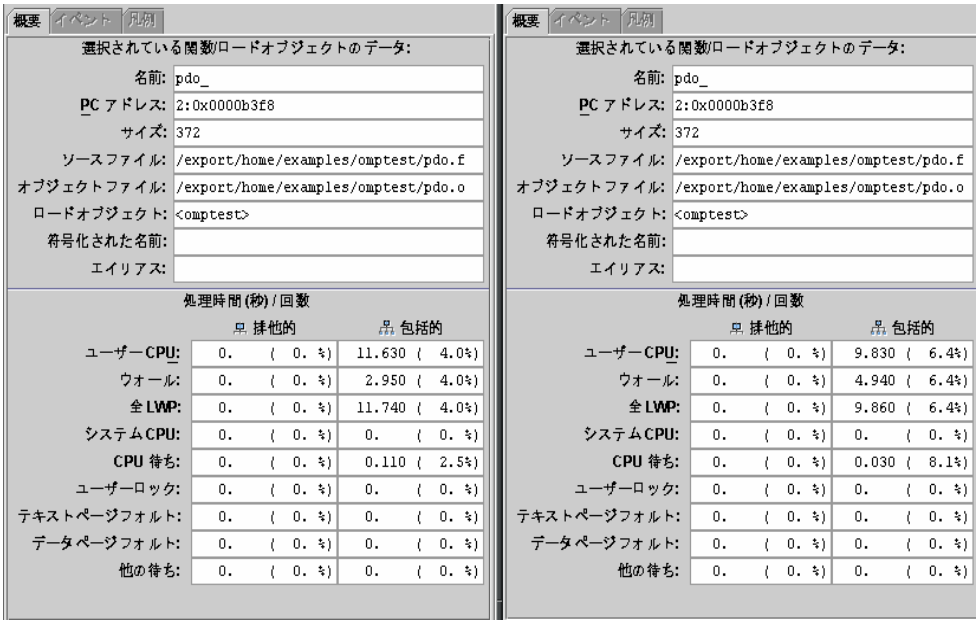
   これに対して 4 つの CPU での実行の場合、`psec_()` のウォール時間は 2 つの CPU での実行のときとほぼ同じですが、ユーザー CPU 時間と全 LWP 時間がともに長くなっています。 `psec_()` の `PARALLEL SECTION` 構文内のセクション数は 2 つだけであるため、その実行に必要なスレッドは 2 つだけです。 つまり、使用可能な 4 つの CPU のうちの 2 つだけがいつでも使用され、残りの 2 つのスレッドは、CPU 時間を消費して仕事を待ちます。他には仕事はないため、この時間は無駄に費やされます。

4. 両方のアナライザウィンドウについて、「関数リスト」から `pdo_` を含む行をクリックします。

   「概要メトリック」タブに `pdo_()` のデータが表示されます。

5. ユーザー CPU 時間、ウォール時間、全 LWP の包括的メトリックを比較します。

   pdo_() のユーザー CPU 時間は、psec_() の時間とほぼ同じです。 しかし、ユーザー CPU 時間に対するウォール時間の比率は、2 つの CPU で約 2 対 1 ですが、4 つの CPU では約 4 対 1 になっています。 このことは、pdo_() の並列化戦略では、利用できる CPU の個数を考慮し、ループを適切にスケジューリングすることによって、複数の CPU で効率性が増すことを意味しています。



   この図では、左側のウィンドウが 4 つの CPU での実行データです。

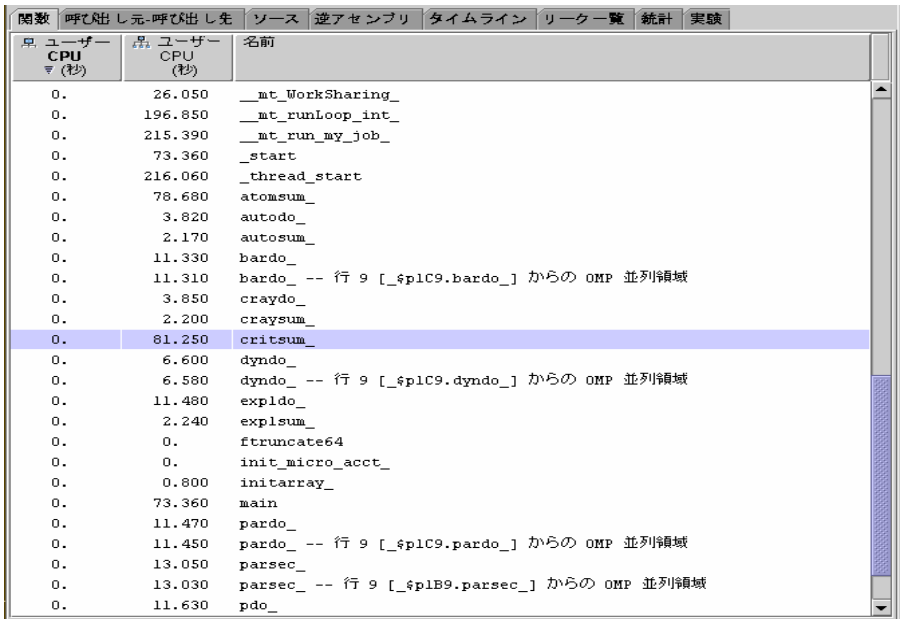
6. omptest.2.er を表示しているアナライザウィンドウを閉じます。

CRITICAL SECTION と REDUCTION 戦略の比較

この節では、それぞれ CRITICAL SECTIONS 指令と REDUCTION 指令を使用する critsec_() および reduc_() という、2 つのルーチンのパフォーマンスを比較します。 この事例では、1 組の do ループに埋め込まれている同じ代入文を並列化戦略によって処理します。その目的は、3 つの 2 次元配列の内容を合計することにあります。

```
t = (a(j,i)+b(j,i)+c(j,i))/k
sum = sum+t
```

1. 4 つの CPU を使用した実験 `omptest.4.er` について、「関数」タブで `critsum_()` と `redsum_()` を見つけます。



2. これら 2 つの関数の包括的ユーザー CPU 時間を比較します。

   `critsum_()` では、`CRITICAL SECTION` による並列化戦略が使用されているため、その包括的ユーザー CPU 時間は `redsum_()` の場合よりもはるかに長くなります。これは、加算演算は 4 つの CPU 間で分散されますが、`sum` への `t` の値の加算は一度に 1 つの CPU しか行えないためです。この種のコーディング構文の場合、これは、あまり効率的な並列化戦略ではありません。

   `redsum_()` の包括的ユーザー CPU 時間は、`critsum_()` よりもかなり短くなります。 これは、`redsum_()` では `REDUCTION` 戦略を採用していて、`(a(j,i)+b(j,i)+c(j,i))/k` の部分合計の計算が複数の CPU に分散され、その後、これらの中間値が `sum` に加算されるためです。この戦略によって、利用可能な CPU がかなり有効に利用されることになります。

例 3 :マルチスレッドプログラムにおけるロック戦略

mttest プログラムは、クライアント - サーバ関係におけるサーバーをエミュレートします。 この関係では、クライアントが要求をキューに入れ、サーバーは明示的なスレッド化を行うことによって複数のスレッドを使用し、それらの要求を処理します。 mttest について収集されたパフォーマンスデータは、さまざまなロック戦略から発生する各種競合と、実行時のキャッシュの影響が反映されたものになります。

mttest は、明示的にマルチスレッド機能を使用するようにコンパイルされており、複数または 1 つの CPU が搭載されたマシンでマルチスレッドプログラムとして動作します。複数 CPU システムと単一 CPU システムのパフォーマンスメトリックには、相違点とともに類似点もあります。

mttest に関するデータの収集

この節の手順に進む前に、6 ページの「サンプルプログラムの実行準備」 と 8 ページの「パフォーマンスアナライザの基本機能」 を参照してください。 この例を開始する前に、mttest をコンパイルします。

この例では、4 つの CPU での実行用の実験と、1 つの CPU での実行用の実験を作成します。 時間ベースのデータだけでなく、同期待ちトレースデータも記録します。 実験には、CPU の数を示すラベルが付きます。

mttest のデータを収集し、パフォーマンスアナライザを起動するには、以下のコマンドを入力します。

```
% cd work-directory/mttest
% collect -s on -o mttest.4.er mttest
% collect -s on -o mttest.1.er mttest -u
% analyzer mttest.4.er &
% analyzer mttest.1.er &
```

collect コマンドはメークファイルに含まれているため、以下のようにコマンドを入力することもできます。

```
% cd work-directory/mttest
% make collect
% analyzer mttest.4.er &
% analyzer mttest.1.er &
```

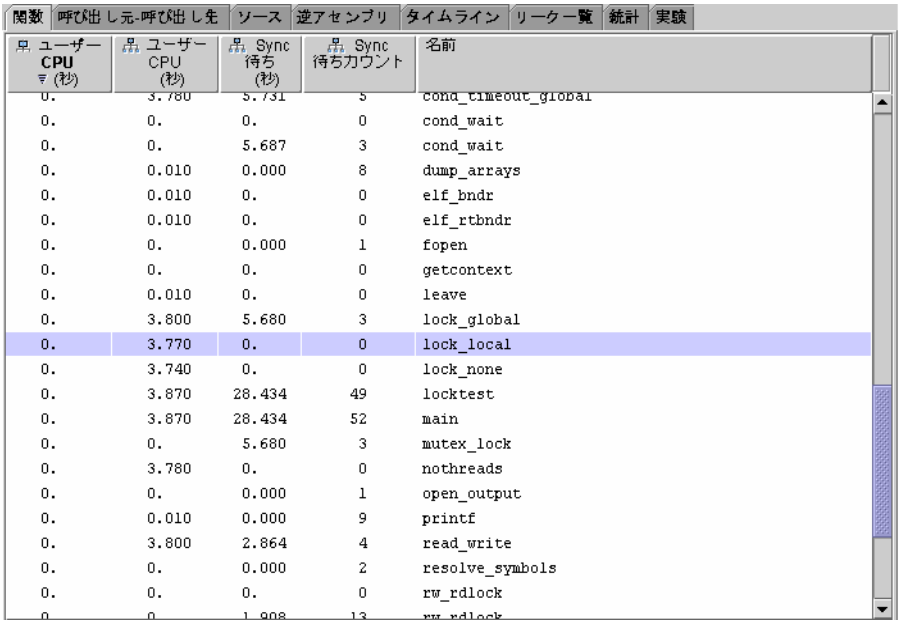
2 つの実験データが読み込まれたら、比較できるように 2 つの「パフォーマンスアナライザ」ウィンドウを左右に並べます。

これで、以降の節の手順に従って mttest 実験データを解析する準備ができました。

ロック戦略が待ち時間に及ぼす影響

1. 4 つの CPU を使用した実験 mttest.4.er の「関数」タブで、lock_local() と lock_global() を見つけます。

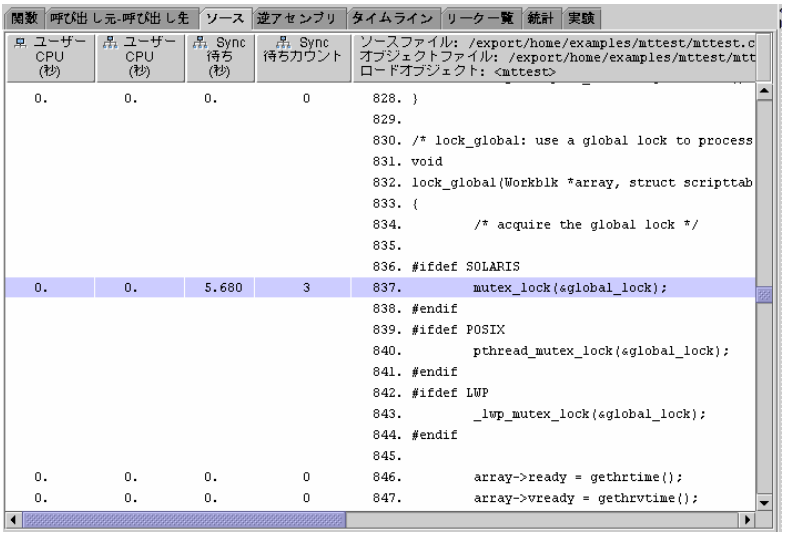
   この 2 つの関数の包括的ユーザー CPU 時間はほぼ同じなので、同量の仕事を行っていることになります。 ただし、lock_global() の同期待ち時間は長く、lock_local() の同期待ち時間はありません。



ユーザー CPU (秒)	ユーザー CPU (秒)	Sync 待ち (秒)	Sync 待ちカウント	名前
0.	3.780	5.731	5	cond_timeout_global
0.	0.	0.	0	cond_wait
0.	0.	5.687	3	cond_wait
0.	0.010	0.000	8	dump_arrays
0.	0.010	0.	0	elf_bndr
0.	0.010	0.	0	elf_rtbndr
0.	0.	0.000	1	fopen
0.	0.	0.	0	getcontext
0.	0.010	0.	0	leave
0.	3.800	5.680	3	lock_global
0.	3.770	0.	0	lock_local
0.	3.740	0.	0	lock_none
0.	3.870	28.434	49	locktest
0.	3.870	28.434	52	main
0.	0.	5.680	3	mutex_lock
0.	3.780	0.	0	nothreads
0.	0.	0.000	1	open_output
0.	0.010	0.000	9	printf
0.	3.800	2.864	4	read_write
0.	0.	0.000	2	resolve_symbols
0.	0.	0.	0	rw_rdlock
0	0	1.908	13	rw_rdlock

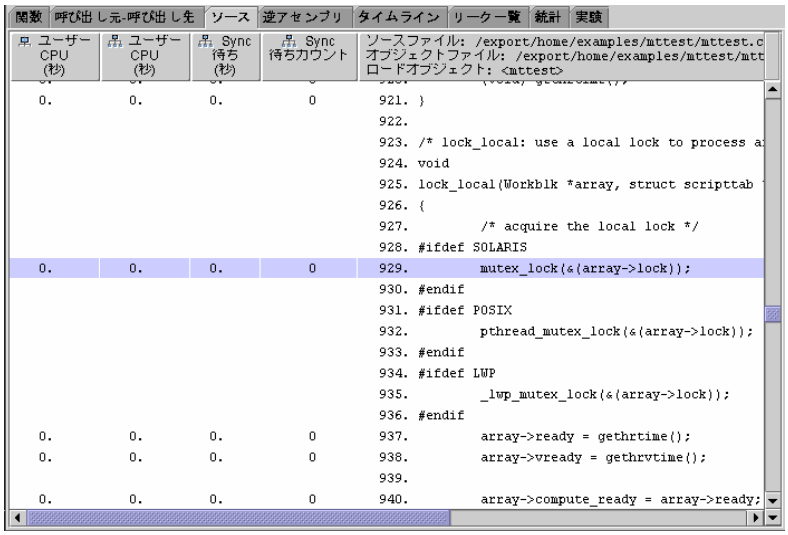
これらの関数の注釈付きソースコードを見ると、この理由が分かります。

2. `lock_global()` をクリックし、「ソース」タブをクリックします。



`lock_global()` では、大域ロックを使用して、すべてのデータが保護されています。このため、実行中のすべてのスレッド間で常にデータへのアクセス競合が発生し、データにアクセスできるのは常に 1 つのスレッドだけになります。残りのスレッドがデータにアクセスするには、作業中のスレッドがロックを解除するまで待つ必要があります。同期待ち時間の原因になっているのはソースコードのこの行です。

3. 「関数」タブで `lock_local()` をクリックし、「ソース」タブをクリックします。



`lock_local()` は、特定のスレッドの作業ブロック内のデータだけをロックします。どのスレッドも他のスレッドの作業ブロックにはアクセスできないため、スレッドは、競合や同期待ちによる無駄な時間なしに処理を続行できます。 このため、このコード行である `lock_local()` の同期待ち時間はゼロになります。

4. 1 つの CPU を使用した実験 `mttest.1.er` について、次のように選択対象メトリックを変更します。

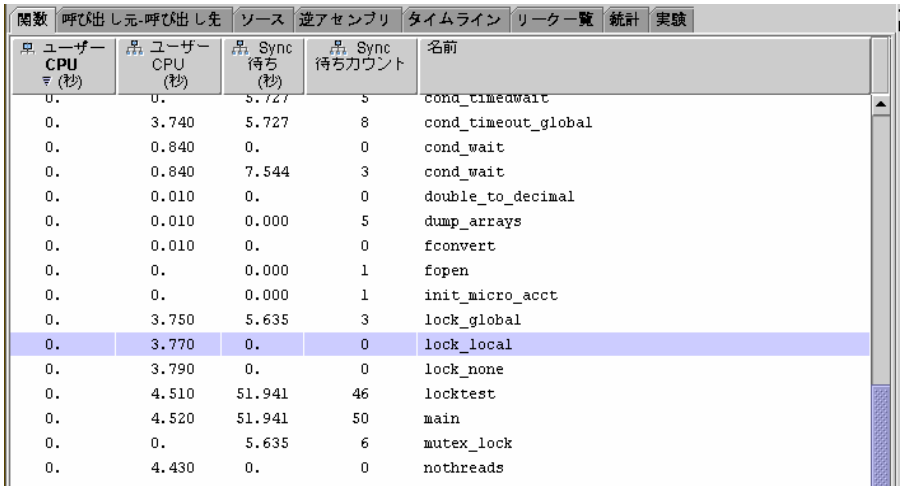
   a. 「表示」>「データ表示方法の設定」を選択します。

   b. 「排他的ユーザー CPU 時間」と「包括的 Sync 待ちカウント」を選択解除します。

   c. 「包括的全 LWP 時間」、「包括的 CPU 待ち時間」、および「包括的他の待ち時間」を選択します。

   d. 「適用」をクリックします。

5. 1 つの CPU を使用した実験の「関数」タブで、`lock_local()` と `lock_global()` を見つけます。



ユーザー CPU (秒)	ユーザー CPU (秒)	Sync 待ち (秒)	Sync 待ちカウント	名前
0.	0.	5.727	5	cond_timedwait
0.	3.740	5.727	8	cond_timeout_global
0.	0.840	0.	0	cond_wait
0.	0.840	7.544	3	cond_wait
0.	0.010	0.	0	double_to_decimal
0.	0.010	0.000	5	dump_arrays
0.	0.010	0.	0	fconvert
0.	0.	0.000	1	fopen
0.	0.	0.000	1	init_micro_acct
0.	3.750	5.635	3	lock_global
0.	3.770	0.	0	lock_local
0.	3.790	0.	0	lock_none
0.	4.510	51.941	46	locktest
0.	4.520	51.941	50	main
0.	0.	5.635	6	mutex_lock
0.	4.430	0.	0	nothreads

4 つの CPU の実験同様、両方の関数の包括的ユーザー CPU 時間は同じであり、このため、両方の仕事量は同じということになります。同期動作も、4 つの CPU を使用したシステムの場合と同じです。`lock_global()` では同期待ちに大量の時間を費やしますが、`lock_local()` では費やしません。

ただし、実際には、`lock_global()` の全 LWP 時間は `lock_local()` よりも短くなっています。これは、それぞれのロックシステムにおけるスレッド実行のスケジューリング方法が原因です。`lock_global()` の大域ロックでは、各スレッドは処理が完了するまで順次動作できます。これに対して `lock_local()` のローカルロックでは、その実行時間のほんの一部が各スレッドに割り当てられ、すべてのスレッドが完了するまでこのプロセスが繰り返されることになります。こうして、どちらの場合も、スレッドは大量の時間を仕事待ちに費やします。`lock_global()` のスレッドは、ロックを待ちます。この待ち時間は「包括的 Sync 待ち時間」メトリックと「その他の待ち時間」メトリックに計上されます。`lock_local()` のスレッドは、CPU を待ちます。この待ち時間は、「CPU 待ち時間」メトリックに計上されます。

6. メトリックの選択内容を `mttest.1.er` のデフォルト値に戻します。

   まだ開かれているはずの「データ表示方法の設定」ダイアログで、次の操作を行います。

   a. 「排他的ユーザー CPU 時間」と「包括的 sync 待ちカウント」を選択します。

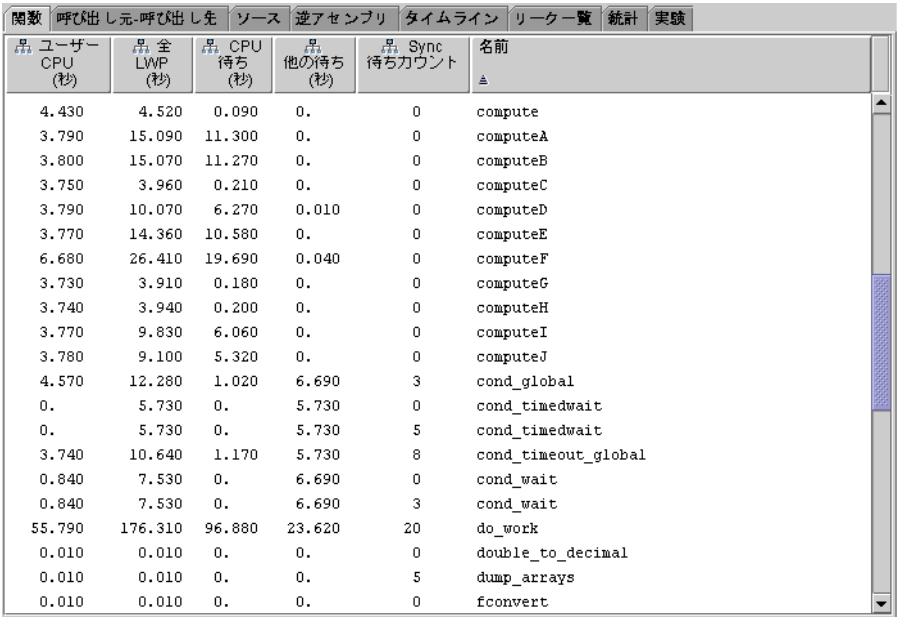
   b. 「包括的全 LWP 時間」、「包括的 CPU 待ち時間」、「包括的他の待ち時間」の時間表示欄を選択解除します。

   c. 「了解」をクリックします。

データ管理がキャッシュのパフォーマンスに及ぼす影響

1. 両方の「パフォーマンスアナライザ」ウィンドウの「関数」タブで、`ComputeA()` と `ComputeB()` を見つけます。

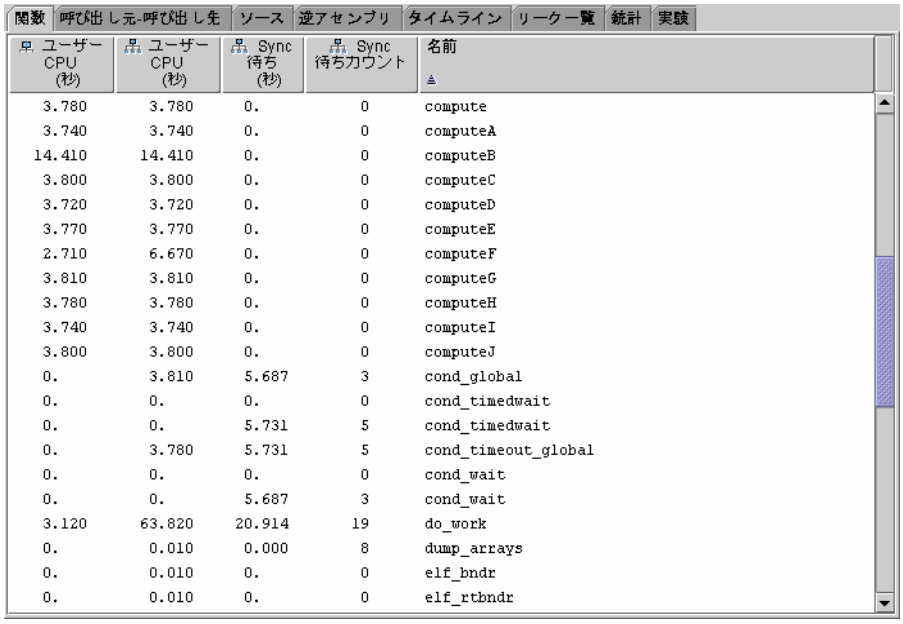
   単一 CPU の実験 `mttest.1.er` の場合、`ComputeA()` と `ComputeB()` の包括的ユーザー CPU 時間はほぼ同じです。



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

ユーザー CPU (秒)	全 LWP (秒)	CPU 待ち (秒)	他の待ち (秒)	Sync 待ちカウント	名前
4.430	4.520	0.090	0.	0	compute
3.790	15.090	11.300	0.	0	computeA
3.800	15.070	11.270	0.	0	computeB
3.750	3.960	0.210	0.	0	computeC
3.790	10.070	6.270	0.010	0	computeD
3.770	14.360	10.580	0.	0	computeE
6.680	26.410	19.690	0.040	0	computeF
3.730	3.910	0.180	0.	0	computeG
3.740	3.940	0.200	0.	0	computeH
3.770	9.830	6.060	0.	0	computeI
3.780	9.100	5.320	0.	0	computeJ
4.570	12.280	1.020	6.690	3	cond_global
0.	5.730	0.	5.730	0	cond_timedwait
0.	5.730	0.	5.730	5	cond_timedwait
3.740	10.640	1.170	5.730	8	cond_timeout_global
0.840	7.530	0.	6.690	0	cond_wait
0.840	7.530	0.	6.690	3	cond_wait
55.790	176.310	96.880	23.620	20	do_work
0.010	0.010	0.	0.	0	double_to_decimal
0.010	0.010	0.	0.	5	dump_arrays
0.010	0.010	0.	0.	0	fconvert

   4 つの CPU を使用した実験 `mttest.4.er` の場合、`ComputeB()` は、`ComputeA()` に比べてはるかに長い包括的ユーザー CPU 時間を費やします。

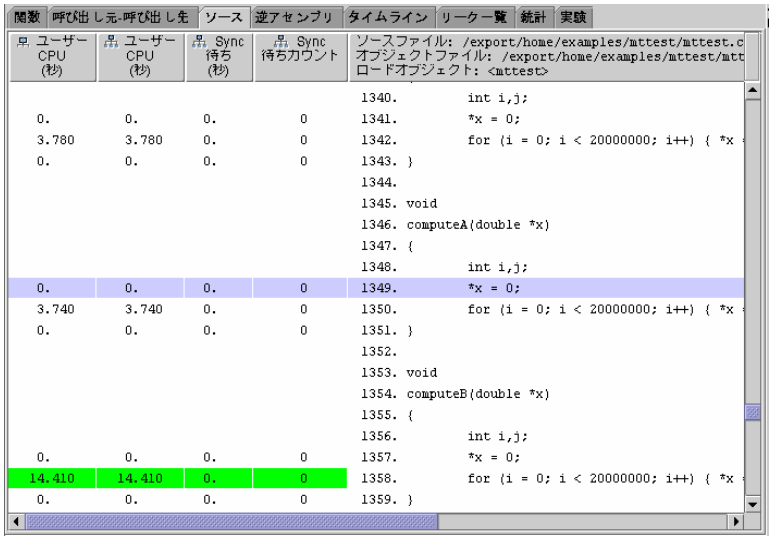


関数 | 呼び出し元-呼び出し先 | ソース | 逆アセンブリ | タイムライン | リーク一覧 | 統計 | 実験

ユーザー CPU (秒)	ユーザー CPU (秒)	Sync 待ち (秒)	Sync 待ちカウント	名前
3.780	3.780	0.	0	compute
3.740	3.740	0.	0	computeA
14.410	14.410	0.	0	computeB
3.800	3.800	0.	0	computeC
3.720	3.720	0.	0	computeD
3.770	3.770	0.	0	computeE
2.710	6.670	0.	0	computeF
3.810	3.810	0.	0	computeG
3.780	3.780	0.	0	computeH
3.740	3.740	0.	0	computeI
3.800	3.800	0.	0	computeJ
0.	3.810	5.687	3	cond_global
0.	0.	0.	0	cond_timedwait
0.	0.	5.731	5	cond_timedwait
0.	3.780	5.731	5	cond_timeout_global
0.	0.	0.	0	cond_wait
0.	0.	5.687	3	cond_wait
3.120	63.820	20.914	19	do_work
0.	0.010	0.000	8	dump_arrays
0.	0.010	0.	0	elf_bndr
0.	0.010	0.	0	elf_rtbndr

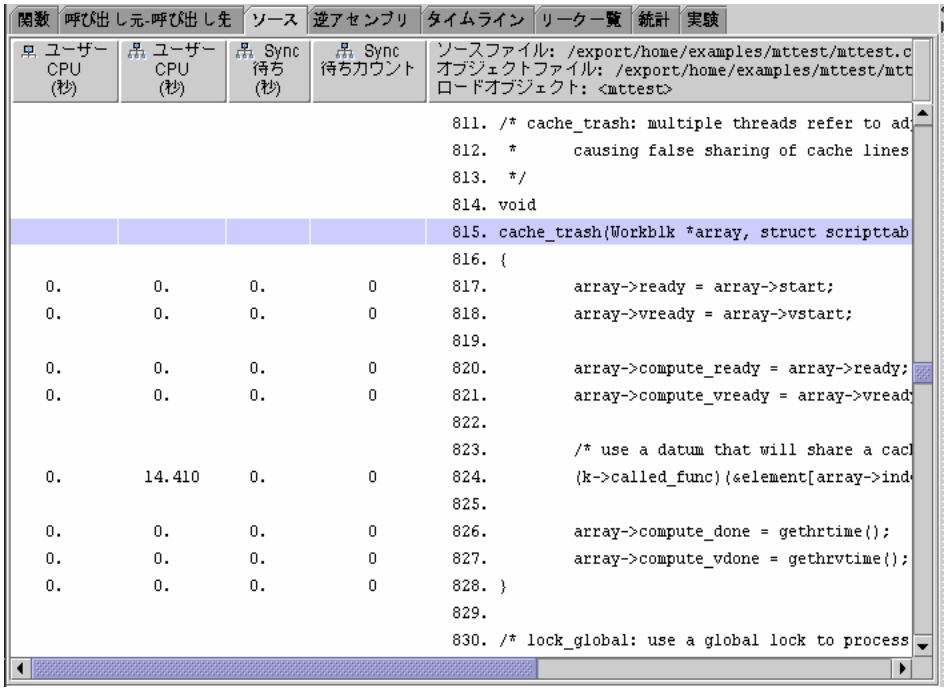
この後の操作は、4 つの CPU を使用した実験 `mttest.4.er` に対して行います。

2. `ComputeA()` をクリックし、「ソース」タブをクリックします。 テキストエディタのウィンドウを下方向にスクロールし、`ComputeA()` と `ComputeB()` のソースコードを表示します。



これらの関数のコードは同じで、どちらも変数に 1 を加算するループです。ユーザー CPU 時間は、すべてこのループで費やされています。 `ComputeB()` が `ComputeA()` より長い時間を費やす原因を探るには、これら 2 つの関数を呼び出すコードを調べる必要があります。

3. 「検索」ツールを使用して `cache_trash` を見つけます。 `cache_trash()` のソースコードが表示されるまで検索操作を繰り返します。



ComputeA() と ComputeB() は、ポインタによる参照で呼び出されるため、ソースコードには、それらの名前は表示されません。

cache_trash() が ComputeB() の呼び出し元であることは、「関数リスト」から ComputeB() を選択し、「呼び出し元-呼び出し先」をクリックすることによって確認できます。

4. computeA() と computeB() の呼び出しを比較します。

   computeA() は、スレッドの作業ブロック内にある 1 つの倍精度浮動小数点数型の値 (&array->list[0]) を引数として呼び出されます。 この引数は、他のスレッドと競合することはなく、読み取りと書き込みを直接行うことができます。

   これに対し、computeB() はメモリー内で連続するワードを占有する一連の倍精度浮動小数点数型の値 (&element[array->index]) を使用して呼び出されます。あるスレッドが、メモリー内のこれらアドレスの 1 つに書き込みを行う場合、キャッシュ内にそのアドレスの内容を保持している他のスレッドは、必ずそのデータを削除する必要があります。 これは、そのデータがすでに古くなっているためです。また、スレッドの 1 つが後でそのデータが必要になった場合は、そのデータが変更されていなくても、メモリーからデータキャッシュにデータをコピーし直す必要があります。この結

果、データキャッシュに存在しないデータにアクセスが試みられるというキャッシュミスが起こり、大量の CPU 時間の浪費になります。 computeB() が computeA() に比べてかなり長いユーザー CPU 時間を費やす原因はここにあります。

単一 CPU の実験では、一度に 1 つのスレッドが動作するだけであり、その間、他のスレッドがメモリーに書き込むことはできません。このため、動作中のスレッドのキャッシュデータが無効になることはありません。 キャッシュミスはなく、メモリーからのコピーもないため、使用できる CPU が 1 つだけの場合、ComputeB() は ComputeA() とちょうど同じパフォーマンス効率になります。

mttest の追加演習

1. 使用するコンピュータにハードウェアカウンタがある場合は、4 つの CPU の実験を再度実行し、キャッシュハードウェアカウンタのうちの 1 つから、キャッシュミスやストールサイクルなどのデータを収集してください。UltraSPARC III ハードウェアでは、次のコマンドを使用できます。

```
% collect -p off -h dcstall -o mttest.3.er mttest
```

   「ファイル」>「追加」を選択すれば、新しい実験の情報を以前の実験の情報と結合できます。 ComputeA と ComputeB のハードウェアカウンタデータを、「関数」タブと「ソース」タブで調べます。

2. メークファイルには、コメント形式で環境変数のオプション設定が含まれています。それらのオプションの一部を変更して、プログラムのパフォーマンスにどのような影響があるか確認してみてください。次の環境変数を試してみることを推奨します。
   - THREADS - スレッドモデルを選択します。
   - OFLAGS - コンパイラの最適化フラグ

例 4 :キャッシュの動作と最適化

この例では、効率的なデータへのアクセスと最適化の問題に取り組みます。標準 BLAS ライブラリに含まれている、行列対ベクトル乗算ルーチン dgemv の 2 つの実装版を使用します。プログラムには、それら 2 つのルーチンのコピーがそれぞれ 3 部含まれています。最初のコピーは、配列要素へのアクセス順序がルーチンのパフォーマ

ンスに及ぼす影響を見るために、最適化なしでコンパイルします。2 つめと 3 つめのコピーは、コンパイラによるループの順序変更と最適化の影響を見るために、それぞれ -O2 および -fast を指定してコンパイルします。

この例では、パフォーマンス解析におけるハードウェアカウンタおよびコンパイラのコメントの使用例についても取り上げます。

cachetest に関するデータの収集

この節の手順に進む前に、6 ページの「サンプルプログラムの実行準備」 と 8 ページの「パフォーマンスアナライザの基本機能」 を参照してください。 この例を開始する前に、chachetest をコンパイルします。

この例では、クロックベースのデータを含む実験だけでなく、異なるハードウェアカウンタから収集したデータを含む実験もいくつか作成します。

コマンド行から cachetest のデータを収集し、パフォーマンスアナライザを起動するには、以下のようにコマンドを入力します。

```
% cd work-directory/cachetest
% collect -o flops.er -S off -p on -h fpadd,,fpmul cachetest
% collect -o cpi.er -S off -p on -h cycles,,insts cachetest
% collect -o dcstall.er -h dcstall cachetest
```

パフォーマンスアナライザは、排他的メトリックだけを表示します。これはデフォルトとは異なり、ローカルのデフォルト値ファイルで設定しています。 詳細は、141 ページの「デフォルト値関連のコマンド」 を参照してください。

これで、以降の節の手順に従って cachetest 実験データを解析する準備ができました。

実行速度

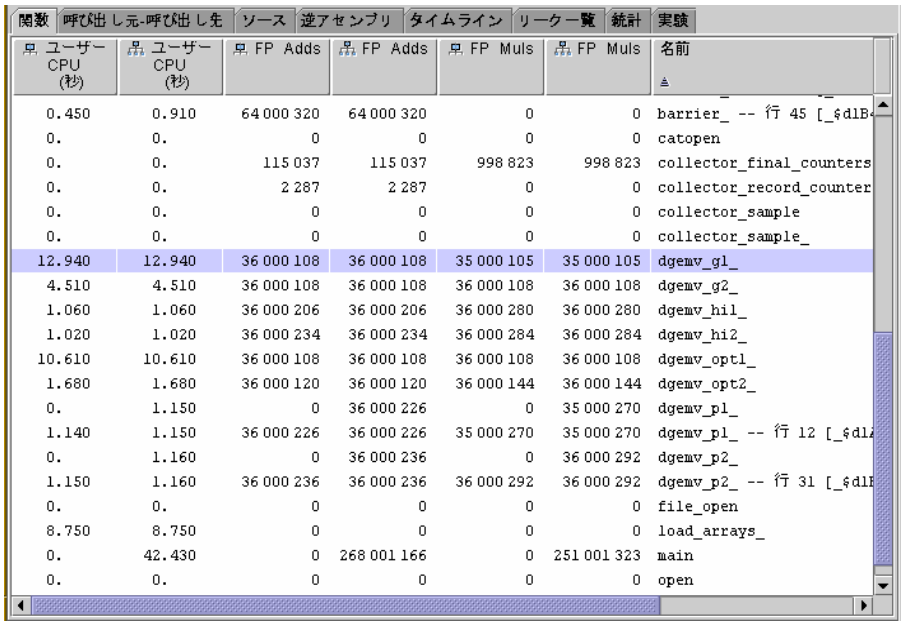
1. 浮動小数点演算実験を対象にアナライザを起動します。

```
% cd work-directory/cachetest
% analyzer flops.er &
```

2. 「名前」カラムのヘッダーをクリックします。

   関数は名前別にソートされ、選択された関数を基準としてセンタリング表示されます。

3. 6 つの関数 (dgemv_g1、dgemv_g2、dgemv_opt1、dgemv_opt2、dgemv_hi1、dgemv_hi2) のそれぞれについて、「FP Adds」と「FP Muls」の値を加算し、ユーザー CPU 時間と 10^6 で除算します。



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

ユーザー CPU (秒)	ユーザー CPU (秒)	FP Adds	FP Adds	FP Muls	FP Muls	名前
0.450	0.910	64 000 320	64 000 320	0	0	barrier_ -- 行 45 [_$dlB
0.	0.	0	0	0	0	catopen
0.	0.	115 037	115 037	998 823	998 823	collector_final_counters
0.	0.	2 287	2 287	0	0	collector_record_counter
0.	0.	0	0	0	0	collector_sample
0.	0.	0	0	0	0	collector_sample_
12.940	12.940	36 000 108	36 000 108	35 000 105	35 000 105	dgemv_g1_
4.510	4.510	36 000 108	36 000 108	36 000 108	36 000 108	dgemv_g2_
1.060	1.060	36 000 206	36 000 206	36 000 280	36 000 280	dgemv_hi1_
1.020	1.020	36 000 234	36 000 234	36 000 284	36 000 284	dgemv_hi2_
10.610	10.610	36 000 108	36 000 108	36 000 108	36 000 108	dgemv_opt1_
1.680	1.680	36 000 120	36 000 120	36 000 144	36 000 144	dgemv_opt2_
0.	1.150	0	36 000 226	0	35 000 270	dgemv_p1_
1.140	1.150	36 000 226	36 000 226	35 000 270	35 000 270	dgemv_p1_ -- 行 12 [_$dli
0.	1.160	0	36 000 236	0	36 000 292	dgemv_p2_
1.150	1.160	36 000 236	36 000 236	36 000 292	36 000 292	dgemv_p2_ -- 行 31 [_$dlI
0.	0.	0	0	0	0	file_open
8.750	8.750	0	0	0	0	load_arrays_
0.	42.430	0	268 001 166	0	251 001 323	main
0.	0.	0	0	0	0	open

   この計算で、各ルーチンの MFLOPS カウント値が得られます。これらのサブルーチンが発行する浮動小数点演算命令数はすべて同じですが、消費される CPU 時間の長さはそれぞれ異なります(カウント値の違いは統計上の問題が原因です)。 パフォーマンスとして、dgemv_g2 が dgemv_g1 よりも良く、dgemv_opt2 が dgemv_opt1 よりも良くなっていますが、dgemv_hi2 と dgemv_hi1 のパフォーマンスはほぼ同じです。

4. ここで得られた MFLOPS カウント値とプログラムによって出力された MFLOPS 値とを比較します。

   ハードウェアカウンタデータの収集にオーバーヘッドがかかるため、データから求められた値の方が小さくなります。

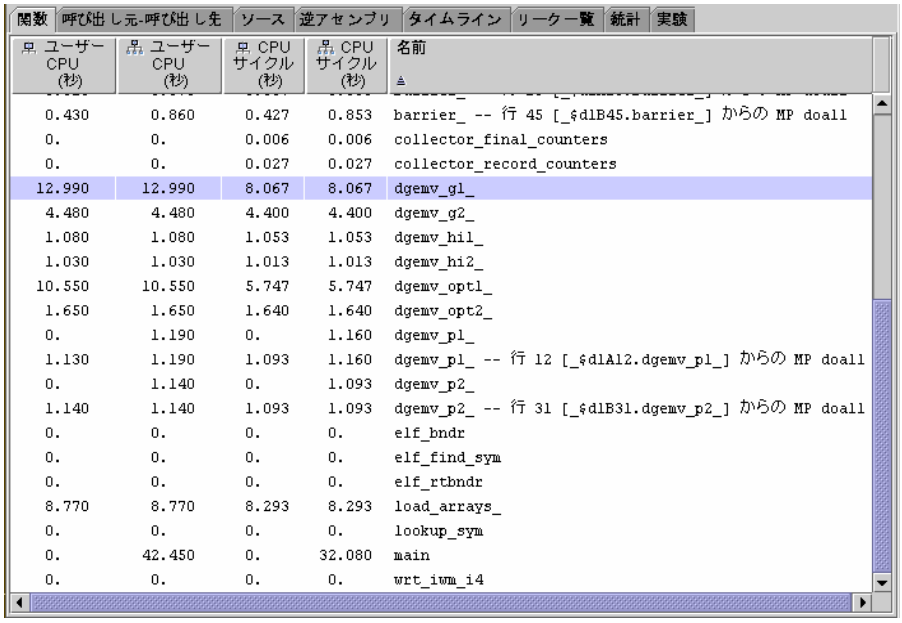
プログラムの構造とキャッシュの動作

この節では、dgemv_g2 のパフォーマンスが dgemv_g1 よりも優れている理由を検討します。パフォーマンスアナライザがすでに稼動している場合には、次の操作を行います。

1. 「ファイル」>「実験ファイルを開く」を選択し、 cpi.er を開きます。
2. 「ファイル」>「実験ファイルの追加」を選択し、 dcstall.er を追加します。

パフォーマンスアナライザが稼動していない場合には、次のコマンドをプロンプトの後に入力します。

```
% cd work-directory/cachetest
% analyzer cpi.er dcstall.er &
```



ユーザー CPU (秒)	ユーザー CPU (秒)	CPU サイクル (秒)	CPU サイクル (秒)	名前
0.430	0.860	0.427	0.853	barrier_ -- 行 45 [_$dlB45.barrier_] からの MP doall
0.	0.	0.006	0.006	collector_final_counters
0.	0.	0.027	0.027	collector_record_counters
12.990	12.990	8.067	8.067	dgemv_g1_
4.480	4.480	4.400	4.400	dgemv_g2_
1.080	1.080	1.053	1.053	dgemv_hi1_
1.030	1.030	1.013	1.013	dgemv_hi2_
10.550	10.550	5.747	5.747	dgemv_opt1_
1.650	1.650	1.640	1.640	dgemv_opt2_
0.	1.190	0.	1.160	dgemv_p1_
1.130	1.190	1.093	1.160	dgemv_p1_ -- 行 12 [_$dlA12.dgemv_p1_] からの MP doall
0.	1.140	0.	1.093	dgemv_p2_
1.140	1.140	1.093	1.093	dgemv_p2_ -- 行 31 [_$dlB31.dgemv_p2_] からの MP doall
0.	0.	0.	0.	elf_bndr
0.	0.	0.	0.	elf_find_sym
0.	0.	0.	0.	elf_rtbndr
8.770	8.770	8.293	8.293	load_arrays_
0.	0.	0.	0.	lookup_sym
0.	42.450	0.	32.080	main
0.	0.	0.	0.	wrt_iwm_i4

1. 「ユーザー CPU 時間」と「CPU サイクル」の値を比較します。

dgemv_g1 の 2 つのメトリックの差は、DTLB (data translation lookaside buffer) ミスが原因です。 CPU が DTLB ミスの解決待ちの間もシステムクロックは動作し続けますが、サイクルカウンタはオフになります。 dgemv_g2 の両者の差は無視できるほどであり、DLTB ミスがわずかであることを示しています。

2. dgemv_g1 と dgemv_g2 の D キャッシュと E キャッシュの引き延ばし時間を比較します。

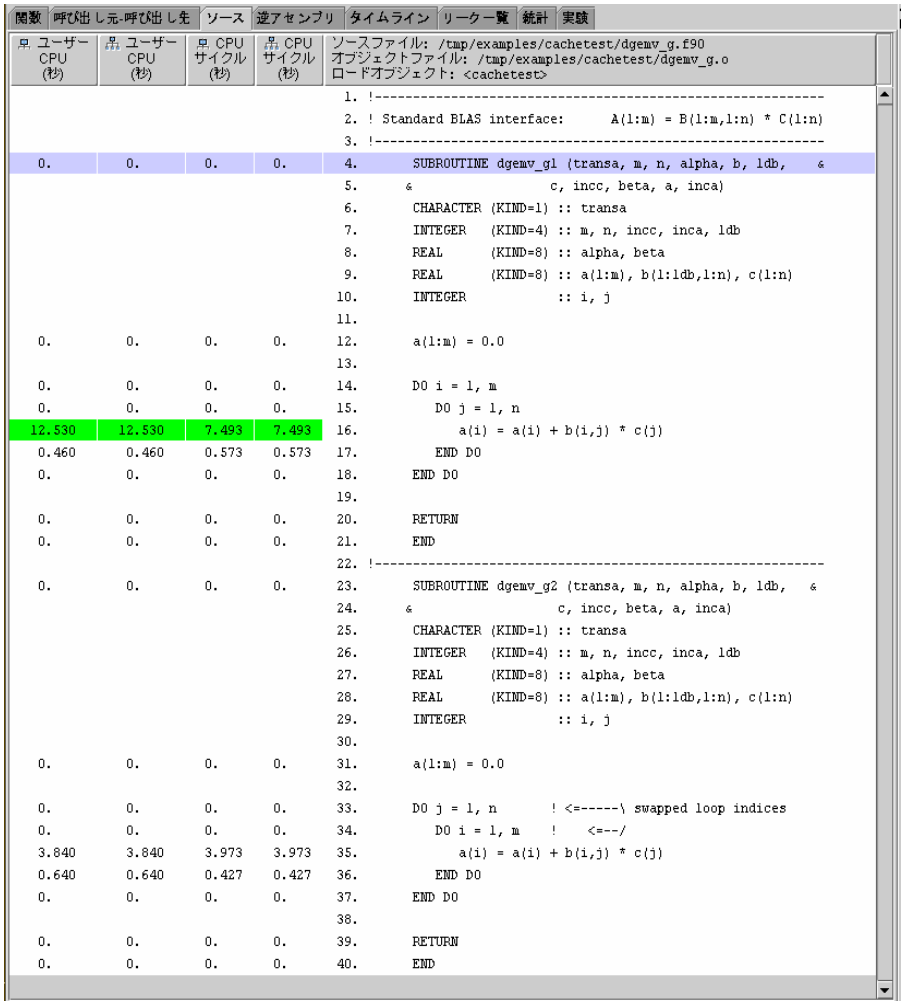
   キャッシュの再読み込み待ちに費やされる時間は、dgemv に比べて dgemv2 の方が短くなっています。 これは、dgemv2 のデータアクセス方法が、キャッシュがより効率的に利用される仕組みになっているためです。

   注釈付きソースコードを調べ、この理由を探ってみます。このためには、最初にメトリックの大部分を選択解除し、必要なデータだけが表示されるようにします。

3. 「表示」 >「データ表示方法の設定」を選択し、「メトリック」タブで「命令の実行」と「CPU サイクル」を選択解除します。

4. dgemv_g1 をクリックして「ソース」タブをクリックします。

5. ディスプレイのサイズを変更してスクロールし、dgemv_g1 と dgemv_g2 の両方のソースコードを表示します。



これら 2 つのルーチン内のループ構造は異なります。 コードが最適化されていないため、dgemv_g1 では、配列内のデータが行でアクセスされ、刻み幅が大きくなっています (この場合は 6000)。これが、DTLB ミスおよびキャッシュミスの原因です。 これに対して dgemv_g2 では、データが列でアクセスされ、ユニットごとの刻み幅になっています。あらゆるループの繰り返しでデータが連続しているため、大きなセグメントをマッピングして、キャッシュに読み込むことができ、そのセグメントが使用されて、別のセグメントが必要になったときにだけ、キャッシュミスが発生します。

プログラムの最適化とパフォーマンス

この節では、2 つの異なる最適化オプション (`-O2` および `-fast`) がプログラムのパフォーマンスに及ぼす影響を検討します。このときコードに発生した変化は、注釈付きソースコードに表示されるコンパイラのコメントに示されます。

1. **実験 `cpi.er` および `dcstall.er` をパフォーマンスアナライザに読み込みます。**

   前節から続けて行う場合には、「表示」>「データ表示方法の設定」を選択し、時間としての「CPU サイクル」と「命令の実行」のメトリックが選択されていることを確認します。

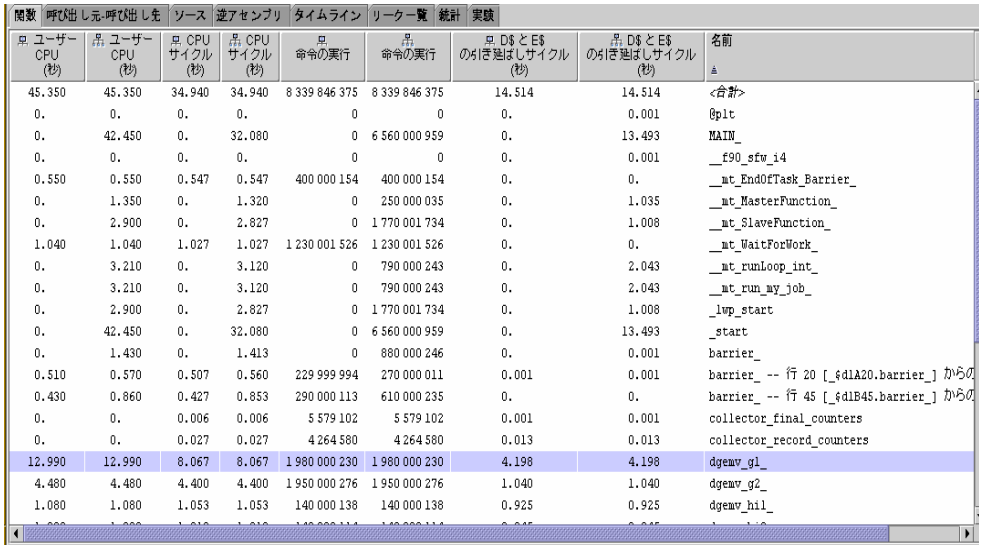
   パフォーマンスアナライザが稼動していない場合には、次のコマンドをプロンプトの後に入力します。

```
% cd work-directory/cachetest
% analyzer cpi.er dcstall.er &
```

2. **「名前」欄のヘッダーをクリックします。**

   関数は名前別にソートされ、選択された関数を基準としてセンタリング表示されます。

3. **`dgemv_opt1` および `dgemv_opt2` のメトリックと、`dgemv_g1` および `dgemv_g2` のメトリックを比較します。**



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

ユーザー CPU (秒)	ユーザー CPU (秒)	CPU サイクル (秒)	CPU サイクル (秒)	命令の実行	命令の実行	D$ と E$ のひき逃しサイクル (秒)	D$ と E$ のひき逃しサイクル (秒)	名前
45.350	45.350	34.940	34.940	8 339 846 375	8 339 846 375	14.514	14.514	<合計>
0.	0.	0.	0.	0	0	0.	0.001	@plt
0.	42.450	0.	32.080	0	6 560 000 959	0.	13.493	MAIN_
0.	0.	0.	0.	0	0	0.	0.001	__f90_sfw_i4
0.550	0.550	0.547	0.547	400 000 154	400 000 154	0.	0.	__mt_EndOfTask_Barrier_
0.	1.350	0.	1.320	0	250 000 035	0.	1.035	__mt_MasterFunction_
0.	2.900	0.	2.827	0	1 770 001 734	0.	1.008	__mt_SlaveFunction_
1.040	1.040	1.027	1.027	1 230 001 526	1 230 001 526	0.	0.	__mt_WaitForWork_
0.	3.210	0.	3.120	0	790 000 243	0.	2.043	__mt_runLoop_int_
0.	3.210	0.	3.120	0	790 000 243	0.	2.043	__mt_run_my_job_
0.	2.900	0.	2.827	0	1 770 001 734	0.	1.008	_lwp_start
0.	42.450	0.	32.080	0	6 560 000 959	0.	13.493	_start
0.	1.430	0.	1.413	0	880 000 246	0.	0.001	barrier_
0.510	0.570	0.507	0.560	229 999 994	270 000 011	0.001	0.001	barrier_ -- 行 20 [_$d1A20.barrier_] からの
0.430	0.860	0.427	0.853	290 000 113	610 000 235	0.	0.	barrier_ -- 行 45 [_$d1B45.barrier_] からの
0.	0.	0.006	0.006	5 579 102	5 579 102	0.001	0.001	collector_final_counters
0.	0.	0.027	0.027	4 264 580	4 264 580	0.013	0.013	collector_record_counters
12.990	12.990	8.067	8.067	1 980 000 230	1 980 000 230	4.198	4.198	dgemv_g1_
4.480	4.480	4.400	4.400	1 950 000 276	1 950 000 276	1.040	1.040	dgemv_g2_
1.080	1.080	1.053	1.053	140 000 138	140 000 138	0.925	0.925	dgemv_hi1_

dgemv_opt1 および dgemv_opt2 のソースコードは、それぞれ dgemv_g1 および dgemv_g2 のコードとまったく同じです。 違いは、これらのルーチンがコンパイラオプションの -O2 を指定してコンパイルされていることです。 ユーザー CPU 時間と CPU サイクルのどちらの測定でも、両方の関数とも CPU 時間が同程度に短くなっており、実行対象命令の数も同程度に少なくなっていますが、どちらのルーチンでも、キャッシュの動作の改善は見られません。

4. 「関数」タブで、dgemv_opt1 および dgemv_opt2 のメトリックと、dgemv_hi1 および dgemv_hi2 のメトリックを比較します。

   dgemv_hi1 および dgemv_hi2 のソースコードは、それぞれ dgemv_opt1 および dgemv_opt2 のコードとまったく同じです。 違いは、これらのルーチンがコンパイラオプションの -fast を指定してコンパイルされていることです。いずれのルーチンも、CPU 時間とキャッシュパフォーマンスは同じです。 dgemv_hi1 および dgemv_hi2 の CPU 時間とキャッシュストールサイクル時間は、dgemv_opt1 および dgemv_opt2 の値よりも短くなっています。 読み込み対象キャッシュの待ち時間は、実行時間のほぼ 80% を占めます。

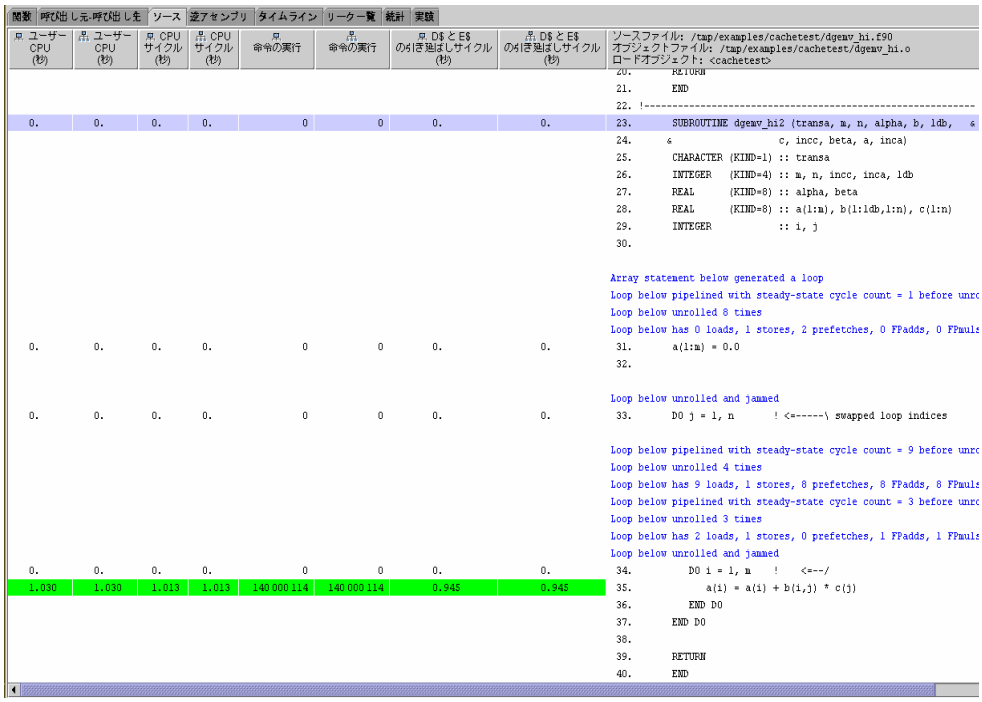
5. dgemv_hi1 をクリックしてから「ソース」タブをクリックします。 ディスプレイのサイズを変更してスクロールし、dgemv_hi1 のソースコードをすべて表示します。

コンパイラは、この関数を最適化するために多くの仕事をしました。DTLB ミスの原因であったループを交換しました。さらに、ループサイクルごとに浮動小数点加算演算と浮動小数点乗算演算が多いループを新たに作成し、プリフェッチ命令を挿入してキャッシュ動作を改良しました。

メッセージは、ソースコード内のループのほか、コンパイラがソースコードから生成するあらゆるループを対象としています。

6. 下方向にスクロールしてdgemv_hi2 のソースコードを表示します。

ループの入れ替えを除けば、コンパイラのコメントは dgemv_hi1 に対するものと同じです。コンパイラが生成した dgemv2_hi の 2 つのバージョンのコードの間に、基本的に違いはありません。



7. 「逆アセンブリ」タブをクリックします。

   「逆アセンブリ」リストを dgemv_g1 や dgemv_opt1 の「逆アセンブリ」リストと比較します。 dgemv_hi1 では、生成された命令数がかなり多くなっていますが、実行命令数は、このルーチンの 3 つのバージョンの中では最少です。最適化によって、生成される命令数が多くなることがありますが、命令の使用効率が向上して実行回数が減少します。

第3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の収集と格納については、第 4 章を参照してください。パフォーマンスデータの解析については、第 5 章 と 第 6 章を参照してください。

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

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の収集は、プログラムとシステムのプロファイルを一定の時間間隔で記録することによって行います。間隔は、システム時間を使用して取得した時間間隔、または特定のタイプのハードウェアイベントの数です。指定の間隔に達するとシグナルがシステムに送られ、次の機会にデータが記録されます。
- トレースデータの収集は、さまざまなシステム関数をラッパー関数で割り込み、それによってシステム関数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 ID
- 呼び出しスタックのコピー

スレッドと軽量プロセスについての詳細は、第 7 章を参照してください。

こうした共通の情報の他に、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 5 種類です。

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

この 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、次の 5 項で説明します。

時間データ

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、10 ミリ秒です。 コレクタは、1 ミリ秒の高分解能プロファイル間隔と、100 ミリ秒の低分解能プロファイル間隔を提供します。

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 3-1 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
ウォール時間	LWP 1 で費やした LWP 時間。一般的な「時計時間」です。
全 LWP 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。
その他の待ち時間	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 deta) 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 (124 ページの「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。

ハードウェアカウンタのオーバーフローデータ

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。

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、または命令変換ルックアサイドバッファ (ITLB) ミスが発生する可能性があります。

ハードウェアカウンタのリスト

ハードウェアカウンタはプロセッサ固有であるため、どのカウンタを利用できるかは、使用しているプロセッサによって異なります。便宜を考え、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` を端末ウィンドウに入力します。

別名があるカウンタのカウンタリスト内のエントリの形式は、次の例のようになっています。

```
CPU Cycles (cycles = Cycle_cnt/*) 9999991 hi=1000003, lo=100000007
```

最初のフィールドの「CPU Cycles」は、対応するパフォーマンスアナライザメトリックの名前です。括弧内の等号の左側にあるのは、カウンタの別名 (上記の例では「cycles」) です。 等号の右側のフィールド (上記の例では「Cycle_cnt/*」) は、cputrack(1) によって使用される内部名 (Cycle_cnt)、スラッシュ、およびそのカウンタに使用可能なレジスタ番号です。 レジスタ番号は、0 または 1 です。カウンタがどちらのレジスタでもカウントできる場合には * とします。 括弧の後の最初のフィールドはデフォルトのオーバーフロー値、次のフィールドはデフォルトの高分解能オーバーフロー値、最後のフィールドはデフォルトの低分解能オーバーフロー値です。

表 3-2 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 3-2　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

別名のないカウンタの、カウンタリスト内のエントリの形式は、次の例のようになっています。

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

Cycle_cnt は、cputrack(1) によって使用されるような内部名です。 文字列 Cycle_cnt Events は、このカウンタのパフォーマンスアナライザメトリックの名前です。この後の括弧に囲まれている文字列が、このイベントをカウント可能なレジスタを示します。 括弧の後の最初のフィールドはデフォルトのオーバーフロー値、次のフィールドはデフォルトの高分解能オーバーフロー値、最後のフィールドはデフォルトの低分解能オーバーフロー値です。

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。

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、スレッドライブラリ `libthread.so` の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。

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

同期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。

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 3-3　同期待ちトレースメトリック

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

この情報から、関数またはロードオブジェクトが頻繁にブロックされるかどうか、または同期ルーチンを呼び出したときの待ち時間が異常に長くなっているかどうかを調べることができます。同期待ち時間が大きいということは、スレッド間の競合が発生

していることを示します。競合は、アルゴリズムの変更、具体的には、ロックする必要があるデータだけがスレッドごとにロックされるように、ロックを構成し直すことで減らすことができます。

ヒープトレース (メモリー割り当て) データ

正しく管理されていないメモリー割り当て関数やメモリー割り当て解除関数を呼び出すと、データの使い方の効率が低下し、プログラムパフォーマンスが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`、`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。 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 のメモリー割り当てでは C メモリー割り当て関数を使用しないので、トレースは行われません。

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 3-4 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
割り当てられたバイト数	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 `free` 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
リークしたバイト数	割り当てられたが解放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るメモリーブロック」です。ここで使用するリークの定義には、この定義も含まれます。

MPI トレースデータ

コレクタは、Message Passing Interface (MPI) ライブラリの呼び出しに関するデータを収集できます。 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_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 3-5　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 関数の呼び出し数
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 関数の呼び出し数
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
その他の 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 3-6 にまとめます。

表 3-6　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_Send、MPI_Bsend、MPI_Rsend、MPI_Ssend
MPI 受信関数	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 (68 ページの「プログラムからのデータ収集の制御」 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 (85 ページの「実験制御関連のオプション」 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 7 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他に、より上位の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。

特定の呼び出しスタックの一番下の関数 (リーフ関数) が、他の関数を呼び出すことはありません。 このため、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。 呼び出しに関わっている 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る。
- 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する。

関数レベルのメトリックの意味:例

図 3-1 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。

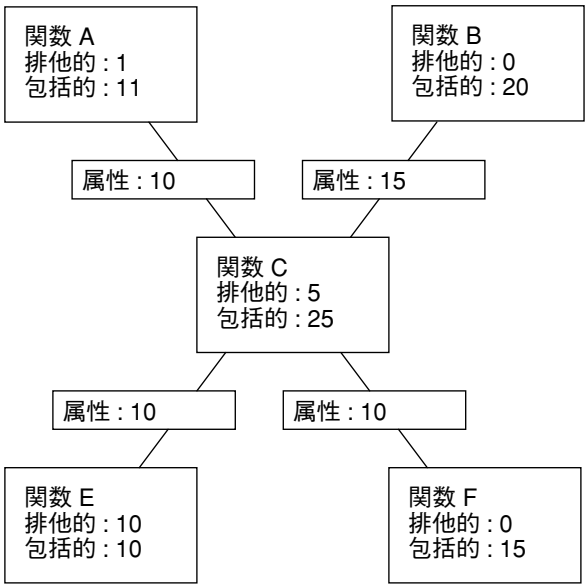


図 3-1　呼び出しツリーにおける排他的、包括的、属性メトリックの関係

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、リーフ関数の排他的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、リーフ関数の排他的メトリックが、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。 最も簡単な再帰では、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4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 3 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。

- データ収集と解析のためのプログラムの準備
- プログラムのコンパイルとリンク
- 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
- `collect` コマンドによるデータの収集
- 統合開発環境 (IDE) でのデータの収集
- 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

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
- モジュール (`.o` ファイル) を動的に読み込む

- 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。

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、スレッドライブラリ `libthread.so` の関数上で割り込み処理を行います。
- 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、MPI ライブラリ `libmpi.so` の関数上で割り込み処理を行います。
- 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 シグナルと SIGEMT シグナルの処理の保証 コレクタは sigaction で割り込み処理を行い、そのシグナルハンドラがプライマリシグナルハンドラであることを確保します。

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。

シグナルハンドラの使用

コレクタは、SIGPROF と SIGEMT の 2 種類の シグナルを使用してプロファイルデータを収集します。 コレクタはこの 2 つ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シグナルをインターセプトして処理しますが、使用対象でないシグナルは、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。 これらのシグナルは、コレクタによって保護されません。 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、キャンセルシグナルの到着が遅れます。

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。

プログラムからのデータ収集の制御

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 インタフェースが用意されています。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。

C または C++ からこれらの API 関数を使用するには、次の文を挿入します。

```
#include "libcollector.h"
```

関数は、次のように定義されます。

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

Fortran から API 関数を使用するには、次の文を挿入します。

```
include libfcollector.h
```

プログラムのリンクでは、`-lfcollector` を使用します。

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。

C インクルードファイルには、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。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。 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。

以下では、データ収集に関係する 4 つの API 関数について説明します。

collector_sample(char *name) (C と C++)

collector_sample(string) (Fortran)

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。ただし、パフォーマンスアナライザは、現在このラベルを使用しません。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。

collector_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。

collector_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。

collector_terminate_expt()

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 プログラムまたは C++ プログラムが、関数を動的にコンパイルしたり、モジュール (.o ファイル) をプログラムのデータ空間に動的に読み込んだりする場合、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);
void collector_module_load(char *modulename, void *vaddr);
void collector_module_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 4 つの 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す。パラメータリストを下表に示します。

表 4-1 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。
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。

collector_module_load()

モジュール modulename がプログラムによってアドレス vaddr のアドレス空間に読み込まれたことをコレクタに通知します。モジュールが読み込まれ、その関数とそのソースと行番号のマッピングとが確認されます。

collector_module_unload()

アドレス vaddr に読み込まれていたモジュールが読み込み解除されたことをコレクタに通知します。

プログラムのコンパイルとリンク

プログラムのコンパイル時に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

ソースコード情報を表示するには、コンパイラオプション -g を使用する必要があります (C++ でフロントエンドのインライン化を有効にするには -g0)。 このオプションを使用すると、コンパイラはシンボルテーブルを生成します。 パフォーマンスアナライザは、このシンボルテーブルを使用し、ソース行番号とファイル名を取得し、コンパイラのコメントを表示します。このオプションを使用しなかった場合、注釈付きのソースコードおよび逆アセンブリコードリストを表示することはできません。 また、パフォーマンスアナライザのウィンドウに関数名が表示されないこともあります。マップファイルを生成したい場合にも、コンパイラオプション -g (または -xF) を使用する必要があります。

何らかの理由でオブジェクト (.o) ファイルの移動や削除を行う必要がある場合には、-xs オプションを使用してプログラムを読み込みます。このオプションを使用すると、ソースファイルに関するすべての情報が実行可能ファイルに入れられます。このオプションにより、解析前に実験とプログラム関係ファイルを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

何から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通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。

最適化レベルを 4 または 5 にして、IA プラットフォーム上で C プログラムをコンパイルすると、コレクタが呼び出しスタックを正確に展開できなくなります。この場合、信頼できるのは、関数の排他的メトリックだけになります。IA プラットフォーム上での C++ プログラムのコンパイルでは、C++ の例外を無効にする -noex (または -features=no@except) 以外の任意の最適化レベルを使用できます。 -noex オプションを使用した場合は、コレクタが呼び出しスタックを正確に展開できないため、信頼できるのは関数の排他的メトリックだけになります。

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。

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。

時間ベースのプロファイルに関する制限事項

プロファイル間隔は、システム時間の分解能の倍数である必要があります。デフォルトの分解能は 10 ミリ秒です。 システム時間の間隔を変更して 1 ミリ秒の分解能にすることによって、さらに高い分解能でプロファイルを行うことができます。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
```

詳細は、『Solaris カーネルのチューンアップ・リファレンスマニュアル』を参照してください。

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、すでに稼動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、97 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。

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。
- Solaris™ 8 より前のバージョンのオペレーティング環境では、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。
- 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、実験は終了されます。
- 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、実験は破損します。

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` 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。

- `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。 `system`、`popen`、および `sh` の呼び出しは、コレクタの処理対象ではありません。
- 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、97 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。

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 以上の Java™ 2 Software Development Kit を使用する必要があります。 Java 仮想マシン[1]のパスを `JDK_1_4_HOME`、`JDK_HOME`、 `JAVA_PATH`、`PATH` の環境変数のいずれか 1 つに指定してください。 。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。
- Java モニタや Java 割り当てのトレースデータを収集することはできません。ただし、Java メソッドから呼び出される C や C++ の関数のトレースデータを収集することはできます。
- 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。
- 64 ビットの JVM™ マシンを使用するには、このマシンをデフォルトマシンとするか、またはデータ収集時にマシンのパスを指定する必要があります。 64 ビット JVM マシン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。

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。

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、格納場所を変更することができます。

派生プロセスの実験は、祖先となる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。

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、100 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 です。数字は、fork または exec の索引です (成功したかどうか)。 たとえば祖先となる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
- 実行可能ファイルに記録されている絶対パス名
- 現在の作業ディレクトリ

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。

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。

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。数百という深さをも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 GB までです。

`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、53 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。 76 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、49 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。 高分解能プロファイルは、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、75 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* (ミリ秒) に設定します。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、システム時間の分解能に設定されます。 高分解能のプロファイルについては、75 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。

`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。 53 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。

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。
- 0, 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。76 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 - 同期待ちのトレースを無効にします。
- on - デフォルトのしきい値 (実行時の測定で値を決定) 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。

ヒープトレースデータは、Java メモリーの割り当てについては記録されません。

-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、57 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 - 定期的な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* - 定期的な標本収集を有効にし、標本収集間隔を *value* に設定します。 間隔値は整数、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 - コレクタが追跡する派生プロセスすべてについて、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。

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。 コレクタは、system(3C)、system(3F)、sh(3F)、および popen(3C) の呼び出しによって作成されたプロセスをたどりません。

-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 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。 collect コマンドは *program* が JVM マシンであるかどうかを確認しません。JVM マシンでない場合は、収集は失敗します。ただし、collect コマンドはプログラムが ELF 実行可能ファイルであるかどうか確認し、ELF 実行可能ファイルでない場合には、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、-y 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

-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* にパスが含まれていない場合は、現在のディレクトリが実験グループのディレクトリになります。

-o *experiment-name*

記録する実験の名前として *experiment-name* を使用します。 *experiment-name* 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。 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、78 ページの「実験名」を参照してください。

-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。

その他のオプション

`-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。

注 - このオプションは、Forte™ Developer 6 update 2 release 以降に変更されています。

`-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。

`-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。

サポートが中止されたオプション

`-a`

アドレス空間データ収集と表示のサポートは停止されました。今回のリリースからは、このオプションは無視され警告が出されます。

統合開発環境 (IDE) でのデータの収集

注 - パフォーマンスアナライザの GUI と IDE は、バージョン 8 および 9 の Solaris オペレーティング環境用 Forte™ for Java™ 4, Enterprise Edition の一部です。

IDE の Solaris Native Language Support モジュールのデバッガを使用すると、パフォーマンスデータを収集できます。IDE でのパフォーマンスデータの収集方法については、Solaris Native Language Support モジュールのオンラインヘルプを参照してください。

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 *value* - プロファイル間隔を *value* (ミリ秒単位) に設定します。デフォルト値は 10 ミリ秒です。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、システム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。 高分解能のプロファイルについては、75 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。

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、53 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。オーバーフロー値として 0 を指定すると、デフォルトのオーバーフロー値と解釈されます。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。

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。

76 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` - 同期待ちのトレースを有効にします。
- `off` - 同期待ちのトレースを無効にします。

- `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、`calibrate`（実行時の測定で決定されたしきい値を使用）または実際の値（マイクロ秒単位）を指定できます。 *value* にゼロ（`0`）を設定した場合は、待ち時間に関係なくすべてのイベントがトレースされます。 デフォルト値は `calibrate` です。

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*

標本採取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` - 定期的な標本採取を有効にします。
- `manual` - 定期的な標本採取を無効にします。 手動の標本採取は、依然として有効のままです。
- `period` *value* - 標本採取の間隔を *value* に設定します (秒単位)。

デフォルトの場合、標本採取間隔 *value* が 1 秒での定期的な標本採取が有効となります。

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。ただし、現在このラベルは使用されません。

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。

`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。

`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、78 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。

`filename` オプションのサポートは停止されました。 代わりとして、`experiment` が導入されました。 旧バージョンの Forte Developer ソフトウェアとの互換性のため、`experiment` の同義語として `filename` が受け付けられます。

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。

サポートが中止されたサブコマンド

`address_space`

アドレス空間データ収集のサポートは停止されました。今回のリリースからは、このオプションは無視されて警告が出されます。

`close`

`disable` と同じです。

enable_once

1 回の実行に限ってデータ収集を有効にするために使用されていました。今回のリリースからは、このオプションは無視されて警告が出されます。

quit

`disable` と同じです。

store filename

`store experiment` と同じです。

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 (コマンドラインバージョンと 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 - パフォーマンスアナライザの GUI と IDE は、バージョン 8 および 9 の Solaris オペレーティング環境用 Forte™ for Java™ 4, Enterprise Edition の一部です。

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。

```
(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、66 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。 ライブラリパスを設定するには、LD_LIBRARY_PATH を使用します。 SPARC V9 の 64 ビットアーキテクチャーを使用している場合は、環境変数 LD_LIBRARY_PATH64 も設定する必要があります。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 4-2 に、これらの環境変数に設定する値をまとめます。

表 4-2 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_64	/opt/SUNWspro/lib/v9

/opt/SUNWspro 以外のディレクトリに Forte Developer ソフトウェア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。100 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、78 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、183 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分からない場合は、`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、80 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。

第5章

パフォーマンスアナライザグラフィカルユーザーインタフェース

パフォーマンスアナライザは、標本コレクタの収集したプログラムのパフォーマンスデータを解析します。 この章では、パフォーマンスアナライザ GUI とその機能、およびその使用方法について簡単に説明します。パフォーマンスアナライザのオンラインヘルプシステムには、新しい機能、GUI ディスプレイ、GUI の使用方法、パフォーマンスデータの意味、パフォーマンス問題の検出、問題の対処方法、クィックリファレンス、キーボードショートカットとニモニック、およびチュートリアルに関する情報があります。

この章では、以下について説明します。

- パフォーマンスアナライザの実行
- パフォーマンスアナライザディスプレイ
- パフォーマンスアナライザの使用法

チュートリアル形式によるパフォーマンスアナライザの概要については、第 2 章 を参照してください。

パフォーマンスアナライザのデータの解析方法と、それらデータのプログラム構造への関連付け方法についての詳細は、第 7 章を参照してください。

注 - パフォーマンスアナライザの GUI と IDE は、バージョン 8 および 9 の Solaris™ オペレーティング環境用 Forte™ for Java™ 4, Enterprise Edition の一部です。

パフォーマンスアナライザの実行

パフォーマンスアナライザは、コマンド行と統合開発環境 (IDE) のどちらからでも起動できます。

IDE からパフォーマンスアナライザを起動するには、次のいずれかを行います。

- メニューバーから「デバッグ」>「パフォーマンスツールキット」>「アナライザを実行」を選択します。

  このオプションでは、最後に収集が行われた実験が自動的に読み込まれます。

- エクスプローラ上で実験をダブルクリックします。

コマンド行からパフォーマンスアナライザを起動するには、analyzer(1) コマンドを使用します。 analyzer コマンドの構文は次のとおりです。

```
% analyzer [-h] [-j jvm-path] [-J jvm-options] [-u] [-V] [-v] [experiment-list]
```

experiment-list は、実験名または実験グループ名のリストです。 実験名については、78 ページの「収集データの格納場所」 を参照してください。実験名を省略した場合は、パフォーマンスアナライザが起動したときに「実験を開く」ダイアログが表示されます。複数の実験名を指定した場合は、そのすべての実験ファイルが読み込まれます。

表 5-1に、analyzer コマンドのオプションをまとめます。

表 5-1　analyzer コマンドのオプション

-h	analyzer コマンドの使用法メッセージを出力します。
-j *jvm-path*	パフォーマンスアナライザの実行に使用する Java™ 仮想マシンのパスを指定します。
-J *jvm-options*	パフォーマンスアナライザの実行に使用する JVM™ マシンのオプションを指定します。

表 5-1　`analyzer` コマンドのオプション(続き)

`-u` *user-directory*	ユーザーディレクトリを指定します。ユーザーディレクトリには、IDE とパフォーマンスアナライザに関する設定情報が格納されます。
`-v`	パフォーマンスアナライザが起動中に情報を出力します。
`-V`	パフォーマンスアナライザのバージョン番号を標準出力に出力します。

パフォーマンスアナライザを終了するには、「ファイル」>「終了」を選択します。

パフォーマンスアナライザディスプレイ

「パフォーマンスアナライザ」ウィンドウは、メニューバー、ツールバー、およびデータ表示のための分割区画で構成されます。分割区画は、パフォーマンスアナライザの各種ディスプレイに使用される複数のタブ区画で構成されます。「パフォーマンスアナライザ」ウィンドウを、図 5-1 に示しています。

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。 メニューバーの中心には、選択された関数またはロードオブジェクトがテキストボックス内に表示されます。 この関数やロードオブジェクトは、関数に関する情報を表示するどのタブからでも選択できます。「ファイル」メニューからは、同じ実験データを使用する新しいパフォーマンスアナライザウィンドウを開くことができます。 各ウィンドウ (新規ウィンドウであってもオリジナルウィンドウであっても) からは、そのウィンドウを閉じたりすべてのウィンドウを閉じたりできます。

ツールバーには、「データ表示方法の設定」ダイアログボックス、「データをフィルタ」ダイアログボックス、および「関数の表示/非表示」ダイアログボックスを開くためのボタンがあります。 これらのダイアログボックスは、「表示」メニューから開くこともできます。 ツールバーには、「検索」ツールも入っています。 表示される順序でボタンのアイコンを以下に示します。

それぞれのタブに何が表示されるかについて、以下に説明します。

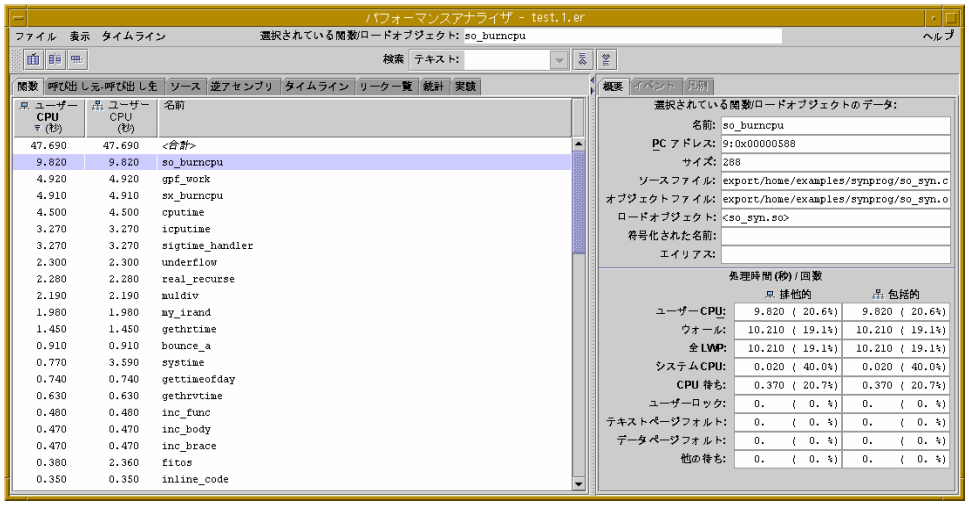
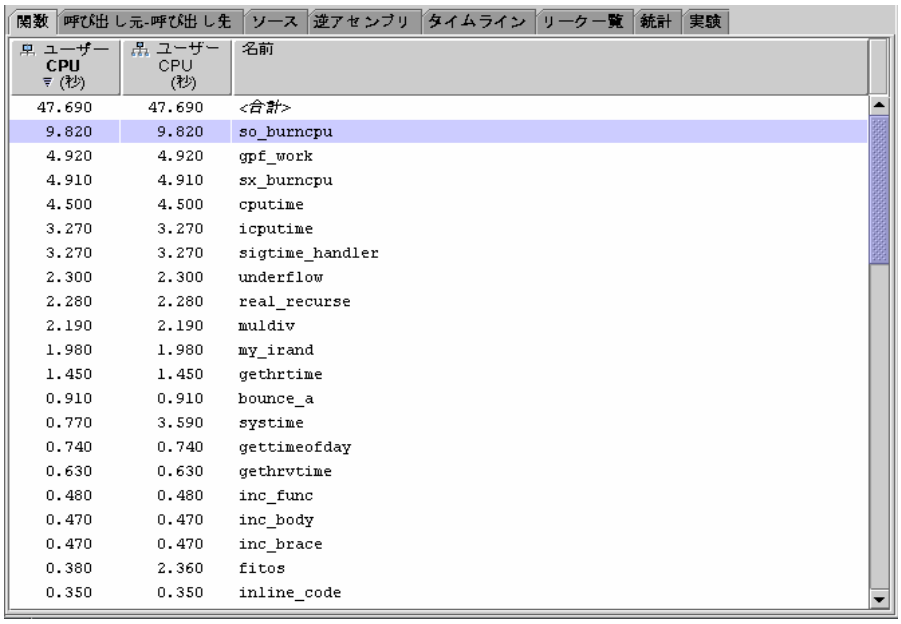


図 5-1 「パフォーマンスアナライザ」ウィンドウ

「関数」タブ

「関数」タブには、関数とロード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。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。 関数という用語には、Fortran サブルーチン、C++ メソッド、および Java™ メソッドが含まれます。 Java HotSpot™ 仮想マシンによってコンパイルされた Java メソッドは「関数」タブに表示されますが、Java 解析メソッドは表示されません。

「関数」タブには、包括的メトリックと排他的メトリックを表示できます。はじめに表示されるメトリックは、収集したデータとデフォルト設定とに基づいています。関数リストは、列のいずれか 1 つにあるデータ別にソートされます。この結果、メトリック値が高い関数を簡単に判別できます。ソート列のヘッダテキストは太字で表示され、列ヘッダの左下に三角形が表示されます。「関数」タブのソートメトリックを変更すると、「呼び出し元-呼び出し先」タブのソートメトリックが変更されます。ただし、「呼び出し元-呼び出し先」タブのソートメトリックが属性メトリックである場合には変更されません。



ユーザー CPU (秒)	ユーザー CPU (秒)	名前
47.690	47.690	<合計>
9.820	9.820	so_burncpu
4.920	4.920	gpf_work
4.910	4.910	sx_burncpu
4.500	4.500	cputime
3.270	3.270	icputime
3.270	3.270	sigtime_handler
2.300	2.300	underflow
2.280	2.280	real_recurse
2.190	2.190	muldiv
1.980	1.980	my_irand
1.450	1.450	gethrtime
0.910	0.910	bounce_a
0.770	3.590	systime
0.740	0.740	gettimeofday
0.630	0.630	gethrvtime
0.480	0.480	inc_func
0.470	0.470	inc_body
0.470	0.470	inc_brace
0.380	2.360	fitos
0.350	0.350	inline_code

図 5-2　「関数」タブ

「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、上の区画にはこ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。「関数」タブに表示される関数は、「呼び出し元-呼び出し先」タブにも表示できます。

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。包括的メトリックと排他的メトリックのどちらかが表示されている場合には、対応する属性メトリックも表示されます。 表示されるデフォルトメトリックは、「関数リスト」に表示されているメトリックから取られます。

属性メトリックの百分率は、選択されている関数の包括的メトリックに属性メトリックが寄与している割合を示しています。 排他的および包括的メトリックの場合は、プログラムのメトリック全体に占める割合を示す百分率値が表示されます。

プログラムの構造内をナビゲートし、呼び出し元や呼び出し先の区画から関数を選択することによって、高メトリック値を検索することができます。いずれかのタブで新しい関数を選択するたびに、「呼び出し元-呼び出し先」タブが更新され、選択された関数を基準としてセンタリングされます。

呼び出し元リストと呼び出し先リストは、いずれか 1 つの列のデータに基づいてソートされます。この結果、メトリック値が高い関数を簡単に判別できます。ソート列のヘッダテキストは、太字で表示されます。「呼び出し元-呼び出し先」タブでソートメトリックを変更すると、「関数」タブのソートメトリックが変更されます。

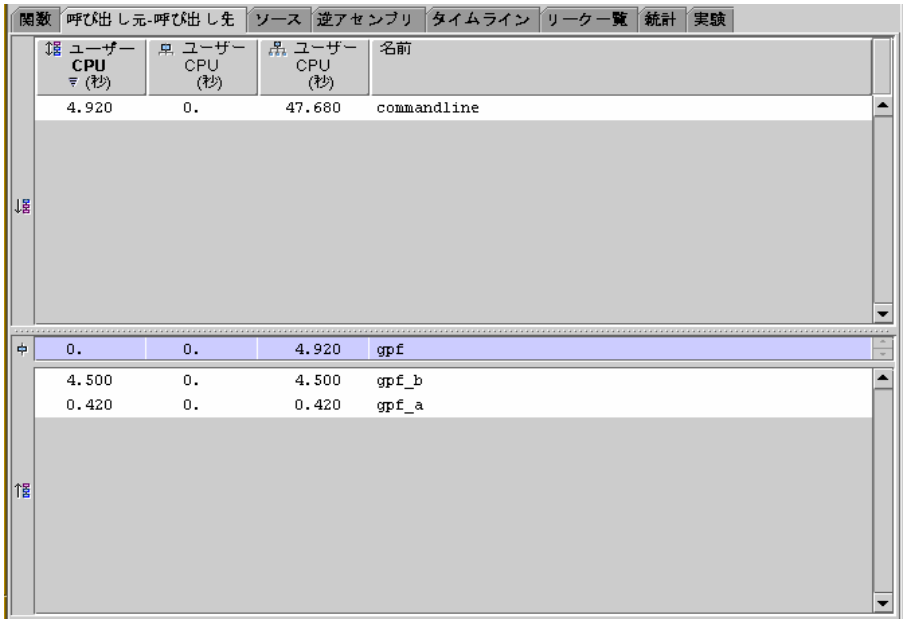


図 5-3　「呼び出し元-呼び出し先」タブ

「ソース」タブ

「ソース」タブには、選択された関数が入っているソースファイルが表示されます。命令の生成対象であるソースファイルの各行に、パフォーマンスメトリックの注釈が付きます。 コンパイラのコメントがある場合には、対応するソース行の上に表示されます。

メトリック値が高い行の場合、メトリックが強調表示されます。高メトリック値とは、ファイル内の任意の行のメトリックの最大値のしきい 百分率を超えるものです。選択した関数のエントリポイントも強調表示されます。

パフォーマンスメトリック、コンパイラコメント、強調表示の選択内容の変更は、「データ表示方法の設定」ダイアログで行います。

コレクタ API を使用して関数情報を提供すると、動的にコンパイルした C や C++ 関数の注釈付きソースコードを表示できますが、表示されるのは、選択した関数のゼロ以外のメトリックだけです。これは、ソースファイル内に他の関数がある場合でも同じです。 Java HotSpot 仮想マシンでコンパイルしたかどうかに関係なく、Java メソッドの注釈付きソースコードを表示することはできません。

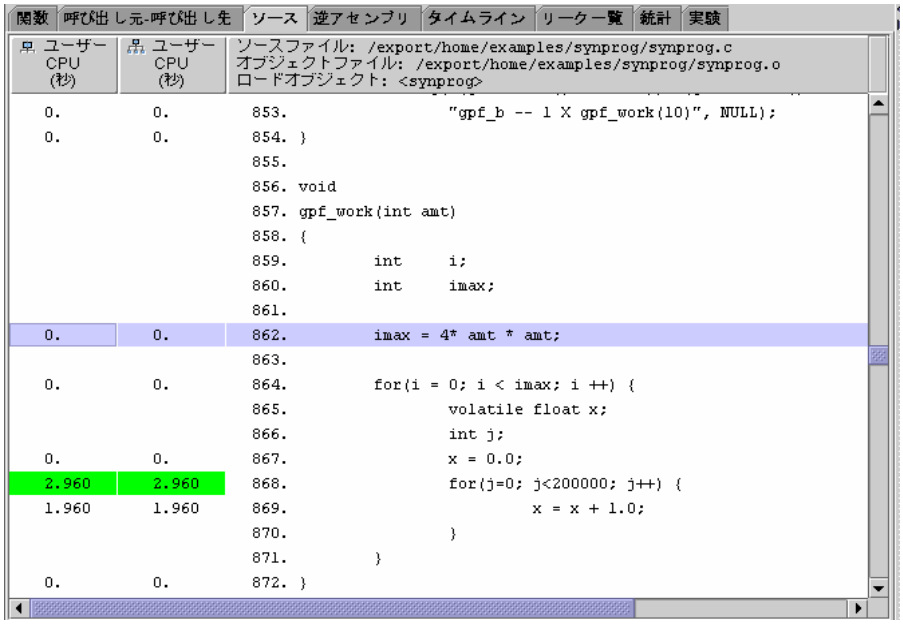


図 5-4　「ソース」タブ

「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。16 進で命令を表示することもできます。

ソースコードがある場合には、ソースコードがリストに挿入されます。それぞれのソース行は、その行が生成する最初の命令の上に表示されます。コンパイラがコードを最適化した結果、命令の順序が変更された場合、ソース行がブロック単位で表示されることがあります。 コンパイラのコメントがある場合には、ソースコードとともに挿入されます。 パフォーマンスメトリックをソースコードに注釈として付けることもできます。

メトリック値が高い行の場合、メトリックが強調表示されます。高メトリック値とは、ファイル内の任意の行のメトリックの最大値のしきい百分率を超えるものです。

パフォーマンスメトリック、コンパイラコメント、強調表示しきい値、ソース注釈、および 16 進表示の選択内容の変更は、「データ表示方法の設定」ダイアログで行います。

選択した関数が動的にコンパイルされた場合、表示されるのはその関数の命令だけです。 コレクタ API を使用して関数情報を提供した場合 (71 ページの「動的な関数とモジュール」 参照)、表示されるのは、指定した関数のゼロ以外のメトリックだけです。これは、ソースファイル内に他の関数がある場合も同じです。 Java コンパイル済みメソッドの命令は、コレクタ API を使用しなくても見ることができます。

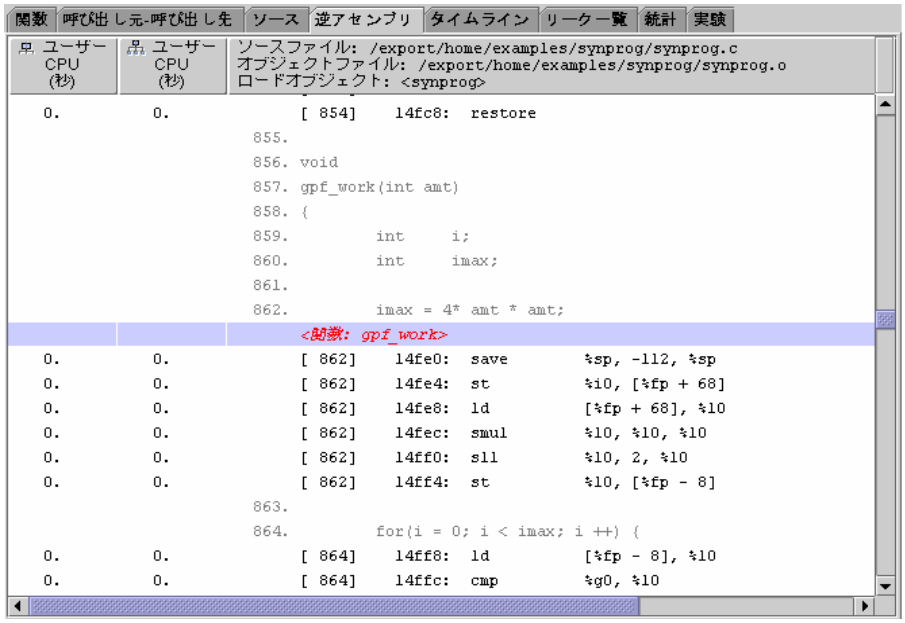


図 5-5 「逆アセンブリ」タブ

「タイムライン」タブ

「タイムライン」タブには、イベントのグラフが時間の関数として表示されます。各 LWP と各実験のサンプルデータとイベントが、集計されず別々に表示されます。「タイムライン」表示により、標本コレクタが記録した個々のイベントを調べることができます。

データは、水平バーに表示されます。1 つの実験がいくつかのバーに表示されます。デフォルトの場合、最上部のバーには標本情報が表示され、各 LWP を表示するバー、各データ型 (時計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)を表示するバーが続き、記録されたイベントがこれらのバーによって表示されます。 各データ型のバーラベルには、*n.m* の形式でデータ型と番号を示すアイコンが入っています。*n* は実験、*m* は LWP を表します。 シス

テムスレッドを実行するためにマルチスレッドプログラムで作成される LWP は「タイムライン」タブでは表示されませんが、その番号は LWP 索引に入っています。 詳細は、160 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。

標本バーには、タイミングメトリックの場合と同じ方法で集計されたプロセス時間が色別に表示されます。 各標本は、各マイクロステートで費やされた時間の割合に基づいて色付きの長方形で表されます。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。標本をクリックすると、「凡例」タブと「概要」タブが選択不可になります。

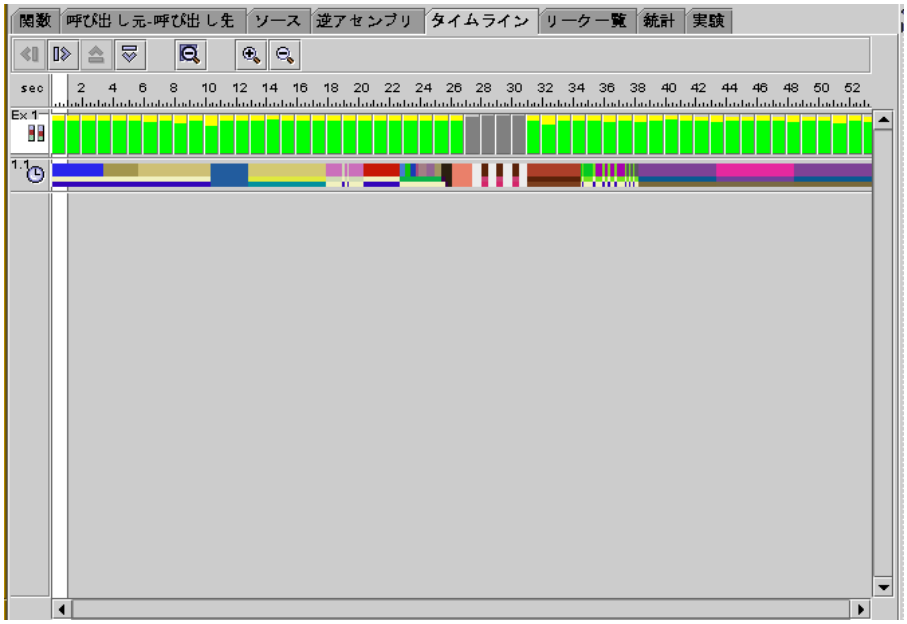


図 5-6　「タイムライン」タブ

他のバーのイベントマーカには、マーカ最上部に表示されるリーフ関数で始まる呼び出しスタックの一部が色別表示されます。イベントマーカ内の色付き長方形をクリックすると、対応する関数が呼び出しスタックの中から選択され、その関数とイベントのデータが「イベント」タブに表示されます。選択された関数は「イベント」タブと「凡例」タブの両方で強調表示され、その名前がメニューバーに表示されます。

「タイムライン」タブを選択すると「イベント」タブが有効になり、選択したイベントの詳細が表示されます。デフォルトの場合、「イベント」タブは「タイムライン」タブを選択すると右の区画に表示されます。 「タイムライン」タブでイベントマーカを選択すると、右の区画にある「凡例」タブが有効になって表示され、関数の色別情報が表示されます。

「リーク一覧」タブ

「リーク一覧」タブには、プログラム内で発生したリークと割り当てのリストが表示されます。各リークエントリは、リークしたバイト数と割り当て用呼び出しスタックで構成されます。各割り当てエントリは、割り当てたバイト数と割り当て用呼び出しスタックで構成されます。

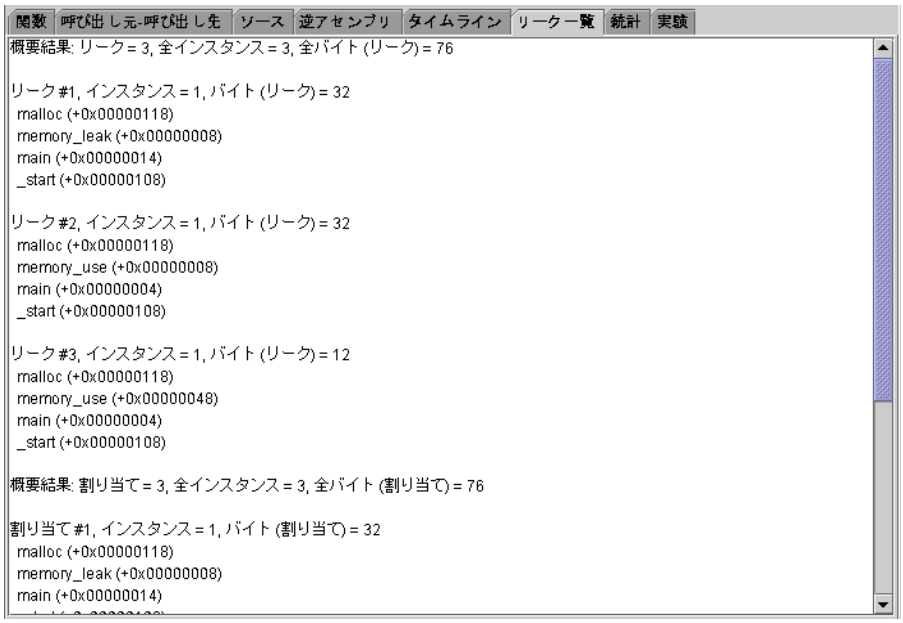


図 5-7　「リーク一覧」タブ

「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合計値のほか、各実験について選択した標本の統計が表示されます。プロセス時間は、メトリックの場合と同じ方法でマイクロステートアカウントについて集計されます。 詳細は、51 ページの「時間データ」 を参照してください。

「統計」タブに表示される統計は、「関数」タブに表示される<合計>関数のタイミングメトリックと原則として一致するはずです。「統計」タブに表示される値は、<合計>のマイクロステートアカウンティング値よりも正確です。 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と「統計」タブの時間値の違いによる寄与要素が含まれます。 これらの寄与要素の発生源は、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 7 および 8 のオペレーティング環境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。

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。

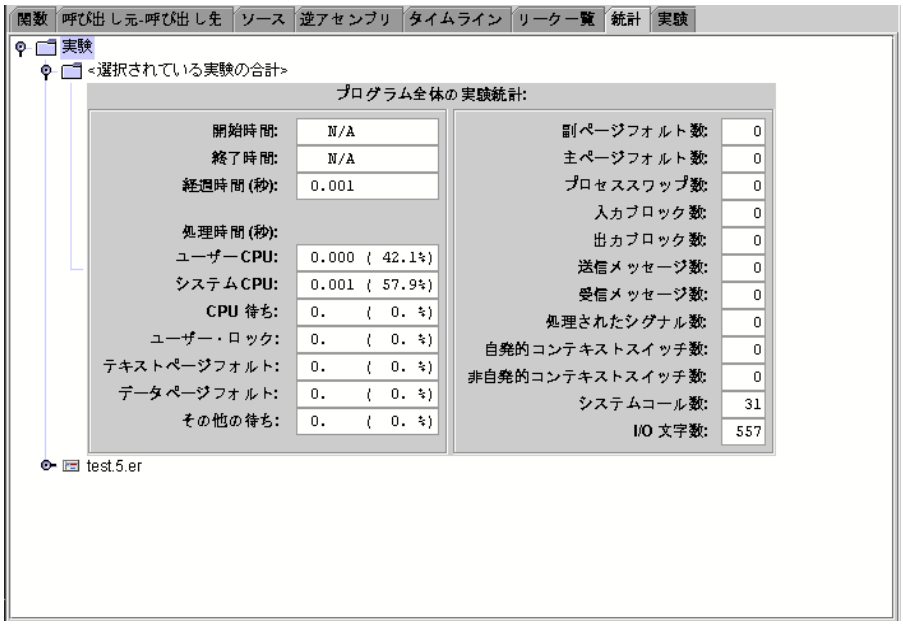


図 5-8　「統計」タブ

「実験」タブ

「実験」タブは、2 つの区画に分割されます。

上の区画には、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。

下の区画には、パフォーマンス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。


関数
呼び出し元-呼び出し先
ソース
逆アセンブリ
タイムライン
リーク一覧
統計
実験
実験
ロードオブジェクト
test.5.er
ターゲットコマンド: `hello'
プロセス pid 7423, ppid 21657, pgrp 7423, sid 21657
コレクタのバージョン: `Forte Developer 7 Performance Analyzer 7.0 2002/03/10'
ホスト `kakusan', OS `SunOS 5.7 ', ページサイズ 8192
実験開始 Thu May 2 15:22:30 2002
データ収集パラメータ:
時間プロファイル, 間隔 = 10 ミリ秒
ヒープトレース
MPI トレース
周期サンプリング, 1 秒
実験サイズ上限, 2000
終了: 0.011840489
エラー/警告ログ:

図 5-9 「実験」タブ

「概要」タブ

「概要」タブの上部には、選択した関数やロードオブジェクトに関する情報が表示されます。情報の内容は、関数やロードオブジェクトの名前、アドレス、およびサイズです。関数の場合には、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も含まれます。「概要」タブの下部には、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。「概要」タブに表示される情報は、メトリックの選択によって影響を受けることがありません。

「概要」タブは、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。


概要 イベント 凡例

選択されている関数/ロードオブジェクトのデータ:

名前:	_start
PC アドレス:	2:0x00000990
サイズ:	288
ソースファイル:	(unknown)
オブジェクトファイル:	(unknown)
ロードオブジェクト:	<hello>
符号化された名前:	
エイリアス:	

処理時間 (秒) / 回数

	排他的	包括的
ユーザーCPU:	0. (0. %)	0. (0. %)
ウォール:	0. (0. %)	0. (0. %)
全 LWP:	0. (0. %)	0. (0. %)
システムCPU:	0. (0. %)	0. (0. %)
CPU 待ち:	0. (0. %)	0. (0. %)
ユーザーロック:	0. (0. %)	0. (0. %)
テキストページフォルト:	0. (0. %)	0. (0. %)
データページフォルト:	0. (0. %)	0. (0. %)
他の待ち:	0. (0. %)	0. (0. %)
割り当てられたバイト:	0 (0. %)	76 (100.0%)
割り当て:	0 (0. %)	3 (100.0%)
リークしたバイト:	0 (0. %)	76 (100.0%)
リーク:	0 (0. %)	3 (100.0%)
MPI:	0. (0. %)	0. (0. %)
送信した MPI バイト:	0 (0. %)	0 (0. %)
MPI の送信:	0 (0. %)	0 (0. %)
受信した MPI バイト:	0 (0. %)	0 (0. %)
MPI の受信:	0 (0. %)	0 (0. %)
他の MPI 呼び出し:	0 (0. %)	0 (0. %)

図 5-10 「概要」タブ

「イベント」タブ

「イベント」タブには、イベントタイプ、リーフ関数、LWP ID、スレッド ID、CPU ID といった、選択したイベントに関するデータが表示されます。 データパネルの下には、呼び出しスタック内の各関数のイベントマーカで使用される色別で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。

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、およびタイミングメトリックのリストが「イベント」タブに表示されます。各タイミングメトリックについて、消費時間量と色が表示されます。標本のタイミング情報は、時計プロファイルで記録されるタイミング情報よりも正確です。

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。

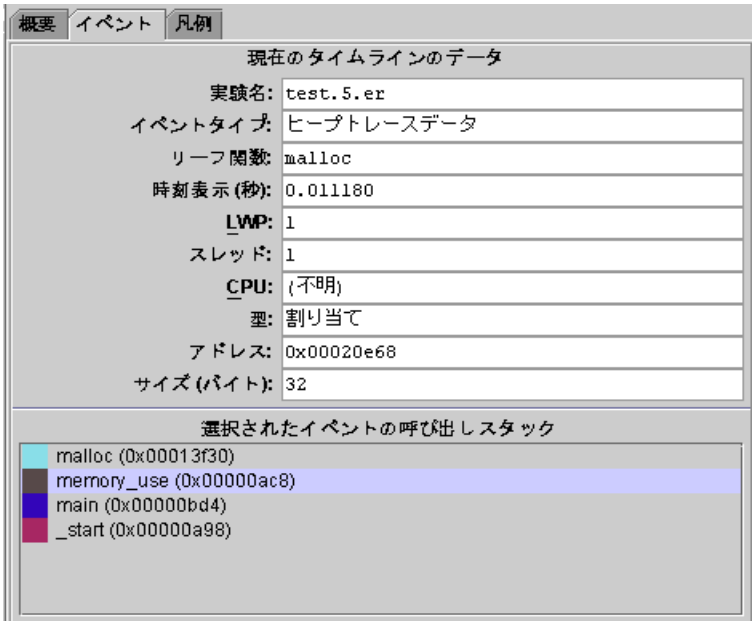


図 5-11　イベントデータを表示する「イベント」タブ

図 5-12 標本データを表示する「イベント」タブ

「凡例」タブ

「凡例」タブには、「タイムライン」タブでのイベントの表示に使用する色と関数のマップが表示されます。「凡例」タブが有効となるのは、「タイムライン」タブでイベントが選択されているときだけです。「タイムライン」タブで標本が選択されている場合には、「凡例」タブはディム選択不可になります。「タイムライン」メニューでカラーチューザを使用すれば、色を変更できます。

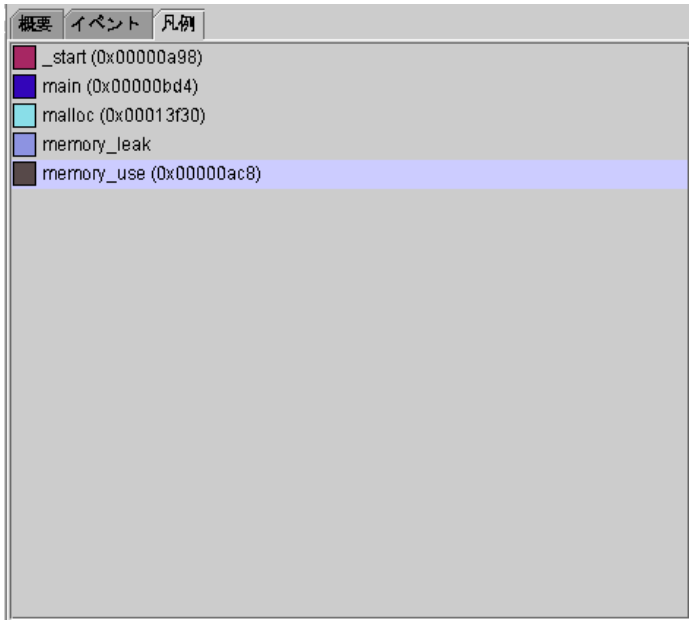


図 5-13 「凡例」タブ

パフォーマンスアナライザの使用法

ここでは、パフォーマンスアナライザの機能の一部と、そのディスプレイの設定方法について、説明します。

メトリックの比較

パフォーマンスアナライザは、読み込まれたデータに関する一組のパフォーマンスメトリックを計算します。このデータは、1 つまたは複数の実験から取り込むことも、事前に定義された実験グループから取り込むこともできます。

同じ 1 つの組に含まれている一部のメトリック群を比較するには、メニューバーから「ファイル」>「新規ウィンドウを開く」を選択し、新しいアナライザウィンドウを表示します。 このウィンドウを閉じるには、新しく開いたウィンドウ内のメニューバーから「ファイル」>「閉じる」を選択します。

複数組のメトリックを計算して表示するには (たとえば、2 つの実験を比較する場合など)、それぞれにパフォーマンスアナライザを起動する必要があります。

実験の選択

パフォーマンスアナライザでは、1 つまたは複数の実験のメトリックを計算することも、事前に定義された実験グループのメトリックを計算することもできます。この節では、パフォーマンスアナライザへの実験の読み込み、実験の追加、パフォーマンスアナライザからの実験の解除について説明します。

実験を開く 実験を開くとパフォーマンスアナライザから実験データのすべてがクリアーされ、新しい一組のデータが読み込まれます(ディスクに格納されている実験データが、この消去の影響を受けることはありません)。

実験の追加 パフォーマンスアナライザに実験を追加すると、パフォーマンスアナライザ内の新しい記憶場所に一組のデータが読み込まれ、すべてのメトリックが再計算されます。各実験のデータは別々に格納されますが、表示するときはすべての実験のメトリックが結合されます。この機能は、異なる実行セッションで同じプログラムのデータを記録する必要がある場合、たとえば、同じプログラムについて、タイミングデータとハードウェアカウントデータの両方が必要な場合などに便利です。

MPI の実行で収集されたデータを調べるには、パフォーマンスアナライザで 1 つの実験を開き、別の実験を追加します。 これによって、すべての MPI プロセスの全体のデータを見ることができます。このことは、実験グループを定義していて、その実験グループを読み込んだときにも当てはまります。

実験の解除 実験を解除すると、パフォーマンスアナライザからその実験のデータがクリアーされ、メトリックが再計算されます(実験ファイルそのものが、このクリアーの影響を受けることはありません)。

実験グループを読み込んでいる場合、個々に実験を解除することはできますが、グループ全体を解除することはできません。

表示するデータの選択

パフォーマンスアナライザに読み込んだ実験データの内のどれを表示するかは、さまざまな方法によって選択できます。

メトリックの選択 表示するメトリックとソートメトリックを選択するには、「データ表示方法の設定」ダイアログの「メトリック」タブと「ソート」タブを使用します。メトリックの選択結果は、すべてのタブに適用されます。「呼び出し元-呼び出し先」

タブでは、表示対象として選択されたメトリックすべてについて属性メトリックが追加されます。「データ表示方法の設定」ダイアログボックスを開くには、次のツールバーボタンを使用します。

すべてのメトリックを、秒単位の時間または回数のいずれかと、プログラム全体のメトリックに占める割合 (百分率) の形式で表示できます。また、循環型のハードウェアカウンタメトリックは、時間、回数、百分率の形式で表示できます。

「ソース」タブと「逆アセンブリ」タブの設定 「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」タブから、高メトリック値の強調表示のしきい値やコンパイラコメントのクラスを選択できます。また、注釈付きソースコードのメトリックを表示するかどうか、および注釈付き逆アセンブリリスト内の命令の 16 進コードを表示するかどうかなどを選択できます。

実験、LWP、スレッド、標本による選別 メトリックを表示する特定の実験、標本、スレッド、および LWP だけを指定することによって、パフォーマンスアナライザが表示する情報を制御できます。「データをフィルタ」ダイアログを使用して選択します。スレッドによる選択と標本による選択は、「タイムライン」表示に適用されません。「データをフィルタ」ダイアログボックスを開くには、次のツールバーボタンを使用します。

関数の表示と非表示「関数の表示/非表示」ダイアログを使用すれば、各関数のメトリックを別々に表示するか、あるいはロードオブジェクトのメトリックを一括表示するかを、ロードオブジェクトごとに選択できます。「関数の表示/非表示」ダイアログボックスを開くには、次のツールバーボタンを使用します。

デフォルトの設定

すべてのデータ表示のデフォルト設定は、デフォルト値ファイルによって決定されます。このファイルを編集して独自のデフォルト値を設定することができます。

デフォルトのメトリックはデフォルト値ファイルから読み取られます。ユーザーのデフォルト値ファイルが存在しない場合は、システムのデフォルト値ファイルが読み取られます。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。 ユーザーのホームディレクトリ内のデフォルト値ファイルは、パフォーマンスアナライザが起動されるたびに読み取られ、それ以外のディレクトリ内のデフォルト値ファイルは、そのディレクトリからパフォーマンスアナライザが起動されたときに読み取られます。 ユーザーのデフォルト値ファイルは、ファイル名を `.er.rc` にする必要があり、`er_print` コマンドを含めることができます。 詳細は、141 ページの「デフォルト値関連のコマンド」 を参照してください。デフォルト値ファイルには、表示する一群のメトリックとその表示順序、ソート基準にするメトリックを指定できます。 下表は、メトリックのシステムデフォルト設定値を示しています。

表 5-2 「関数」タブに表示されるデフォルトメトリック

データ型	デフォルトメトリック
時間ベースのプロファイル	包括的および排他的ユーザー CPU 時間
ハードウェアカウンタオーバーフローのプロファイル	包括的および排他的時間 (周期数を数えるカウンタ) またはイベントカウント (その他のカウンタ)
同期遅延トレース	包括的 Sync 待ちカウントと包括的 Sync 遅延時間
ヒープトレース	包括的リークと包括的リークバイト数
MPI トレース	包括的 MPI 時間、包括的送信した MPI バイト、包括的 MPI の送信、包括的受信した MPI バイト、包括的 MPI の受信、および包括的 MPI その他

表示される関数またはロードオブジェクトのメトリックそれぞれについて、システムデフォルトは秒単位またはカウント単位で値を選択します。行は、デフォルト値リスト内の先頭のメトリックを基準にソートされます。

C++ プログラムの場合は、関数名を長い形式または短い形式のどちらの形式でも表示できます。デフォルトは長形式です。 この選択は、デフォルト値ファイルで行うこともできます。

「データ表示方法の設定」ダイアログボックスで行った設定内容は、デフォルト値ファイルに保存できます。

デフォルト値ファイルとデフォルト値ファイルで使用できるコマンドについては、141 ページの「デフォルト値関連のコマンド」 を参照してください。

名前またはメトリック値の検索

「検索」ツール パフォーマンスアナライザのツールバーには、「関数」タブと「呼び出し元-呼び出し先」タブの「名前」列で、また「ソース」タブと「逆アセンブリ」タブのコード列で、テキスト検索に使用できる「検索」ツールがあります。 「ソース」タブと「逆アセンブリ」タブでの高メトリック値を検索するときにも、「検索」ツールを利用できます。 ソースファイル内の最大値の指定しきい値を超えた高メトリック値は、強調表示されます。 強調表示しきい値の選択方法については、121 ページの「表示するデータの選択」 を参照してください。

マップファイルの作成と利用

パフォーマンスアナライザでは、実験のパフォーマンスデータを使用してマップファイルを作成できます。 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命令キャッシュ動作の効率化を図ることができます。マップファイルは、関数の読み込み順に関する情報をリンカーに提供します。

マップファイルを作成するには、`-g` オプションまたは `-xF` オプションを使用してプログラムをコンパイルする必要があります。これらのオプションにより、必要なシンボルテーブルがオブジェクトファイルに挿入されます。

マップファイル内の関数の順序は、メトリックソート順序によって決まります。 特定のメトリックに基づいて関数の順序を決めるには、そのメトリックに対応するパフォーマンスデータを収集する必要があります。 メトリックを慎重に選択してください。 デフォルトメトリックが常に最善であるとは限りません。ヒープトレースデータを記録する場合には、デフォルトメトリックは非常に不適切です。

マップファイルを使用してプログラムの順序を変更するには、`-xF` オプションを使用してプログラムをコンパイルする必要があります。このオプションを使用すると、コンパイラは個々に配置変更できる関数を生成し、プログラムを `-M` オプションと結び付けます。

```
% compiler-name -xF -c source-file-list
% compiler-name -M mapfile-name -o program-name object-file-list
```

第6章

コマンド行パフォーマンス解析ツール er_print

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、ファイルやプリンタ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。標本コレクタを使用して実験ファイルにデータを保存しておくことによって、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、アドレス空間データ、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
- 関数リスト関連のコマンド
- 呼び出し元と呼び出し先リスト関連のコマンド
- ソースおよび逆アセンブリコードリスト関連のコマンド
- メモリー割り当てリスト関連のコマンド
- フィルタ関連のコマンド
- メトリックリスト関連のコマンド
- デフォルト値関連のコマンド
- 出力関連のコマンド
- その他の表示関連のコマンド
- マップファイル作成コマンド
- 制御関連のコマンド
- 情報関連のコマンド
- サポートが中止されたコマンド

コレクタが収集するデータについては、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 5 章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 6-1 に、er_print のオプションをまとめます。

表 6-1　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 6-2 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 6-2　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 6-3 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 6-3 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。
%	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 "."、"+"、"%" 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、`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 6-4 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、52 ページの「ハードウェアカウンタのオーバーフローデータ」 を参照してください。

表 6-4　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	全 LWP 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	その他の待ち時間
同期遅延メトリック	sync	Sync 待ち時間
	syncn	Sync 待ち回数
メモリー割り当てメトリック	alloc	割り当て数
	balloc	割り当てられたバイト数
	leak	リーク数
	bleak	リークしたバイト数
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
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 6-4 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。

関数リスト関連のコマンド

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で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すべて、および非表示の関数を持つロードオブジェクトが含まれます。詳細は、functions コマンドを参照してください。

出力する行数は、limit コマンドを使用して制限できます (143 ページの「出力関連のコマンド」 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、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。

fsummary

関数リスト内のすべての関数ついて、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 (143 ページの「出力関連のコマンド」 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。

`metrics` *metric-list*

関数リストに表示するメトリックを指定します。*metric-list* 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` コマンドが終了すると、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。

```
現在: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、127 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。

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 (143 ページの「出力関連のコマンド」 を参照)。

sort *metric-keyword*

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-keyword* は、127 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。

呼び出し元と呼び出し先リスト関連のコマンド

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 (143 ページの「出力関連のコマンド」 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。

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。

cmetrics *metric-list*

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-list* 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。

```
現在: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、127 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。

csort *metric-keyword*

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-keyword* は、127 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。

ソースおよび逆アセンブリコードリスト関連のコマンド

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。

source | src { *file* | *function* } [*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*) のない曖昧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。

disasm { *file* | *function* } [*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 *class-list*

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。

- b[asic] - 基本レベルのメッセージを表示します。
- v[ersion] - ソースファイル名、最終修正日付、コンパイラコンポーネントのバージョン、コンパイル日付とオプションなどのバージョンメッセージを表示します。
- pa[rallel] - 並列化に関するメッセージを表示します。
- q[uery] - 最適化に影響するコードに関する問い合わせメッセージを表示します。
- l[oop] - ループの最適化と変換に関するメッセージを表示します。
- pi[pe] - ループのパイプライン化に関するメッセージを表示します。
- i[nline] - 関数のインライン化に関するメッセージを表示します。
- m[emops] - ロード、ストア、プリフェッチなどのメモリー操作に関するメッセージを表示します。
- f[e] - フロントエンドのメッセージを表示します。
- all - すべてのメッセージを表示します。

■ none - メッセージを表示しません。

all および none クラスは常に単独で指定します。

scc コマンドを省略した場合は、basic がデフォルトのクラスになります。class-list が空の 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。

互換性のために必要であれば、強調表示しきい値を t[hreshold]=*nn* の形式で指定することもできます。*nn* は、しきい値の百分率です。詳細は、sthresh の解説を参照してください。

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 *class-list*

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。

■ h[ex] - 命令の 16 進値を表示します。

■ 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。

■ 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。

互換性の面で必要であれば、強調表示しきい値を t[hreshold]=*nn* の形式で指定することもできます。*nn* は、しきい値の百分率です。 詳細は、dthresh の解説を参照してください。

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。

メモリー割り当てリスト関連のコマンド

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。

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。

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。

フィルタ関連のコマンド

ここでは、表示する実験、標本、スレッド、LWP を選択したり、現在の選択内容を一覧表示したりするコマンドを説明します。

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、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。

`lwp_select` *lwp-selection*

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。

`sample_select` *sample-selection*

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。

`thread_select` *thread-selection*

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。

`object_select` *object-list*

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。

選択内容の一覧表示

この節では、選択内容を一覧表示するためのコマンドを示し、その後でいくつかの例を紹介します。

`exp_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。

`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。

object_list

ロードオブジェクトの一覧を表示します。 ロードオブジェクトの関数が関数リストに表示されている場合にはロードオブジェクトの名前の前に "+" が付き、関数が関数リストに表示されていない場合には "-" が付きます。

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。

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。

以下は、実験リストの表示例です。

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

標本、スレッド、LWP の一覧も、これと同じ形式で表示されます。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
```

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 ロードオブジェクト
=== ================================================
yes /tmp/var/synprog/synprog
yes /opt/SUNWspro/lib/libcollector.so
はい /usr/lib/libdl.so.1
はい /usr/lib/libc.so.1
```

メトリックリスト関連のコマンド

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。

デフォルト値関連のコマンド

ここでは、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ためのコマンドを説明します。 これらのコマンドは、デフォルト値の設定に使用できるだけであり、er_print に対する入力で使用することはできません。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。

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-list*

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、127 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-list*

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-list* のエントリに表示文字列 "!" 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-list* 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、127 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-name*

C++ の関数名を復号化する API をサポートする共有オブジェクトへのパスを設定します。 この共有オブジェクトは、GNU 標準の `libiberty.so` インタフェースに適合していて、C 関数の `cplus_demangle()` をエクスポートする必要があります。

出力関連のコマンド

ここでは、`er_print` の出力を制御するコマンドを説明します。

`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++ のみ) 。

`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。

その他の表示関連のコマンド

`header` *experiment-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_list` コマンドを使用して調べることができます。*experiment-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。

experiment-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。

`overview` *experiment-ID*

指定した実験の標本のうち、現在選択されている標本各々の標本データを出力します。*experiment-ID* は、`exp_list` コマンドを使用して調べることができます。*experiment-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。 *experiment-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` *experiment-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環境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。

experiment-ID は、`exp_list` コマンドを使用して調べることができます。*experiment-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。*experiment-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。

experiment-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。

マップファイル作成コマンド

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。

制御関連のコマンド

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。

script *script*

script 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。

情報関連のコマンド

help

er_print コマンドの一覧を表示します。

{ Version | version }

現在の er_print のバージョン情報を表示します。

サポートが中止されたコマンド

address_space

アドレス空間データ収集と表示のサポートは停止されました。今回のリリースからは、このコマンドは無視されて警告が出されます。

osummary

ロードオブジェクトリストは、関数リストに組み込まれました。 ロードオブジェクトのメトリックを表示するには、`object_select` コマンドと `fsummary` コマンドを使用してください。今回のリリースからは、このコマンドは無視されて警告が出されます。

第7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。

- 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
- 注釈付きコードリスト

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、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、それぞれの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。プロファイル間隔の最後で各 LWP のマイクロステートのカウント値は 1 インクリメントされ、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、LWP がユーザーモードで CPU を使用した場合だけです。プロファイル信号がスケジューリングされたときに LWP がユーザーモードの場合、ユーザー CPU 状態の配列要素は 1 であり、その他のすべての状態の配列要素は 0 になります。 LWP がユーザーモードでない場合は、次回 LWP がユーザーモードになったときにデータが記録され、配列には、さまざまな状態のカウント値の累計値が含まれます。

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 LWP がユーザーモードでない場合は、LWP 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 7-1 に、各マイクロステートとメトリックの対応関係をまとめます。

表 7-1　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	その他の待ち時間
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	その他の待ち時間
LMS_STOPPED	停止 (`/proc`、ジョブ制御、`lwp_stop` のいずれか)	その他の待ち時間
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、小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他に、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。

- LWP 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 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。

- LWP 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 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 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。LWP を実行するプロセッサの個数より実行する LWP 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、LWP 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。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。

- 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。

これの不正確さの他にも、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 1% 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全 LWP 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。

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。 これらの寄与要素の発生源は、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™ 7 および 8 のオペレーティング環境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

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、「Sync 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「Sync 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。 同期イベント待ちのスレッドがあるときにすべての LWP がビジーである場合、「ユーザーロック時間」はゼロになりますが、「Sync 待ち時間」はゼロにはなりません。 これは、スレッドが動作している LWP ではなく、特定のスレッドに同期待ち時間が関連付けられているためです。

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。

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。

ハードウェアカウンタオーバーフローのプロファイリング

ハードウェアカウンタオーバーフローのプロファイルデータには、カウンタ ID とオーバーフロー値が含まれます。この値は、カウンタがオーバーフローするように設定されている値よりも大きくなることがあります。 これは、オーバーフローが発生して、そのイベントが記録されるまでの間に命令が実行されるためです。このことは、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。 詳細は、181 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。

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。

MPI トレース

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プロセス各々と通信する。

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)。

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、166 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最も単純なものは、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 `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、LWP はシステムモードになります。 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

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 オペレーティング環境の一部ではなく、Solaris 8 オペレーティング環境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 7 と Solaris 8 のオペレーティング環境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 オペレーティング環境のスレッドライブラリと Solaris 8 オペレーティング環境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。

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、Forte™ Developer 7 コンパイラ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。Forte Developer 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。

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 7 と Solaris 8 のオペレーティング環境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 オペレーティング環境のスレッドライブラリと Solaris 8 オペレーティング環境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。

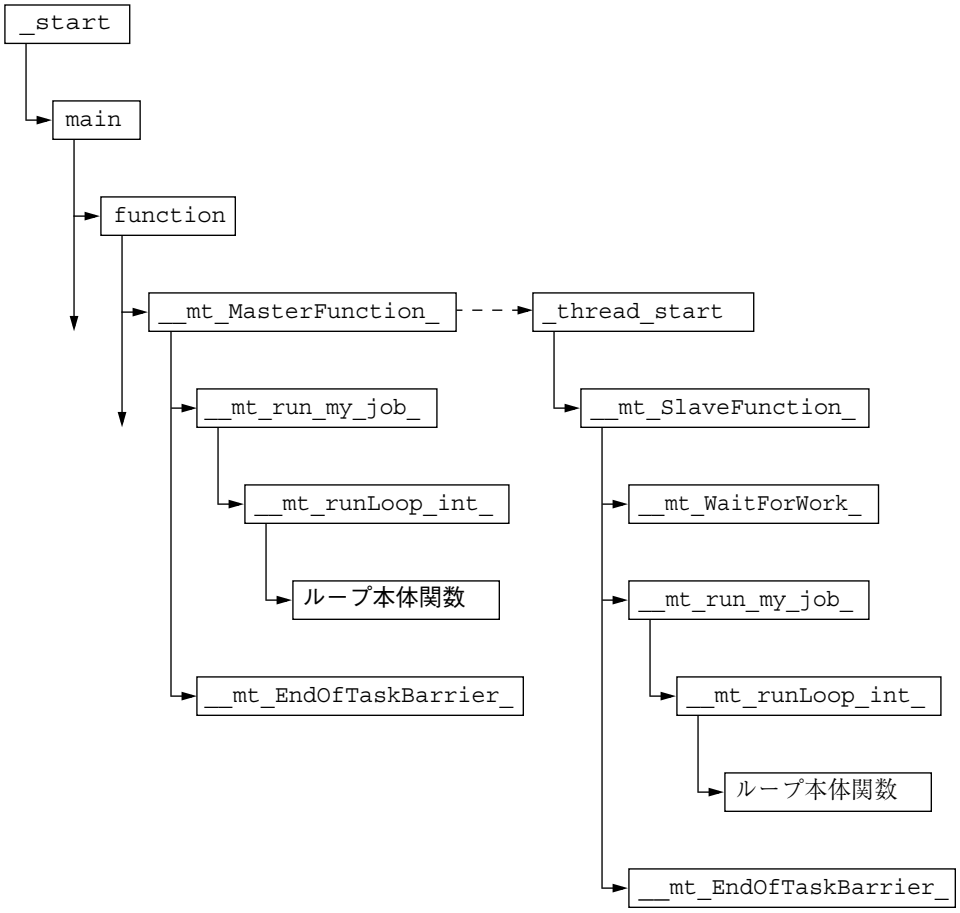


図 7-1　Parallel Do または Parallel For 構造を含むマルチスレッドプログラムの呼び出しツリー

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 7-1 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 7-2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 7-1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。

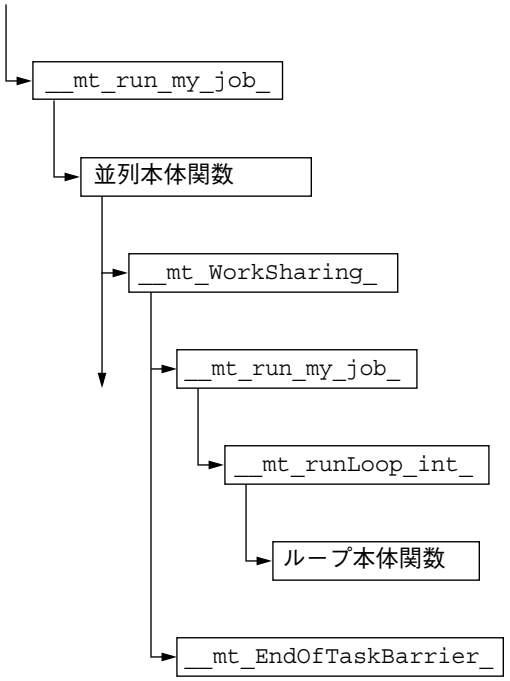


図 7-2　Worksharing Do または Worksharing For 構造を含む並列領域の呼び出しツリー

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 (63 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 7 と 8 のオペレーティング環境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。

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

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ず、<合計>は記録された PC を持つ最後の関数の呼び出し元として表示されます。

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、同じ関数に異なる 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も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の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、-g0 を指定すると有効になります。

- 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、160 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、161 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。

アウトライン関数

フィードバックの最適化で、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、通常は実行対象とみなされないコードです。具体的には、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。 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。

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、71 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。

<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、71 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。この問題について最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。

<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他に、<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>関数はその他の関数の呼び出し元として表示される可能性があります。

注釈付きコードリスト

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ます。この節では、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

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リソース消費状況を示します。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 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
- 実行可能ファイルに記録されている絶対パス名
- 現在の作業ディレクトリ

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最も問題となるのは、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 7-2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 7-2　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、命令に不正な行番号が付けられている。

表 7-2　注釈付きソースコードのメトリック(続き)

メトリック	意味
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、スレッド同期用に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	この行が原因のすべての PC のメトリックの合計がゼロ以外の数値になりました。

コンパイラのコメント

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。

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。

<不明> 行

PC に対応するソース行を特定できない場合、その PC のメトリックは常に、注釈付きソースファイルの最初に挿入される特別なソース行に原因があるとされます。このソース行のメトリックが高いということは、オブジェクトモジュールのコードの一部にマップが行われていない行があることを示します。こうした場合は、注釈付き逆アセンブリコードが、マップのない命令が行っている処理を調べるのに役立つことがあります。

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。

並列化指令

並列化指令を含むコードからコンパイラが本体関数を生成した場合、並列 loop や section の包括的メトリックは並列化指令に対応します。なぜならば、この行が、コンパイラ生成本体関数の呼び出しサイトであるからです。 包括的メトリックと排他的メトリックは、コードの loops または sections にも現われます。 これらのメトリックは、並列化指令の包括的メトリックに加算されます。

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 7-2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。

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われます。 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コードを理解するのは簡単ではありません。リーフ PC は、次に実行する命令のアドレスです。 このため、命令が原因のメトリックは、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。

プログラムリンケージテーブル (PLT) 命令

1 つのロードオブジェクトの関数が別の共有オブジェクトの関数を呼び出した場合、実際の呼び出しは PLT の 3 つの命令のシーケンスにまず送られ、次に実際の宛先に送られます。アナライザは PLT に対応する PC を削除し、これらの PC のメトリックを呼び出し命令に割り当てます。したがって、呼び出し命令のメトリックが予想以上に高い場合には、呼び出し命令ではなく PLT 命令が原因となっていることが考えられます。 156 ページの「共有オブジェクト間の関数の呼び出し」 も参照してください。

第8章

実験の操作と注釈付きコードリストの表示

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
- `er_src` による注釈付きコードリストの表示
- 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隠し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 コマンドを使用することはできません。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、これらのコマンド `er_cp`(1)、`er_mv`(1)、`er_rm`(1) を説明します。

表示可能な実験ファイルには、実験が作成されたときに、実験への絶対パスが書き込まれます。実験を移動するときに、これらのユーティリティを使用せずにパスを変更すると、実験ファイル内のパスが実際の実験の格納場所と一致しなくなります。この後、新しい格納場所にある実験に対してアナライザや `er_print` を実行すると、パスが無効であるために実験が見つからなかったり、異なる実験が選択されたりすることになります (古い格納場所で新しい実験が作成された場合)。これらのユーティリティは、実験をコピーまたは移動するときに実験名からパスを削除します。

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 [-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。

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。

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

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。-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。

er_src による注釈付きコードリストの表示

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 [ options ] object item tag
```

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。 オブジェクトファイルを指定した場合、このオプションは省略できます。

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。必要がなければ、このオプションは省略できます。必要があるにもかかわらず、省略した場合は、その候補を示すメッセージが表示されます。

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。

-c *commentary-classes*

表示するコンパイラのコメントクラスを指定します。 *commentary-classes* は、コロンで区切ったクラスのリストです。 これらのクラスについては、134 ページの「ソースおよび逆アセンブリコードリスト関連のコマンド」 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、141 ページの「デフォルト値関連のコマンド」 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。

-d

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。

-o *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、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。

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 [-q] [-F] [-V] experiment
```

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。

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。

-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

現在の er_export のバージョン情報を表示します。

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™ オペレーティング環境に付属しています。`tcov` は、Forte™ Developer 製品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 4 章および第 5 章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`	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`	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
- 拡張 tcov プロファイル用の共有ライブラリの作成

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 - 関数名

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	Name
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() 関数に最も実行時間が費やされていることが分かります。 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分かりませんが、次節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 プラットフォームでは、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。 例については、14 ページの「メトリックの対応と gprof の誤った推論」 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、呼び出しもとと呼び出し先の組です。

注 - Solaris 7 および 8 プラットフォームでは、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 回の呼び出しの他に、自身を 10,392 回呼び出します。
- `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。最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 (205 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。既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

記号

A

C

D

E

F

G

J

L

M

O

P

S

T

あ

い

え

お

か

さ

し

す

せ

そ

ち

て

と

に

は

ひ

ふ

へ

ほ

ま

み