プログラムのパフォーマンス解析

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

Part No. 816-0886-01
2001 年 8 月 Revision A

原典： *Analyzing Program Performance With Sun WorkShop (Forte Developer 6 update 2)*
Part No: 806-7989-10
Revision A

製品名の変更について

Sun は新しい開発製品戦略の一環として、Sun の開発ツール群の製品名を Sun WorkShop™ から Forte™ Developer に変更いたしました。製品自体の内容に変更はなく、従来通りの高品質をお届けいたします。

これまでの Sun の主力製品である基本プログラミングツールに、Forte Fusion™ や Forte™ for Java™ といった Forte 開発ツールの得意とする、マルチプラットフォームおよびビジネスアプリケーション実装の機能を盛り込むことで、より広範囲できめ細かな製品ラインが完成されました。

WorkShop 5.0 で使用されていた名称と、Forte Developer 6 で使用される新しい名称の対応については、以下の表をご覧ください。

旧名称	新名称
Sun Visual WorkShop™ C++	Forte™ C++ Enterprise Edition 6
Sun Visual WorkShop™ C++ Personal Edition	Forte™ C++ Personal Edition 6
Sun Performance WorkShop™ Fortran	Forte™ for High Performance Computing 6
Sun Performance WorkShop™ Fortran Personal Edition	Forte™ Fortran Desktop Edition 6
Sun WorkShop Professional™ C	Forte™ C 6
Sun WorkShop™ University Edition	Forte™ Developer University Edition 6

製品名の変更に加えて、次の 2 つの製品について大きな変更があります。

- Forte for High Performance Computing には Sun Performance WorkShop Fortran に含まれていたすべてのツール、および C++ コンパイラが含まれます。したがって、High Performance Computing のユーザーは開発用に 1 つの製品だけを購入すれば済むことになります。
- Forte Fortran Desktop Edition は以前の Sun Performance WorkShop Personal Edition と同じです。ただし、この製品に含まれる Fortran コンパイラでは、自動並列化されたコード、および明示的な指令に基づいた並列コードは生成できません。この機能は Forte for High Performance Computing に含まれる Fortran コンパイラでは使用できます。

Sun の開発製品を引き続きご利用いただきましてありがとうございます。今後もみなさまのご要望にお応えする製品をお届けできるよう努力してまいります。

目次

図目次

表目次

はじめに

このマニュアルでは、Sun WorkShop™ 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。

- 標本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++ のいずれかのプログラミング言語と Sun WorkShop 統合プログラミング環境、 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。プロファイリングツールの `prof`、`gprof`、`tcov` を利用する上で、Sun WorkShop 統合プログラミング環境に関する実用的な知識は必要ありません。

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

第 4 章では、パフォーマンスアナライザのグラフィカルインタフェースを使用し、標本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 5 章では、`er_print` コマンド行インタフェースを使用し、標本コレクタが収集したデータを解析する方法を説明しています。

第 6 章では、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 7 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」

- 小さい三角(△)は意味のある空白を示します。

```
ΔΔ36.001
```

- FORTRAN 77 の例はタブ書式で、Fortran 95 の例は自由書式で示します。FORTRAN 77 と Fortran 95 に共通の例は、特に明示がない限りタブ書式で示します。
- FORTRAN 77 規格では、「FORTRAN」とすべて大文字で表記する旧表記規則を使用しています。サンのマニュアルでは、FORTRAN と Fortran の両方を使用しています。現在の表記規則では、「Fortran 95」と小文字を使用しています。
- オンラインマニュアル (man) ページへの参照は、トピック名とセクション番号とともに表示されます。たとえば、GETENV への参照は、`getenv(3F)` と表示されます。`getenv(3F)` とは、このページにアクセスするためのコマンドが `man -s 3F getenv` であるという意味です。

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)	`#`

サポートしているプラットフォーム

この Sun WorkShop™ リリースでは、Solaris™ SPARC™ プラットフォーム版と Solaris™ Intel プラットフォーム版をオペレーティング環境とするバージョン 2.6、7、および 8 をサポートしています。

Sun WorkShop の開発ツールとマニュアルページへのアクセス

Sun WorkShop 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ていません。SunWorkShop 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 SunWorkshop コンポーネントディレクトリを必要とします。SunWorkshop マニュアルページにアクセスするには、PATH 環境変数に SunWorkshop マニュアルページが必要です。

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、『Sun WorkShop 6 update 2 インストールガイド』を参照するか、システム管理者にお問い合わせください。

注 – この節に記載されている情報は Sun WorkShop 6 update 2 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。 Sun WorkShop 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。

Sun WorkShop 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 Sun WorkShop 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 Sun WorkShop 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 Sun WorkShop 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

Sun WorkShop マニュアルページへのアクセス方法

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

MANPATH 環境変数を設定する必要があるかどうか判断するには

1. 次のように入力して、workshop マニュアルページを表示します。

```
% man workshop
```

2. 出力された場合、内容を確認します。

   workshop(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 Sun WorkShop 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

Sun WorkShop マニュアルへのアクセス

Sun WorkShop 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、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。

  Netscape™ Communicator 4.0 または互換性がある Netscape バージョンのブラウザで次のファイルにポイントします。

  `/opt/SUNWspro/docs/ja/index.html`

  製品ソフトウェアが `/opt` ディレクトリにインストールされていない場合は、システム上でこのディレクトリに相当するパスをシステム管理者に問い合わせてください。

- マニュアルは、docs.sun.com の Web サイトで入手できます。

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。

関連マニュアル

次の表では、docs.sun.com の Web サイトで利用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。

Sun のマニュアルの注文

製品マニュアルはdocs.sun.com Web サイトまたは Fatbrain.com インターネットブックストアを通じて米国 Sun Microsystems,Inc. に直接注文できます。
Fatbrain.com の Sun Documentation Center へは次の URL でアクセスできます。

```
http://www.fatbrain.com/documentation/sun
```

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

パフォーマンスプロファイリングと解析ツールの概要

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、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、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、標本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。

- 標本コレクタは、プロファイリングと呼ばれる統計的手法を用いてパフォーマンスデータ (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オーバーフローデータ、アドレス空間データ、オペレーティングシステムに関する要約情報など) を収集します。標本コレクタについての詳細は、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、標本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、逆アセンブリ命令のレベルでパフォーマンスに関するさまざまなメトリックを表示します。これらのメトリックは、クロックベース、同期遅延、ハードウェアカウンタの 3 つのグループに分類されます。アナライザはまた、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の改善を可能にする「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についての詳細は、第 4 章を参照してください。

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。
- どのソース行および逆アセンブリ命令が特に多くの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のメインウィンドウには、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、補助ウィンドウにその関数の呼び出し元と呼び出し先が表示されます。このウィンドウでは、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 つのウィンドウがあり、ソースコードのウィンドウには、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ウィンドウには、各命令のメトリック付きの逆アセンブリコードとソースコード、およびコンパイラのコメントが表示されます。

パフォーマンスの調整は、ソフトウェア開発者にとって主要な作業ではないこともありますが、標本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、詳細で正確な解析を可能にします。`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。

標本コレクタおよびパフォーマンスアナライザと同等の機能は、コマンド行からも利用できます。

- データの収集は `collect` コマンドを使用して行えます (第 3 章を参照)。
- 標本コレクタは、`dbx` から `collector` サブコマンドを使用して実行できます (第 3 章を参照)。
- コマンド行ユーティリティの `er_print` は、アナライザの ASCII 形式の各種表示データを印刷します。詳細は、第 5 章を参照してください。
- コマンド行ユーティリティの `er_src` は、コンパイラのコメント付きのソースおよび逆アセンブリコードを表示します (ただし、パフォーマンスデータは含まれません)。詳細は、第 7 章を参照してください。

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ための UNIX® ツールです。Solaris 2.6、7、8 (SPARC™ および Intel プラットフォーム版) に付属しています。

- `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。

第2章

標本コレクタとパフォーマンスアナライザの使用方法

この章では、標本コレクタとパフォーマンスアナライザの使用方法を説明します。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、`synprog` という C プログラムを例に、関数、行、命令レベルで見たプログラムの実行時間の問題と、パフォーマンスアナライザによるオブジェクトモジュールの再帰呼び出しと動的読み込みの処理方法を明らかにします。
- 例 2 : OpenMP による並列化戦略。この例では、`omptest` という Fortran プログラムを例に、OpenMP 指令を使用し、並列化手法を変更したときの効率性の違いを明らかにします。
- 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、`mttest` という C プログラムを例に、明示的なマルチスレッド処理を利用して、スレッド間の処理スケジューリング手法を変更したときの効率性の違いを明らかにします。
- 例 4 : キャッシュの動作と最適化。この例では、`cachetest` という Fortran 90 プログラムを例に、ハードウェアカウンタを使用したメモリーアクセスが実行速度に及ぼす影響を明らかにします。

これらの例では、パフォーマンスに関する次のような一般的な疑問を解決します。

- アプリケーションによってどのようなリソースが使用されるのか。
- アプリケーションのどの部分でリソースの多くが使用されるのか。
- 特定に地点に達するまでにアプリケーションがどのような実行過程を経ているのか。

コンパイラによって可能な範囲でコードがすでに最適化されている場合、アプリケーションがそれ以上に効率的に動作するようにするには、パフォーマンス解析によってアルゴリズムそのものを改良する方法を探ることになります。アプリケーションが最も多くのリソースを使用する部分を特定すると、パフォーマンスアナライザが提供するさまざま手段を使用してコードを検討し、なぜその部分で最も多くのリソースが使用されるのかを調べることができます。

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。

サンプルプログラムの実行準備

この節では、Sun WorkShop™ 6 update 2 製品が、/opt ディレクトリにインストールされていると仮定して説明しています。Sun WorkShop 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確認してください。

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/WorkShop/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. Sun WorkShop 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/WorkShop/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` - 特に条件はありません。
- `omptest` - 次の条件が満たされている必要があります。
  - 少なくとも 4 つの CPU を搭載した SPARC マシンで実行すること。
  - Forte™ for High Performance Computing (HPC) ライセンスを取得していること (Fortran 95 コンパイラの並列化機能を利用するために必要)。
- `mttest` - 少なくとも 4 つの CPU を搭載した SPARC マシンで実行すること。
- `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。そのうちの一部、特に命令セットアーキテクチャーを選択する `-xarch` 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、このオプションを 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、整数乗算や除算命令を使用するのではなく、libc.so から .mul および .div ルーチンを呼び出すコードを生成し、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、159 ページの「<未知> 関数」を参照してください。

これら 4 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。

標本コレクタとパフォーマンスアナライザの実行

サンプルプログラムのデータ収集には、コマンド行インタフェース (CLI) またはグラフィカルユーザーインタフェース (GUI) のどちらでも使用できます。これらインタフェースの具体的な使用方法は、各サンプルプログラムの節で説明します。この節では、Sun WorkShop GUI から標本コレクタとパフォーマンスアナライザを実行する一般的な手順を示します。データの収集は、コマンド行から collect コマンドまたは dbx の collector コマンドを使用して行うこともできます (これらのコマンドについては第 3 章で説明します)。

また、この節では、パフォーマンスアナライザのいくつかの基本機能についても説明します。

Sun WorkShop のメインウィンドウを開くには、コマンド行で次のコマンドを入力します。

```
% workshop &
```

「標本コレクタ」ウィンドウは、「デバッグ」ウィンドウから開きます。Sun WorkShop のメインウィンドウから「デバッグ」ウィンドウを開くには、「デバッグ」ボタンをクリックします。

それまでに Sun WorkShop GUI を実行したことがない場合は、ここで「新規プログラムのデバッグ」ダイアログが表示されます。テキストボックスにサンプルプログラムの名前を入力するか、リストボックスからサンプルプログラムを選択してください。「デバッグ」ウィンドウのメニューバーから「デバッグ」➤「新規プログラム」を選択し、「新規プログラムのデバッグ」ダイアログを表示してプログラムを読み込むこともできます。

「標本コレクタ」ウィンドウを開くには、「デバッグ」ウィンドウのメニューバーから「ウィンドウ」➤「標本コレクタ」を選択します。「標本コレクタ」ウィンドウは、66 ページの 図 3-1 に示しています。このウィンドウでは、実験名を入力し、収集するデータの種類と収集間隔を選択します。

プログラムを実行してデータを収集するには、「標本コレクタ」ウィンドウまたは「デバッグ」ウィンドウにある「開始」ボタンをクリックします。

パフォーマンスアナライザを起動するには、「標本コレクタ」ウィンドウまたは Sun WorkShop のメインウィンドウにある「解析」ボタンをクリックします。

「アナライザ」ウィンドウは、92 ページの 図 4-2 に示しています。

「標本コレクタ」ウィンドウからパフォーマンスアナライザを起動すると、前回収集された実験データが自動的に読み込まれます。

標本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、パフォーマンスアナライザのメインウィンドウに、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、84 ページの「パフォーマンスメトリックの種類と目的」を参照してください。

関数リストから関数を選択して「呼び出し元 - 呼び出し先」ボタンをクリックすると、関数の呼び出し元と呼び出し先に関する情報の入った「呼び出し元 - 呼び出し先」ウィンドウが表示されます。このウィンドウには、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」ウィンドウには、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できます。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。

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。

- 単純なメトリック解析 (12 ページ): 関数リスト、注釈付きソースコードリスト、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。
- gprof の誤った推論 (15 ページ): 呼び出し元 - 呼び出し先リストを使用し、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。

- 再帰の効果 (17 ページ): 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。
- 動的にリンクされた共有オブジェクトの読み込み (21 ページ): 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、「サンプルプログラムの実行準備」および「標本コレクタとパフォーマンスアナライザの実行」の 2 つの節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
% collect synprog
% analyzer test.1.er &
```

GUI を使用して 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、以下の操作を行います。

1. 「デバッグ」ウィンドウを開いて、synprog を読み込みます。
2. 「デバッグ」ウィンドウから「デバッグ」➤「デバッグディレクトリを変更」を選択し、ディレクトリを *~/work-directory*/synprog に変更します。

   これで、共有オブジェクトの so_syn.so および so_syx.so が、必要なときに検出されるようになります。
3. 「ウィンドウ」➤「標本コレクタ」を選択し、実験名が test.1.er であることを確認します。

   この例では、クロックベースのメトリックとデフォルトの収集間隔を使用します。「標本コレクタ」ウィンドウのデータオプションの設定を変更する必要はありません。
4. 「開始」ボタンをクリックします。

   プログラムが実行され、標本コレクタがデータを収集します。

5. プログラムが終了したら、「解析」ボタンをクリックします。

   test.1.er の実験データが「アナライザ」ウィンドウに読み込まれます。

   これで、以降の節の手順に従って synprog 実験データの解析に進むことができます。

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、icputime() では整数型の変数です。

1. 関数リストから cputime() および icputime() を見つけます。

   これら 2 つの関数の排他的ユーザー CPU 時間を比較します。cputime() は、icputime() より実行時間がかなり長くなっています。

   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。

2. cputime() をクリックして選択し、「ソース...」をクリックします。

   テキストエディタが開き、cputime() 関数の注釈付きソースコードが表示されます。テキストエディタのウィンドウは、必要に応じてサイズを変更してください。

```
                          400. int
                          401. cputime(int k)
                          402. {
                          403.         int     i;      /* temp value for loop */
                          404.         int     j;      /* temp value for loop */
                          405.         volatile float  x;      /* temp variable for f.p. calculation */
                          406.         hrtime_t        start;
                          407.         hrtime_t        vstart;
                          408.
    0.          0.        409.         start = gethrtime();
    0.          0.        410.         vstart = gethrvtime();
                          411.
                          412.         /* Log the event */
    0.          0.        413.         wlog("start of cputime", NULL);
                          414.
    0.          0.        415.         if(k == 0) {
    0.          0.        416.                 k = 80;
                          417.         }
    0.          0.        418.         for (i = 0; i < k; i ++) {
    0.          0.        419.                 x = 0.0;
    0.840       0.840     420.                 for(j=0; j<1000000; j++) {
##  2.870       2.870     421.                         x = x + 1.0;
                          422.                 }
                          423.         }
                          424.
                          425.         whrvlog( (gethrtime() - start),  (gethrvtime() - vstart),
    0.          0.        426.                 "cputime", NULL);
    0.          0.        427.         return 0;
```

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。

3. icputime() をクリックして選択し、「ソース...」をクリックします。

テキストエディタには、cputime() のソースコードに代わって、icputime() のソースコードが表示されます。ループ行と x をインクリメントする行を見てみます。

```
                                433. int
                                434. icputime(int k)
                                435. {
                                436.         int     i;      /* temp value for loop */
                                437.         int     j;      /* temp value for loop */
                                438.         volatile long   x;      /* temp variable for long calculation */
                                439.         hrtime_t        start;
                                440.         hrtime_t        vstart;
                                441.
  0.            0.              442.         start = gethrtime();
  0.            0.              443.         vstart = gethrvtime();
                                444.
                                445.         /* Log the event */
  0.            0.              446.         wlog("start of icputime", NULL);
                                447.
  0.            0.              448.         if(k == 0) {
  0.            0.              449.                 k = 80;
                                450.         }
  0.            0.              451.         for (i = 0; i < k; i ++) {
  0.            0.              452.                 x = 0;
  1.150         1.150           453.                 for(j=0; j<1000000; j++) {
  1.420         1.420           454.                         x = x + 1;
                                455.                 }
                                456.         }
                                457.
                                458.         whrvlog( (gethrtime() - start),  (gethrvtime() - vstart),
  0.            0.010           459.                 "icputime", NULL);
  0.            0.              460.         return 0;
```

ループ行で費やされている時間は、cputime() のループ行で費やされている時間とほぼ同じですが、x がインクリメントされる行の実行時間は、cputime() 内の対応する行に比べて短くなっています。

次に、これらの関数の注釈付き逆アセンブリコードを見て、この理由を調べます。

4. 関数リスト内の cputime() をクリックし、「逆アセンブル...」をクリックします。

テキストエディタに、cputime() の注釈付き逆アセンブリコードが表示されます。下方向にスクロールして、x がインクリメントされるソースコード行の命令の部分を表示します。

```
                                421.                                x = x + 1.0;
    0.210         0.210                [ 421]    13944:  ld        [%fp - 16], %f2
## 1.610         1.610                [ 421]    13948:  fstod     %f2, %f4
    0.            0.                   [ 421]    1394c:  ldd       [%l3], %f2
    0.540         0.540                [ 421]    13950:  faddd     %f4, %f2, %f2
    0.510         0.510                [ 421]    13954:  fdtos     %f2, %f2
    0.            0.                   [ 421]    13958:  st        %f2, [%fp - 16]
    0.230         0.230                [ 420]    1395c:  ld        [%fp - 12], %l0
    0.420         0.420                [ 420]    13960:  add       %l0, 1, %l1
    0.190         0.190                [ 420]    13964:  cmp       %l1, %l2
    0.            0.                   [ 420]    13968:  bl        0x13944
    0.            0.                   [ 420]    1396c:  st        %l1, [%fp - 12]
    0.            0.                   [ 418]    13970:  ld        [%fp - 8], %l0
    0.            0.                   [ 418]    13974:  add       %l0, 1, %l1
    0.            0.                   [ 418]    13978:  ld        [%fp + 68], %l0
    0.            0.                   [ 418]    1397c:  cmp       %l1, %l0
    0.            0.                   [ 418]    13980:  bl        0x13914
    0.            0.                   [ 418]    13984:  st        %l1, [%fp - 8]
```

fstod および fdtos 命令の実行にかなりの時間が費やされています。これらの命令は、x の値を単精度浮動小数点値から倍精度浮動小数点値、その逆に倍精度浮動小数点値を単精度浮動小数点値に変換します。この変換が必要なのは、1.0 (倍精度浮動小数点定数) ずつ x をインクリメントできるようにするためです。

5. 関数リスト内の icputime() をクリックし、「逆アセンブル...」をクリックします。

テキストエディタに、cputime() の注釈付き逆アセンブリコードが表示されます。下方向にスクロールして、x がインクリメントされるソースコード行の命令の部分を表示してください。

```
                                454.                                x = x + 1;
    0.210         0.210                [ 454]    13a84:  ld        [%fp - 16], %l0
    1.210         1.210                [ 454]    13a88:  add       %l0, 1, %l0
    0.            0.                   [ 454]    13a8c:  st        %l0, [%fp - 16]
    0.240         0.240                [ 453]    13a90:  ld        [%fp - 12], %l0
    0.670         0.670                [ 453]    13a94:  add       %l0, 1, %l1
    0.240         0.240                [ 453]    13a98:  cmp       %l1, %l2
    0.            0.                   [ 453]    13a9c:  bl        0x13a84
    0.            0.                   [ 453]    13aa0:  st        %l1, [%fp - 12]
    0.            0.                   [ 451]    13aa4:  ld        [%fp - 8], %l0
    0.            0.                   [ 451]    13aa8:  add       %l0, 1, %l1
    0.            0.                   [ 451]    13aac:  ld        [%fp + 68], %l0
    0.            0.                   [ 451]    13ab0:  cmp       %l1, %l0
    0.            0.                   [ 451]    13ab4:  bl        0x13a64
    0.            0.                   [ 451]    13ab8:  st        %l1, [%fp - 8]
```

関係している命令は、読み込み、加算、格納だけであり、変換が不要なため、これらに要する時間は、cputime() 内の対応する命令セットに要する時間の約 1/3 で済んでいます。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。

さらに進んで...

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。

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. gpf_work() をクリックし、次に「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」ウィンドウが表示されます。このウィンドウについては、103 ページの「関数の呼び出し元と呼び出し先メトリックの表示」で説明します。また、この章の 8 ページの「標本コレクタとパフォーマンスアナライザの実行」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、呼び出し先区画の方は空です。こうした関数を「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が、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 10 倍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. 呼び出し元区画内の gpf_a() をクリックして選択します。

   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. gpf_a() が選択状態になった関数リストにある「ソース」ボタンをクリックします。

   テキストエディタのウィンドウに、gpf_a() の注釈付きソースコードが表示されます。

4. 下方向にスクロールし、gpf_a() および gpf_b() 両方のコードを表示します。

```
                          725. void
                          726. gpf_a()
                          727. {
                          728.         hrtime_t        start;
                          729.         hrtime_t        vstart;
                          730.         int     i;
                          731.
  0.          0.          732.         start = gethrtime();
  0.          0.          733.         vstart = gethrvtime();
                          734.
  0.          0.          735.         for(i = 0; i < 9; i ++) {
  0.          0.340       736.                 gpf_work(1);
                          737.         }
                          738.
                          739.         whrvlog( (gethrtime() - start),  (gethrvtime() - vstart),
  0.          0.          740.                 "gpf_a -- 9 X gpf_work(1)", NULL);
  0.          0.          741. }
                          742.
                          743. void
                          744. gpf_b()
                          745. {
                          746.         hrtime_t        start;
                          747.         hrtime_t        vstart;
                          748.
  0.          0.          749.         start = gethrtime();
  0.          0.          750.         vstart = gethrvtime();
                          751.
  0.          3.700       752.         gpf_work(10);
                          753.
                          754.         whrvlog( (gethrtime() - start),  (gethrvtime() - vstart),
  0.          0.          755.                 "gpf_b -- 1 X gpf_work(10)", NULL);
  0.          0.          756. }
```

   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() の方は、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まれる理由を調べてみます。

5. テキストエディタ内で画面を下方向にスクロールして、gpf_work() のコードを表示します。

```
                                758. void
                                759. gpf_work(int amt)
                                760. {
                                761.         int     i;
                                762.         int     imax;
                                763.
      0.            0.          764.         imax = 4* amt * amt;
                                765.
      0.            0.          766.         for(i = 0; i < imax; i ++) {
                                767.                 volatile float x;
                                768.                 int j;
      0.            0.          769.                 x = 0.0;
      0.790         0.790       770.                 for(j=0; j<200000; j++) {
➡ ##  3.250         3.250       771.                         x = x + 1.0;
                                772.                 }
                                773.         }
      0.            0.          774. }
```

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。

しかしながら、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、誤って gpf_a() からの呼び出しは、gpf_b() からの呼び出しの 10 倍の時間がかかることになります。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。標本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。

- 関数 recurse() は直接的な再帰の例です。この関数は real_recurse() を呼び出し、この 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。

いずれの場合も、排他的メトリックは実際に処理が行われる関数にだけ属し (上記の例では real_recurse() と bounce_a())、最終的な関数を呼び出すすべての関数の包括的メトリックに加算されます。

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

1. 関数リストから recurse() 関数を見つけてクリックします。

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

   a. メニューバーから「表示」➤「検索」を選択します。

      「検索」ダイアログが表示されます。

   b. テキストボックスに関数名として recurse を入力します。

   c. 関数リスト内で目的の関数 recurse() が強調表示されるまで「適用」をクリックします。

   d. 「取消し」をクリックして、ダイアログを閉じます。

   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() を呼び出すことしか行わないためです。

注 - 場合によっては、recurse() に、排他的 CPU 時間として小さな値が表示されることがあります。これは、プロファイリングは統計的な性質をもつものであり、synprog に対する実験で、recurse() 関数にプロファイルイベントがいくつか記録されることがあるためです。しかし、この排他的な時間は、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」をクリックします。

   「呼び出し元 - 呼び出し先」ウィンドウが表示され、recurse() が関数 real_recurse() を呼び出していることが分かります。

3. 呼び出し先区画内の real_recurse() をクリックします。

これで、「呼び出し元 - 呼び出し先」ウィンドウに real_recurse() に関する情報が表示されます。

- 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に表示されます。このことは、recurse() が real_recurse() を呼び出した後、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。
- 表示を簡素化するため、自身の呼び出し先としての real_recurse() は呼び出し先区画に表示されないようになっています。
- 中央の関数区画には、実際に時間が消費される real_recurse() の排他的メトリックと包括的メトリックが表示されます。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。
- 呼び出し元区画には、real_recurse() の排他的メトリックも表示されます。関数によって排他的メトリックが生成されると、「呼び出し元 - 呼び出し先」ウィンドウ内のその関数が現れるどの区画にも、その関数の排他的メトリックが表示されます。

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。

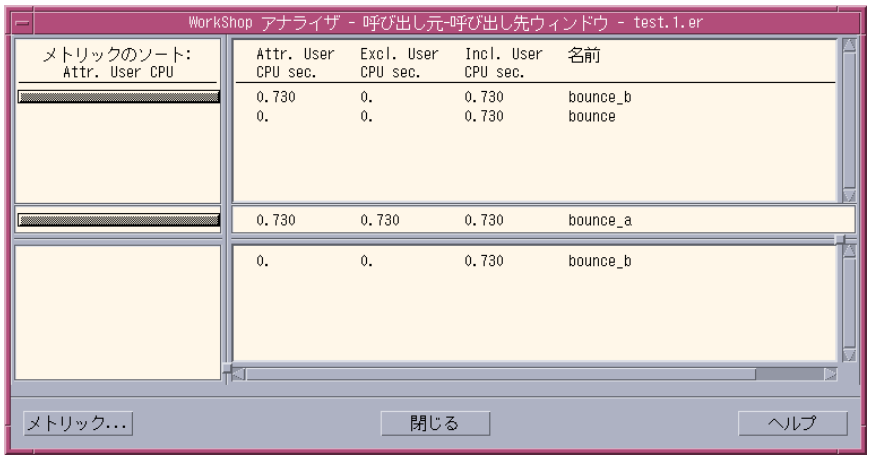
1. **関数リストから bounce() を見つけてクリックします。**

   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」をクリックします。

   「呼び出し元 - 呼び出し先」ウィンドウが表示され、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。

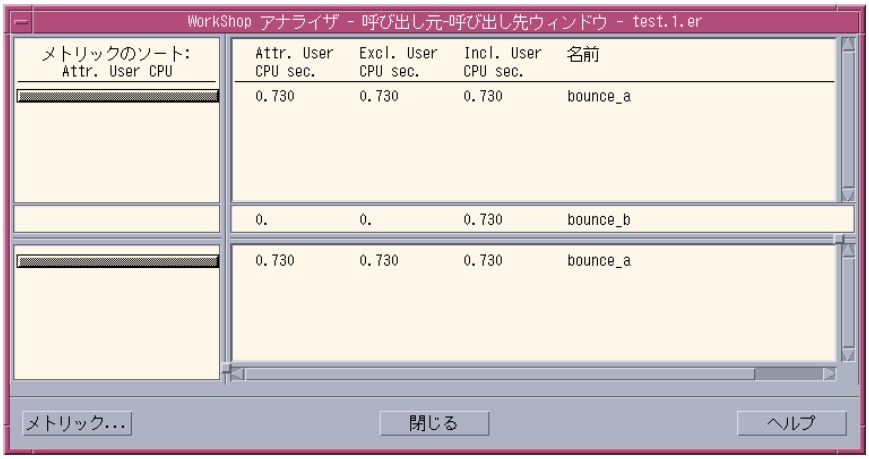


これで、「呼び出し元 - 呼び出し先」ウィンドウに bounce_a() に関する情報が表示されます。

- bounce_a() の呼び出し元として、bounce() および bounce_b() の両方が呼び出し元区画に表示されます。
- また、bounce_b() は、呼び出し先区画にも表示されます。ある関数が自身を再帰的に呼び出すのではなく、中間的な関数を呼び出す場合は、その中間的な関数が呼び出し先区画に表示されます。
- 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。

   これで、「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() の呼び出し元と呼び出し先の両方として bounce_b() が表示されます。同時に、呼び出し元区画と呼び出し先区画の両方に、bounce_a() の排他的メトリックと包括的メトリックが表示されます。これは、関数によって排他的メトリックが生成された場合、アナライザは、その関数が表示される「呼び出し元 - 呼び出し先」ウィンドウ内のどの区画にも、その関数のメトリックを表示するためです。

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、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、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。

このやや手の込んだ練習問題は、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。

- **関数リストをスクロールして、sx_burncpu() および so_burncpu() 両方のメトリックの部分を表示します。**

  so_burncpu() は sx_burncpu() と同じ処理を行いますが、2 回実行されるため、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 倍になっています。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。

- 1 つ目の事例では、ある配列から別の 2 つの配列を更新するコード部分に、それぞれ PARALLEL SECTIONS 指令と PARALLEL DO 指令を使用したときを比較することによって、スレッド間の作業負荷均衡の問題を見てみます。
- 2 つ目の事例では、配列要素を合計してスカラー結果を生成するコード部分に、それぞれ CRITICAL SECTION 指令と REDUCTION 指令を使用したときを比較することによって、メモリーアクセスにおけるスレッド間の競合のコストを見てみます。

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、148 ページの「並列実行とコンパイラ生成の本体関数」および 158 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、「サンプルプログラムの実行準備」および「標本コレクタとパフォーマンスアナライザの実行」の 2 つの節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。

C シェルのコマンド行から omptest 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/omptest
% setenv PARALLEL 4
% collect -o omptest.1.er omptest
% setenv PARALLEL 2
% collect -o omptest.2.er omptest
% unsetenv PARALLEL
% analyzer omptest.1.er &
% analyzer omptest.2.er &
```

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.1.er omptest
$ PARALLEL=2; export PARALLEL
$ collect -o omptest.2.er omptest
$ unset PARALLEL
$ analyzer omptest.1.er &
$ analyzer omptest.2.er &
```

これらの収集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/omptest
$ make collect
$ analyzer omptest.1.er &
$ analyzer omptest.2.er &
```

GUI を使用して omptest 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、以下のようにします。

1. 「デバッグ」ウィンドウを開くか、「デバッグ」ウィンドウのメニューバーから「デバッグ」➤「新規プログラム」を選択し、omptest を読み込みます。
2. 「新規プログラムデバッグ」ダイアログで「環境変数」をクリックするか、「デバッグ」➤「実行時の引数の編集」を選択して「実行時の引数の編集」ダイアログを表示し、「環境変数」をクリックします。

   「環境変数」ダイアログが表示されます。この実験用に、PARALLEL 環境変数に 4 つの CPU を設定します。

   a. 「名前」テキストボックスに PARALLEL、「値」テキストボックスに 4 を入力します。

   b. 「追加」をクリックし、次に「了解」をクリックします。
3. 「デバッグ」ウィンドウのメニューバーから「ウィンドウ」➤「標本コレクタ」を選択し、実験名として omptest.1.er と入力します。

   この例では、クロックベースのメトリックとデフォルトの収集間隔を使用します。「標本コレクタ」ウィンドウのデータオプションを変更する必要はありません。
4. 「開始」ボタンをクリックします。

   プログラムが実行され、標本コレクタがデータを収集します。
5. プログラムが終了したら、「解析」ボタンをクリックします。

   omptest.1.er の実験データが「アナライザ」ウィンドウに読み込まれます。
6. 「デバッグ」ウィンドウから「デバッグ」➤「実行時の引数の編集」を選択し、「実行時の引数の編集」ダイアログで「環境変数」をクリックします。

   2 つ目の実験用に 2 つの CPU を設定します。

a. リストボックスから PARALLEL 環境変数をクリックします。

b. 「値」テキストボックスの値を 2 に変更します。

c. 「変更」をクリックし、次に「了解」をクリックします。

7. 「ウィンドウ」➤「標本コレクタ」を選択します。

   実験名が自動的に omptest.2.er に更新されます。どの設定も変更する必要はありません。

8. 「開始」ボタンをクリックします。

   プログラムが実行され、標本コレクタがデータを収集します。

9. プログラムが終了したら、「解析」ボタンをクリックします。

   omptest.2.er の実験データが「アナライザ」ウィンドウに読み込まれます。

10. 2 つの実験データの生成を完了したら、「環境変数」ダイアログボックスを使用して PARALLEL 環境変数を削除します。

    これで、以降の節の手順に従って omptest 実験データの解析に進むことができます。

PARALLEL SECTIONS と PARALLEL DO 戦略の比較

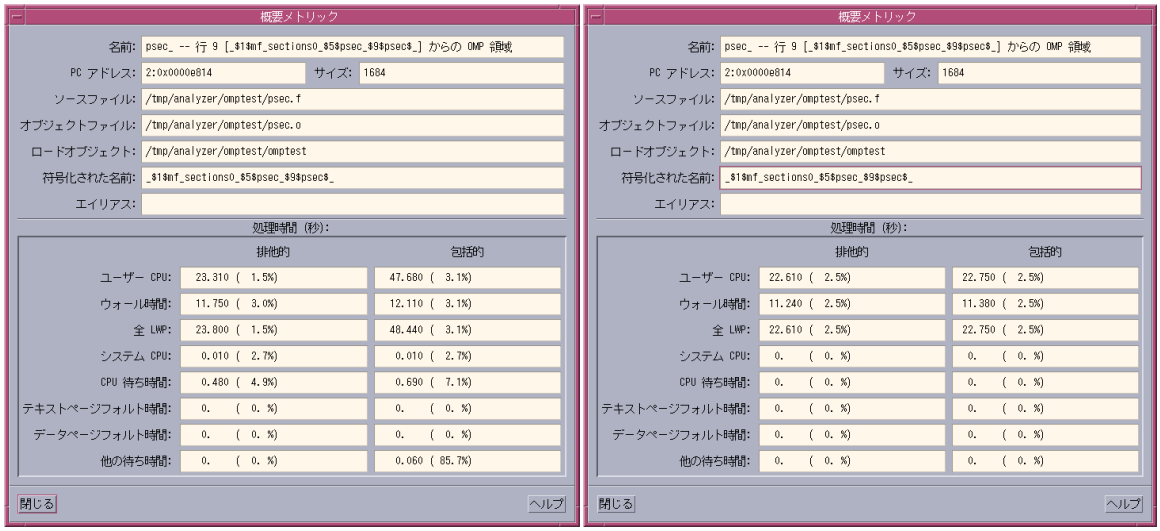
この節では、それぞれ PARALLEL SECTIONS 指令と PARALLEL DO 指令を使用する 、psec_() および pdo_() という 2 つのルーチンのパフォーマンスを比較します。これらルーチンのパフォーマンスは、CPU の個数の関数として比較されます。

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.1.er、もう一方のウィンドウに omptest.2.er を読み込みます。

1. 両方のアナライザウィンドウについて、関数リストから psec_ を含む行をクリックし、「表示」➤「概要メトリックを表示」を選択します。

   この関数はリストの最後にあります。また、その他にも、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. メトリックを比較できるように、「概要メトリック」ウィンドウを左右に並べます。



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 1 対 2 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。

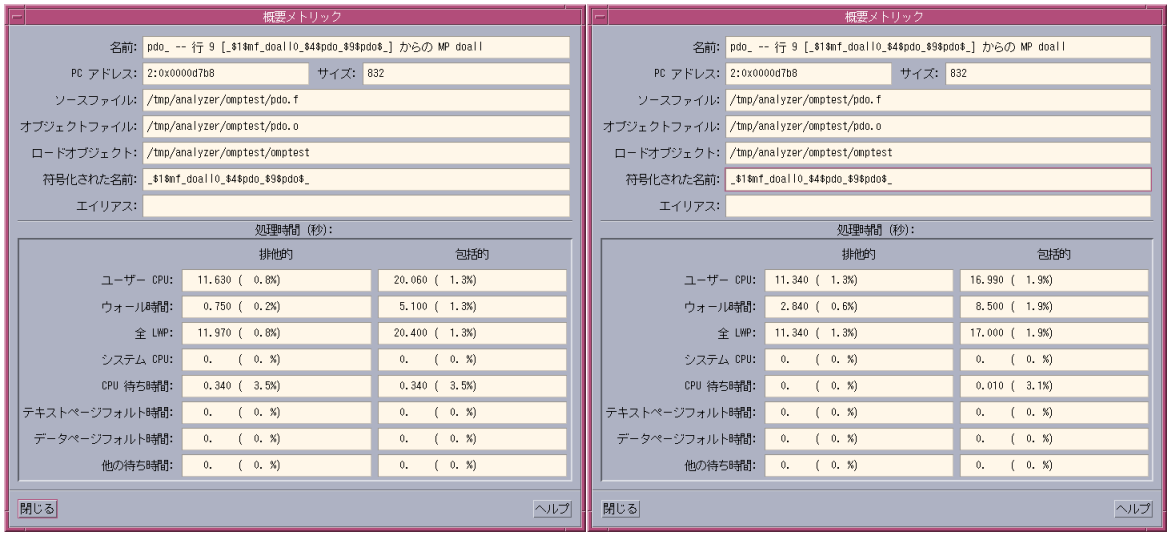
   これに対し 4 つの CPU での実行の場合、`psec_()` のウォール時間は 2 つの CPU での実行のときとほぼ同じですが、ユーザー CPU 時間と全 LWP 時間がともに長くなっています。`psec_()` の `PARALLEL SECTION` 構文内のセクション数は 2 つだけであるため、その実行に必要なスレッドは 2 つだけです。つまり、いつでも、使用可能な 4 つの CPU のうちの 2 つだけが使用され、残る 2 つの CPU の CPU 時間は仕事待ちに費やされることになります。つまり、行える仕事がないにもかかわらず、時間が浪費されることになります。

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」ウィンドウに `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 1 対 2 ですが、4 つの CPU では約 1 対 4 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。



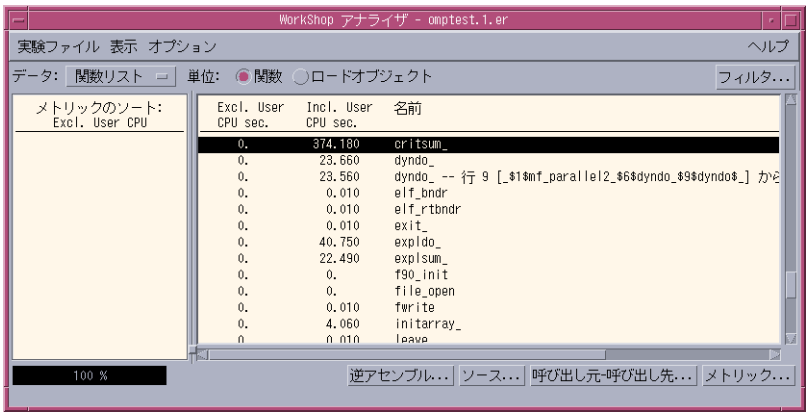
この図では、左側のウィンドウが 4 つの CPU での実行データです。

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。この場合の並列化戦略では、do ループに組み込まれた同じ代入文を処理し、3 つの 2 次元配列の内容を合計します。

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

1. 4 つの CPU の実験について、`omptest.1.er` の関数リストから `critsum_()` および `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 時間は大変な長さになります。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 しかできないためです。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_()` よりずっと短くなります。これは、`redsum_()` では `REDUCTION` 戦略が採用されていて、`(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、「サンプルプログラムの実行準備」および「標本コレクタとパフォーマンスアナライザの実行」の 2 つの節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。クロックベースのデータだけでなく、同期待ち監視データも記録します。

コマンド行から 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、以下のようにコマンドを入力します。

```
% cd ~/work-directory/mttest
% collect -s on -o mttest.1.er mttest
% analyzer mttest.1.er &
% collect -s on -o mttest.2.er mttest -u
% analyzer mttest.2.er &
```

これらの収集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.1.er &
% analyzer mttest.2.er &
```

GUI を使用して 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、以下の操作を行います。

1. 「デバッグ」ウィンドウを開くか、「デバッグ」ウィンドウのメニューバーから「デバッグ」➤「新規プログラム」を選択し、mttest を読み込みます。
2. 「デバッグ」ウィンドウのメニューバーから「ウィンドウ」➤「標本コレクタ」を選択し、実験名として mttest.1.er と入力します。
3. 「同期待ちの監視」チェックボックスが選択されていることを確認します。
4. 「開始」ボタンをクリックし、データの収集を開始します。

   プログラムが実行され、標本コレクタがデータを収集します。
5. プログラムが終了したら、「解析」ボタンをクリックします。

   mttest.1.er の実験データが、アナライザウィンドウに読み込まれます。
6. 「デバッグ」ウィンドウのメニューバーから「デバッグ」➤「実行時の引数の編集」を選択します。

   「引数」ボックスに値 -u を入力して「了解」をクリックします。このオプションは、強制的に mttest が単一プロセッサ上で動作するようにします。
7. 「デバッグ」ウィンドウのメニューバーから「ウィンドウ」➤「標本コレクタ」を選択します。

   実験名が mttest.2.er で、「同期待ちの監視」チェックボックスが選択されていることを確認します。
8. 「開始」ボタンをクリックします。

   プログラムが実行され、標本コレクタがデータを収集します。
9. プログラムが終了したら、「解析」ボタンをクリックします。

   mttest.2.er の実験データがもう 1 つのアナライザウィンドウに読み込まれます。

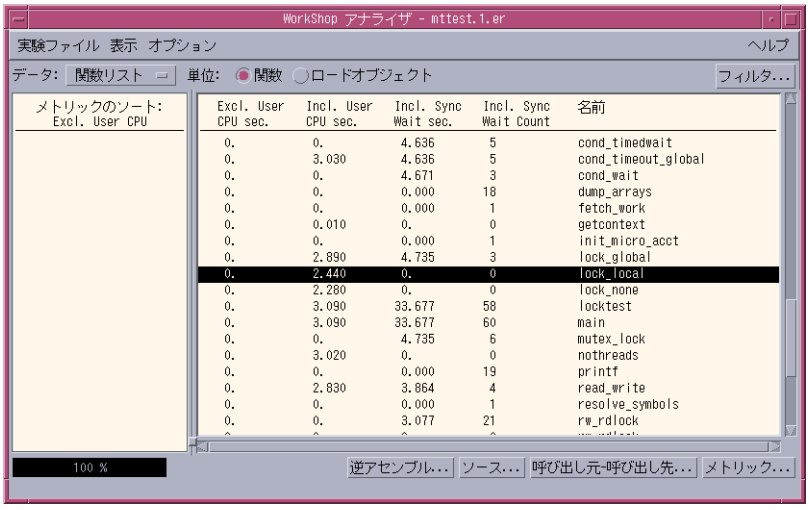
2 つの実験データが読み込まれたら、比較できるように 2 つのアナライザウィンドウを左右に並べます。

これで、以降の節の手順に従って mttest 実験データの解析に進むことができます。

ロック戦略が待ち時間に及ぼす影響

1. 4 つの CPU の実験の関数リストを下方向にスクロールし、lock_local() および lock_global() のデータの部分を表示します。

   両方の関数の包括的ユーザー CPU 時間はほぼ同じです。これは、両方の関数の仕事量が同じであることを示しています。しかし、lock_global() では、同期待ちに多くの時間が費やされていますが、lock_local() には同期待ちの時間はありません。



   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() の方のデータ行をクリックし、「ソース」ボタンをクリックします。

   テキストエディタのウィンドウに、lock_global() の注釈付きソースコードが表示されます。

```
                                                  830. void
                                                  831. lock_global(Workblk *array, struct scripttab *k)
                                                  832. {
                                                  833.         /* acquire the global lock */
                                                  834.
                                                  835. #ifdef SOLARIS
  0.          0.          6.993       3           836.         mutex_lock(&global_lock);
                                                  837. #endif
                                                  838. #ifdef POSIX
                                                  839.         pthread_mutex_lock(&global_lock);
                                                  840. #endif
                                                  841. #ifdef LWP
                                                  842.         _lwp_mutex_lock(&global_lock);
                                                  843. #endif
```

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. lock_global() の方のデータ行をクリックし、「ソース」ボタンをクリックします。

   テキストエディタのウィンドウに lock_global() の注釈付きソースコードが表示されます。

```
                                                  923. void
                                                  924. lock_local(Workblk *array, struct scripttab *k)
                                                  925. {
                                                  926.         /* acquire the local lock */
                                                  927. #ifdef SOLARIS
  0.          0.          0.          0           928.         mutex_lock(&(array->lock));
                                                  929. #endif
                                                  930. #ifdef POSIX
                                                  931.         pthread_mutex_lock(&(array->lock));
                                                  932. #endif
                                                  933. #ifdef LWP
                                                  934.         _lwp_mutex_lock(&(array->lock));
                                                  935. #endif
```

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ています。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、つまり、lock_local() の同期待ち時間はゼロになります。

次の手順に進む前に、注釈付きソースコードを表示しているテキストエディタのウィンドウを閉じてください。

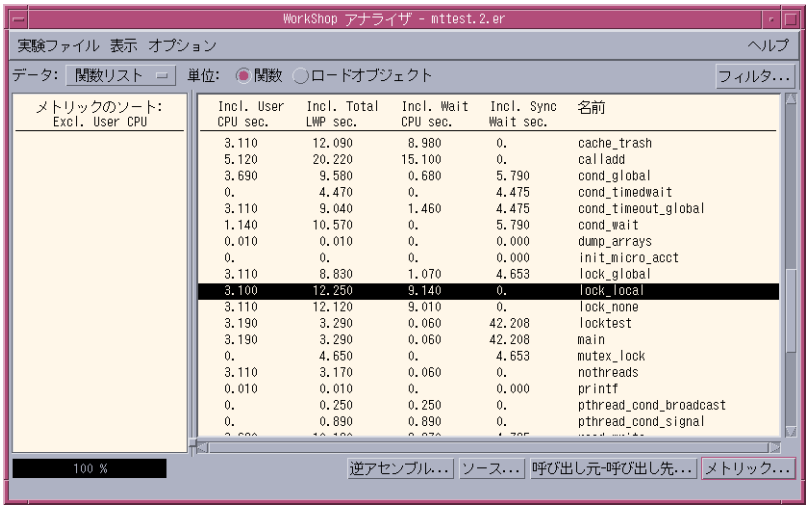
4. 単一 CPU の実験の mttest.2.er の関数リストから、「メトリック」をクリックします。

   「メトリック」ダイアログが表示されます。次の変更を加えます。

   a. 「排他的ユーザー CPU 時間」および「包括的 Sync 待ちカウント」の「時間表示」欄を選択解除します。

b. 同じく「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」欄を選択します。

5. 下方向にスクロールし、`lock_local()` および `lock_global()` のデータの部分を表示します。



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。また、同期の動作も、4 つの CPU システムのときと同じであり、`lock_global()` では、同期待ちに多くの時間が費やされていますが、`lock_local()` では、同期待ちの時間はありません。

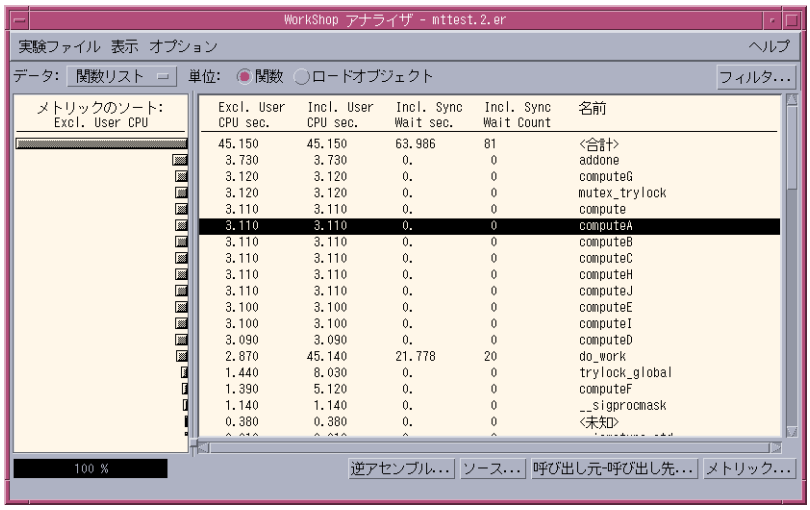
ただし、実際には、`lock_global()` の全 LWP 時間は `lock_local()` より短くなっています。これは、それぞれのロックシステムにおけるスレッド実行のスケジューリング方法が原因です。`lock_global()` の大域ロックでは、各スレッドは処理が完了するまで順次動作できます。これに対して `lock_local()` のローカルロックでは、その実行時間のほんの一部が各スレッドに割り当てられ、すべてのスレッドが完了するまでこのプロセスが繰り返されることになります。こうして、どちらの場合も、スレッドは大量の時間を仕事待ちに費やします。`lock_global()` 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

6. 次に進む前に「メトリック」をクリックし、「デフォルト」、「了解」の順にクリックして再度デフォルトのメトリックを読み込みます。

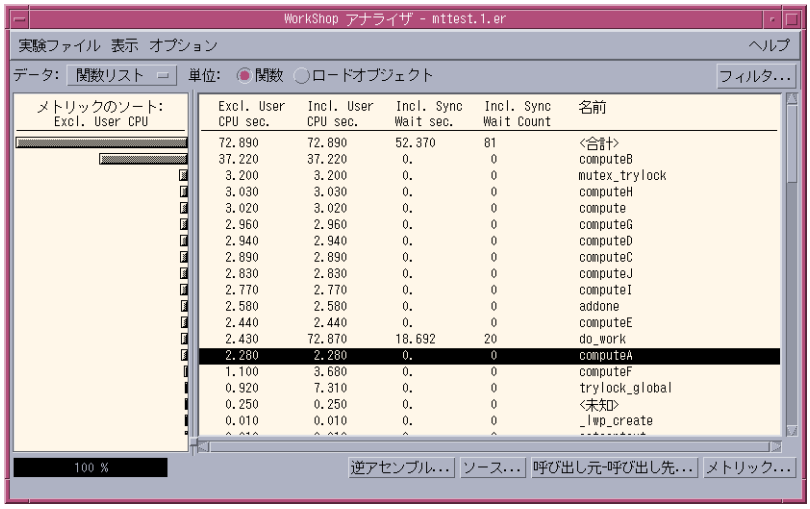
データ管理がキャッシュのパフォーマンスに及ぼす影響

1. 両方のアナライザウィンドウについて、関数リストをスクロールして `ComputeA()` および `ComputeB()` のデータの部分を表示します。

   単一 CPU の実験 `mttest.2.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。

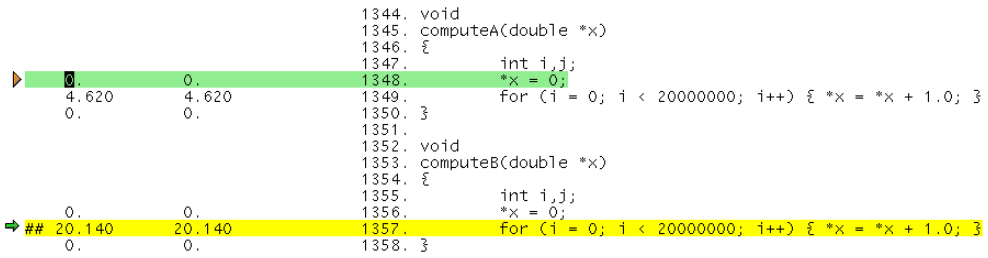


   4 つの CPU の実験の `mttest.1.er` の場合、`ComputeB()` は、`ComputeA()` に比べて長い包括的ユーザー CPU 時間を費やします。



   この後の操作は、4 つの CPU の実験である `mttest.1.er` に対して行います。

2. ComputeA() をクリックして「ソース」ボタンをクリックします。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() および computeB() のソースコードを表示します。



(この図の注釈付きソースコードには、ユーザー CPU 時間だけが表示されています。実際の表示では、選択解除していない限り、他のメトリックも表示されます。)

これらの関数のコードは同じで、ともに変数に 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。

ソースコードを調べ終えたら、テキストエディタのウィンドウを閉じてください。

3. 関数リスト内の ComputeA() をクリックし、「呼び出し元 - 呼び出し先」ボタンをクリックします。

「呼び出し元 - 呼び出し先」ウィンドウが開き、中央の表示区画に選択した関数の ComputeA()、上の区画にその呼び出し元が表示されます。

4. 呼び出し元の lock_none() をクリックします。

これで、「呼び出し元 - 呼び出し先」ウィンドウだけでなく、関数リストにも、選択された関数として lock_none() が表示されます。

5. 関数リスト内の「ソース」ボタンをクリックします。

テキストエディタのウィンドウに、lock_none() の注釈付きソースコードが表示されます。

6. 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、ComputeB() を呼び出す関数である cache_trash() のコードも見えるようにします。

```
                                793. void
                                794. lock_none(Workblk *array, struct scripttab *k)
                                795. {
    0.          0.              796.         array->ready = array->start;
    0.          0.              797.         array->vready = array->vstart;
                                798.
    0.          0.              799.         array->compute_ready = array->ready;
    0.          0.              800.         array->compute_vready = array->vready;
                                801.
                                802.         /* do some work on the current array */
    0.          4.620           803.         (k->called_func)(&array->list[0]);
                                804.
    0.          0.              805.         array->compute_done = gethrtime();
    0.          0.              806.         array->compute_vdone = gethrvtime();
                                807.
    0.          0.              808. }
                                809.
                                810. /* cache_trash: multiple threads refer to adjacent words,
                                811.  *      causing false sharing of cache lines, and trashing
                                812.  */
                                813. void
                                814. cache_trash(Workblk *array, struct scripttab *k)
                                815. {
    0.          0.              816.         array->ready = array->start;
    0.          0.              817.         array->vready = array->vstart;
                                818.
    0.          0.              819.         array->compute_ready = array->ready;
    0.          0.              820.         array->compute_vready = array->vready;
                                821.
                                822.         /* use a datum that will share a cache line with others */
    0.         20.140           823.         (k->called_func)(&element[array->index]);
                                824.
    0.          0.              825.         array->compute_done = gethrtime();
    0.          0.              826.         array->compute_vdone = gethrvtime();
    0.          0.              827. }
                                828.
```

(この図の注釈付きソースコードには、ユーザー CPU 時間だけが表示されています。実際の表示では、選択解除していない限り、他のメトリックも表示されます。)

`ComputeA()` と `ComputeB()` は、ともに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。

7. **`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、直接読み取りと書き込みを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。

さらに進んで...

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。「標本コレクタ」ウィンドウの「解析」ボタンをクリックするのではなく、「アナライザ」ウィンドウから「実験ファイル」➤「追加」を選択することによって、この新しい実験の情報を以前の実験と結合することができます。

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。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 つ目と 3 つ目のコピーは、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、-O3 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、「サンプルプログラムの実行準備」および「標本コレクタとパフォーマンスアナライザの実行」の 2 つの節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 cachetest.1.er cachetest
% collect -o cachetest.2.er -h fpadd,,fpmul cachetest
% collect -o cachetest.3.er -h cycles,,insts1 cachetest
% collect -o cachetest.4.er -h dcstall cachetest
% analyzer cachetest.1.er cachetest.2.er &
```

これらの 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/cachetest
% make collect
% analyzer cachetest.1.er cachetest.2.er
```

GUI を使用して 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、以下の操作を行います。

1. 「デバッグ」ウィンドウを開くか、「デバッグ」ウィンドウのメニューバーから「デバッグ」➤「新規プログラム」を選択し、cachetest を読み込みます。
2. 「デバッグ」ウィンドウのメニューバーから「ウィンドウ」➤「標本コレクタ」を選択し、実験名として cachetest.1.er と入力します。

3. 「開始」ボタンをクリックします。

   プログラムが実行され、標本コレクタが最初の実験用のクロックベースのデータを収集します。

4. プログラムが終了したら、「解析」ボタンをクリックします。

   `cachetest.1.er` の実験データがアナライザウィンドウに読み込まれます。

5. 「標本コレクタ」ウィンドウで最初のハードウェアカウンタのチェックボックスをクリックし、対応するテキストボックスから `fpadd` を選択するか、`fpadd` と入力します。

6. 2 つ目のハードウェアカウンタのチェックボックスをクリックし、対応するテキストボックスで `fpmu1` を選択するか、`fpmu1` と入力します。

7. 「開始」ボタンをクリックします。

   プログラムが実行され、標本コレクタが 2 つ目の実験用のハードウェアカウンタのデータを収集します。

8. ハードウェアカウンタのテキストボックスで `cycles` を選択するか、`cycles` と入力します。さらに、ハードウェアカウンタ `insts1` についても同じ操作を行います。

9. 「開始」ボタンをクリックします。

   プログラムが実行され、標本コレクタが 3 つ目の実験用のハードウェアカウンタのデータを収集します。

10. 2 つ目のハードウェアカウンタを選択解除し、最初のハードウェアカウンタのテキストボックスで `dcstall` を選択するか、`dcstall` と入力します。

11. 「開始」ボタンをクリックします。

    プログラムが実行され、標本コレクタが 4 つ目の実験用のハードウェアカウンタのデータを収集します。

12. アナライザのウィンドウから「実験ファイル」➤「追加」を選択し、`cachetest.2.er` を追加します。

    詳細は、94 ページの「パフォーマンスアナライザへの実験の追加」を参照してください。

    アナライザのウィンドウが開き、排他的メトリックのデータだけが表示されます。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、132 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。

表示の設定

実際のプログラムのルーチンに関する情報だけを表示するようにし、システムコールを除外するには、次の操作を行います。

1. 「表示」➤「次の条件を含んだロードオブジェクトを選択」を選択します。

   すべてのロードオブジェクトが選択状態になっています。

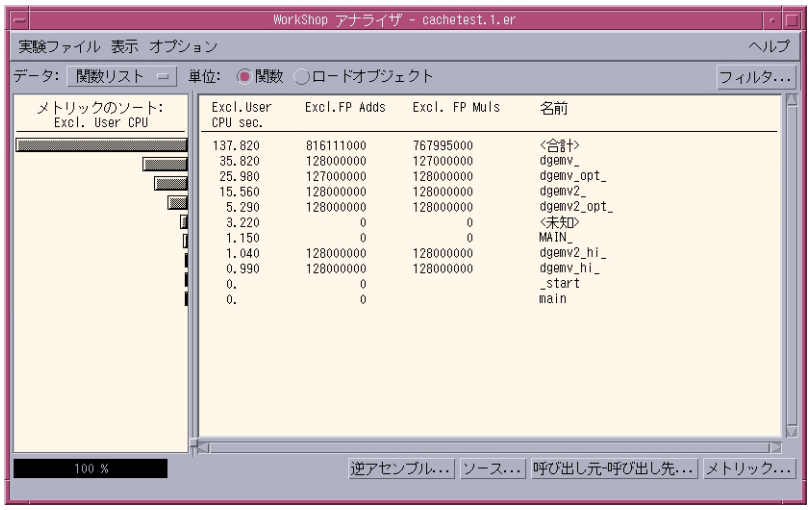
2. 「すべてを選択解除」ボタンをクリックし、ロードオブジェクトリストから `cachetest` を選択して「了解」をクリックします。

   ロードオブジェクトの名前にはパス名が含まれます。メインのロードオブジェクトのルーチンから収集したデータだけが表示されるようになります。

   これで、以降の節の手順に従って `cachetest` 実験データの解析に進むことができます。

実行速度

1. 6 つの関数 (`dgemv`、`dgemv2`、`dgemv_opt`、`dgemv2_opt`、`dgemv_hi`、`dgemv2_hi`) 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。



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)。パフォーマンス的には、`dgemv2` が `dgemv` より、`dgemv2_opt` が `dgemv_opt` より良くなっていますが、`dgemv2_hi` と `dgemv_hi` のパフォーマンスはほぼ同じです。

2. **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 値を比較します。**

   最初の実験については、データから得られた値は出力された値と同じです (統計上の誤差内)。その他の実験では、ハードウェアカウンタデータ収集のオーバーヘッドのために、出力された値は得られた値より小さくなります。クロックベースのデータだけを収集する最初の実験では、オーバーヘッドが最小になります。

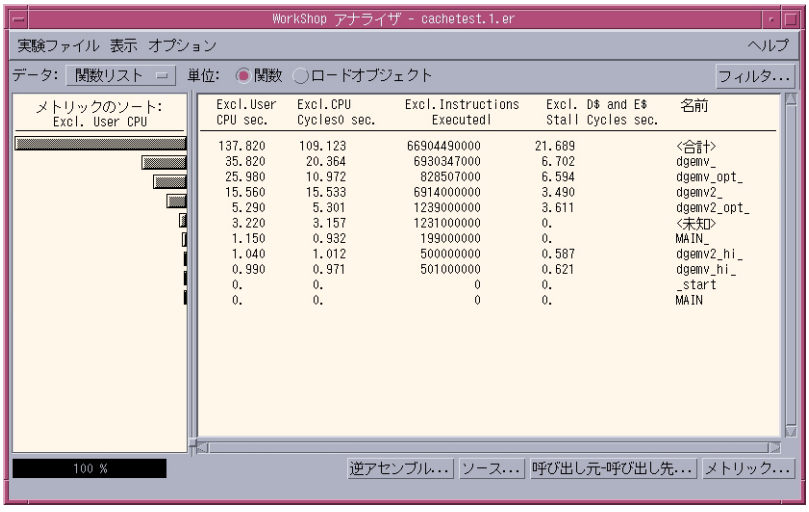
プログラムの構造とキャッシュの動作

この節では、`dgemv2` がパフォーマンス的に `dgemv` より優れている理由を検討します。

1. **「実験ファイル」➤「解除」を選択し、`cachetest.2.er` を解除します。**

2. **「実験ファイル」➤「追加」を選択し、`cachetest.3.er` と `cachetest.4.er` を追加します。**

   実験ごとに「実験ファイルの追加」ダイアログを開く必要があります。関数名が見えるように、必要に応じてウィンドウのサイズを変更してください。



3. ユーザー CPU 時間と CPU サイクルの値を比較します。

   dgemv の両者の差は、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。dgemv2 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。

4. dgemv と dgemv2 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 の方がずっと短くなっています。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。

5. 「メトリック」をクリックし、「命令の実行」と「CPU サイクル」のメトリックを選択解除します。

6. dgemv をクリックし、「ソース」をクリックします。テキストエディタのウィンドウが表示されたら、必要に応じてウィンドウのサイズを変更し、スクロールして、dgemv と dgemv2 両方のソースコードが見えるようにします。

```
    Excl. User  Excl. D$ and E$
    CPU sec.    Stall Cycles sec.
                                      1. !-----------------------------------------------------------
                                      2. ! Standard BLAS interface:      A(1:m) = B(1:m,1:n) * C(1:n)
                                      3. !-----------------------------------------------------------
▷   0.          0.                    4.       SUBROUTINE dgemv (transa, m, n, alpha, b, ldb,    &
                                      5.      &                  c, incc, beta, a, inca)
                                      6.       CHARACTER (KIND=1) :: transa
                                      7.       INTEGER   (KIND=4) :: m, n, incc, inca, ldb
                                      8.       REAL      (KIND=8) :: alpha, beta
                                      9.       REAL      (KIND=8) :: a(1:m), b(1:ldb,1:n), c(1:n)
                                     10.       INTEGER            :: i, j
                                     11.
    0.          0.                   12.       a(1:m) = 0.0
                                     13.
    0.          0.                   14.       DO i = 1, m
    0.          0.004                15.          DO j = 1, n
⇨ ## 33.940     6.696                16.             a(i) = a(i) + b(i,j) * c(j)
    1.880       0.002                17.          END DO
    0.          0.                   18.       END DO
                                     19.
    0.          0.                   20.       RETURN
    0.          0.                   21.       END
                                     22. !-----------------------------------------------------------
    0.          0.                   23.       SUBROUTINE dgemv2 (transa, m, n, alpha, b, ldb,   &
                                     24.      &                  c, incc, beta, a, inca)
                                     25.       CHARACTER (KIND=1) :: transa
                                     26.       INTEGER   (KIND=4) :: m, n, incc, inca, ldb
                                     27.       REAL      (KIND=8) :: alpha, beta
                                     28.       REAL      (KIND=8) :: a(1:m), b(1:ldb,1:n), c(1:n)
                                     29.       INTEGER            :: i, j
                                     30.
    0.          0.                   31.       a(1:m) = 0.0
                                     32.
    0.          0.                   33.       DO j = 1, n       ! <=-----\ swapped loop indices
    0.010       0.001                34.          DO i = 1, m    !    <=--/
   14.080       3.489                35.             a(i) = a(i) + b(i,j) * c(j)
    1.470       0.000                36.          END DO
    0.          0.                   37.       END DO
                                     38.
    0.          0.                   39.       RETURN
▉   0.          0.                   40.       END
```

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 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 4000)。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 dgemv2 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 (-O3 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. 関数リスト内の「メトリック」をクリックし、「命令の実行」の「値を表示」と「CPU サイクル」の「時間を表示」を選択します。
2. dgemv_opt および dgemv2_opt のメトリックと dgemv および dgemv2 のメトリックをそれぞれ比較します。

   dgemv_opt および dgemv2_opt のソースコードは、それぞれ dgemv および dgemv2 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 -O3 を指定してコンパイルされていることです。ユーザー CPU 時間または CPU サイクルのどちらの測定でも、両方のルーチンとも CPU 時間が同程度に短くなっていますが、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。全体として実行命令数は少なくなり、dgemv に比べて dgemv_opt の方が実行命令数は少ないですが、パフォーマンス向上の度合いは似ています。
3. dgemv_opt をクリックし、「ソース」をクリックします。テキストエディタのウィンドウが表示されたら、必要に応じてウィンドウのサイズを変更し、スクロールして、dgemv_opt と dgemv2_opt 両方のソースコードが見えるようにします。

```
    0.                4.       SUBROUTINE dgemv_opt (transa, m, n, alpha, b, ldb,    &
                      5.      &                     c, incc, beta, a, inca)
                      6.       CHARACTER (KIND=1) :: transa
                      7.       INTEGER   (KIND=4) :: m, n, incc, inca, ldb
                      8.       REAL      (KIND=8) :: alpha, beta
                      9.       REAL      (KIND=8) :: a(1:m), b(1:ldb,1:n), c(1:n)
                     10.       INTEGER            :: i, j
                     11.

                   Loop below pipelined with steady-state cycle count = 2 before unrolling
                   Loop below unrolled 3 times
                   Loop below has 0 loads, 2 stores, 0 prefetches, 0 FPadds, 0 FPmuls, and 0 FPdivs per iteration
    0.              12.       a(1:m) = 0.0
                    13.
    0.              14.       DO i = 1, m

                   Loop below unrolled 4 times
                   Loop below has 4 loads, 0 stores, 0 prefetches, 1 FPadds, 1 FPmuls, and 0 FPdivs per iteration
                   Loop below pipelined with steady-state cycle count = 4 before unrolling
    0.              15.          DO j = 1, n
## 25.980           16.             a(i) = a(i) + b(i,j) * c(j)
                    17.          END DO
                    18.       END DO
                    19.
                    20.       RETURN
                    21.       END
                    22. !-----------------------------------------------------------
    0.              23.       SUBROUTINE dgemv2_opt (transa, m, n, alpha, b, ldb,   &
                    24.      &                     c, incc, beta, a, inca)
                    25.       CHARACTER (KIND=1) :: transa
                    26.       INTEGER   (KIND=4) :: m, n, incc, inca, ldb
                    27.       REAL      (KIND=8) :: alpha, beta
                    28.       REAL      (KIND=8) :: a(1:m), b(1:ldb,1:n), c(1:n)
                    29.       INTEGER            :: i, j
                    30.

                   Loop below pipelined with steady-state cycle count = 2 before unrolling
                   Loop below unrolled 3 times
                   Loop below has 0 loads, 2 stores, 0 prefetches, 0 FPadds, 0 FPmuls, and 0 FPdivs per iteration
    0.              31.       a(1:m) = 0.0
                    32.
    0.              33.       DO j = 1, n       ! <=-----\ swapped loop indices

                   Loop below pipelined with steady-state cycle count = 6 before unrolling
                   Loop below unrolled 3 times
                   Loop below has 4 loads, 2 stores, 0 prefetches, 1 FPadds, 1 FPmuls, and 0 FPdivs per iteration
    0.              34.          DO i = 1, m    !    <---/
    5.290           35.             a(i) = a(i) + b(i,j) * c(j)
                    36.          END DO
                    37.       END DO
                    38.
                    39.       RETURN
                    40.       END
```

(この図の注釈付きソースコードには、ユーザー CPU 時間だけ表示されています。実際の表示では、選択解除していない限り、他のメトリックも表示されます。)

コンパイラのコメントを見ると、各ルーチン内の初期化ループと内部ループが展開されていて、パイプライン化されていることが分かります。dgemv_opt の内部ループの刻み幅は dgemv 内のストライドと同じであるため、キャッシュの動作は同じです。

4. 関数リストから dgemv_hi と dgemv2_hi を見つけます。

   dgemv_hi および dgemv2_hi のソースコードは、それぞれ dgemv および dgemv2 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。両方のルーチンとも、dgemv_opt と dgemv2_opt および CPU 時間、キャッシュパフォーマンスはそれぞれ同じですが、CPU 時間は短くなっています。

5. dgemv_hi をクリックし、「ソース」をクリックします。テキストエディタのウィンドウが表示されたら、必要に応じてウィンドウのサイズを変更し、スクロールして、dgemv_hi 全体のソースコードが見えるようにします。

```
  0.                4.        SUBROUTINE dgemv_hi (transa, m, n, alpha, b, ldb,    &
                    5.       &                    c, incc, beta, a, inca)
                    6.        CHARACTER (KIND=1) :: transa
                    7.        INTEGER   (KIND=4) :: m, n, incc, inca, ldb
                    8.        REAL      (KIND=8) :: alpha, beta
  0.                9.        REAL      (KIND=8) :: a(1:m), b(1:ldb,1:n), c(1:n)
                   10.        INTEGER            :: i, j
                   11.

                  Loop below has 0 loads, 1 stores, 1 prefetches, 0 FPadds, 0 FPmuls, and 0 FPdivs per iteration
                  Loop below pipelined with steady-state cycle count = 1 before unrolling
                  Loop below unrolled 8 times
  0.               12.        a(1:m) = 0.0
                   13.

                  Loop below interchanged with loop on line 15
                  Loop below unrolled and jammed
                  Loop below pipelined with steady-state cycle count = 5 before unrolling
                  Loop below unrolled 8 times
                  Loop below has 5 loads, 1 stores, 5 prefetches, 4 FPadds, 4 FPmuls, and 0 FPdivs per iteration
  0.               14.        DO i = 1, m

                  Loop below unrolled and jammed
                  Loop below interchanged with loop on line 14
  0.               15.           DO j = 1, n
## 0.990           16.              a(i) = a(i) + b(i,j) * c(j)
                   17.           END DO
                   18.        END DO
                   19.
                   20.        RETURN
                   21.        END
```

コンパイラは、このルーチンを最適化するにあたって多くの仕事をしています。具体的には、行 14 と 15 のループを入れ替え、1 回のループサイクルで 4 つの浮動小数点加算と 4 つの浮動小数点乗算を含むループを作成し、キャッシュの動作を改善するための先読み命令を挿入しています。

6. 下方向にスクロールして dgemv2_hi のソースコードを見ます。

ループの入れ替えを除けば、コンパイラのコメントは dgemv_hi 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. 関数リスト内の「逆アセンブリ」をクリックします。

dgemv_hi と、dgemv または dgemv_opt の逆アセンブリコードを比較します。dgemv_hi 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、dgemv ルーチンの 3 つのバージョンの中では最低です。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の収集

この章では、標本コレクタを紹介し、標本コレクタが収集するデータの内容と使用方法を説明します。データを収集する方法は、Sun WorkShop の「デバッグ」ウィンドウを利用する方法、dbx 内から `collector` コマンドを使用する方法、コマンド行から `collect` コマンドを使用する方法があります。

この章では、以下について説明します。

- 標本コレクタが収集するデータの内容
- 収集データの格納場所
- 必要なディスク容量の概算
- プログラムからのデータ収集の制御
- データ収集と解析のためのプログラムのコンパイル
- プログラムに関する制限事項
- `collect` コマンドによるデータの収集
- Sun WorkShop 統合プログラミング環境からのデータの収集
- 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

標本コレクタが収集するデータの内容

標本コレクタは、プログラムおよびそのプログラムが動作するカーネルからのイベントに関するデータを収集します。また、大域データを収集し、データを分類するためのマーカーを記録します。

各イベントから収集されたデータをプロファイルパケット、データを収集するプロセスをプロファイリングといいます。また、マーカーが記録されたときに収集されたデータを標本パケット、その標本パケットを記録することを標本化といいます。収集されたイベントデータは、パフォーマンスアナライザによってパフォーマンスメトリックに変換されます。

あらゆるプロファイルパケットには、次の情報が含まれています。

- 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
- 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 6 章を参照してください。

こうした共通の情報の他に、各プロファイルパケットには、データの種類に固有のデータが含まれます。標本コレクタが収集可能なこの種のデータは、次の 3 つの種類に分類されます。

- 時間ベースのデータ
- 同期待ち監視データ
- ハードウェアカウンタのオーバーフローデータ

時間ベースのデータ

時間ベースのプロファイリングでは、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、パフォーマンスアナライザによって各状態に費やされた、プロファイル間隔の分解能を持つ時間値に変換されます。

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。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、もっと高い分解能でプロファイリング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。

同期待ち監視データ

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。こうしたイベントは同期遅延イベントと呼ばれ、標本コレクタはそれらイベントを記録することができます。同期遅延イベントを収集し、記録するプロセスを同期待ちの監視といいます。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。

同期遅延イベントのデータは、スレッドライブラリ (`libthread.so`) 内の関数の呼び出しを監視することによって収集されます。Sun MPI (Message Passing Interface) を使用する並列プログラムでは、同期イベントのもう 1 つのクラスとして、MPIブロック化呼び出しがあり、MPI ライブラリに対する呼び出しは、スレッドライブラリに対する呼び出しと同じ方法で監視されます。

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監視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することによっ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。

ハードウェアカウンタのオーバーフローデータ

ハードウェアカウンタオーバーフローのプロファイリングでは、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、プロファイルパケットが記録されます。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。一般にハードウェアカウンタは、命令キャッシュミスやデータキャッシュミス、CPU サイクル、浮動小数点演算、命令実行などのイベントの追跡に使用されます。カウンタがオーバーフロー値に達すると、標本コレクタは、そのオーバーフロー値とカウンタの種類を含むプロファイルパケットを記録します。

UltraSPARC III および IA ハードウェアには、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。標本コレクタは、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、オーバーフローを監視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。

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システム固有であるため、どのカウンタを利用できるかは、使用しているシステム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。標本コレクタには、利用可能なハードウェアカウンタのリストがあります。実際にハードウェアカウンタを利用する方法については、59 ページの「collect コマンドによるデータの収集」、64 ページの「Sun WorkShop 統合プログラミング環境からのデータの収集」、69 ページの「dbx の collector サブコマンドによるデータの収集」を参照してください。

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/0) 1000003  h=200003
```

最初のフィールドの "`CPU Cycles`" は、パフォーマンスアナライザにおける対応するメトリックの名前です。"=" 符号の左側に括弧で囲まれているのは、カウンタの別名 (上記の例では "`cycles`") です。"=" 符号の右側のフィールド (上記の例では "`Cycle_cnt/0`") は、`cputrack`(1) によって使用される内部名 (`Cycle_cnt`) と、そのカウンタに使用可能なレジスタ番号です。次のフィールドはデフォルトのオーバーフロー値、最後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を表します。

表 3-1 に、SPARC 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。

表 3-1　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

標準カウンタ名	メトリック名	説明
`cycles`	CPU Cycles0	CPU サイクル数 (レジスタ 0 でカウント)
`cycles1`	CPU Cycles1	CPU サイクル数 (レジスタ 1 でカウント)
`insts`	Instructions Executed0	実行命令数 (レジスタ 0 でカウント)
`insts1`	Instructions Executed1	実行命令数 (レジスタ 1 でカウント)

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  h=200003
```

"`Cycle_cnt`" は `cputrack`(1) で使用される内部名、文字列 "`Cycle_cnt Events`" は、パフォーマンスアナライザにおけるこのカウンタに対するメトリック名です。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。次のフィールドはデフォルトのオーバーフロー値、最後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を表します。

カウンタリストでは、別名のあるカウンタが最初に置かれ、その後にレジスタ 0 で使用可能なカウンタ、レジスタ 1 で使用可能なカウンタが続きます。別名のあるカウンタは、別名が付いた状態と別名がない状態の計 2 回現れます。別名がないものには、カウンタに異なるオーバーフロー値を割り当てることができます。

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リングについては、以下の制限事項があります。

- ハードウェアカウンタのオーバーフローデータの収集対象にできるプロセッサは、パフォーマンスツールが認識する、UltraSPARC III プロセッサや一部の IA プロセッサだけです。その他のプロセッサでは、ハードウェアカウンタオーバーフローのプロファイリングは行えません。

- 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。
- 1 つの実験でハードウェアカウンタのオーバーフローデータと時間ベースのデータを収集することはできません。
- `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。
- 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を行う場合は、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しないでください。使用した場合、パフォーマンス実験の結果は保証できません。

標本ポイント

標本コレクタでは、データを標本に分類することができます。それぞれの標本は、同じ実験内の時間間隔を表します。データを標本に分類するのは、コードの異なる部分を解析したり、コードのパフォーマンスの時間変化を調べたりする便利な手段です。

標本の境界としては、ヘッダー、タイムスタンプ、カーネルからの実行統計情報、アドレス空間データ (要求があった場合) からなる標本パケットが使用されます。標本ポイントで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。

標本パケットは、次の状況で記録されます。

- 「デバッグ」ウィンドウまたは `dbx` で設定されたブレークポイント位置
- 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)
- 「デバッグ」ウィンドウから「コレクタ」➤「新規標本」が選択されたか、「新規標本」ボタンがクリックされた場合 (手動サンプリングが選択されている場合)
- `collector_sample` の呼び出し時 (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) (56 ページの「プログラムからのデータ収集の制御」を参照)

- 指定した信号が送信されたとき (collect コマンドで -l オプションが使用されている場合) (62 ページの「実験制御関連のオプション」を参照)
- 収集が終了したとき

標本収集の間隔 (秒単位) は整数値で指定します。デフォルト値は 1 秒です。

大域情報

各標本パケットには、プログラムの大域情報が記録されます。この大域情報は、次の情報で構成されます。

- **実行統計** - ページフォルトおよび入出力データ、コンテキスト切り替え、およびさまざまなページ常駐 (ワーキングセットおよびページング) 統計情報で構成されます。この情報は、パフォーマンスアナライザの「実行統計」区画に表示されます (112 ページの「実行統計情報の表示」を参照)。
- **アドレス空間データ (任意)** - プログラムのアドレス空間の全セグメントに関するページ参照情報およびページ変更情報で構成されます。この情報は、パフォーマンスアナライザの「アドレス空間」区画に表示されます (113 ページの「アドレス空間情報の表示」を参照)。

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全体を「実験」といいます。新しい実験のデフォルト名は、test.1.er です。接頭辞の .er は必須で、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) という形式の名前が選択された場合、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 という名前の場合は、その後に mytest.2.er、mytest.3.er のようになります。標本コレクタはまた、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、その実験名に *n* の部分がない場合は、実験名の .er の前に .*n* (*n*=1) を挿入し、使用されていない名前が見つかるまで必要に応じて *n* をインクリメントします。

実験はグループにまとめることができます。こうしたグループは、実験グループファイルで定義します。このファイルは、1 行に 1 つの実験名からなるプレーンテキストファイルで、デフォルトでは現在のディレクトリに格納されます。実験グループファイルのデフォルト名は `test.erg` で、名前に `.erg` が含まれていない場合は、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。実験グループは `collect` コマンド (55 ページの「collect コマンドによるデータの収集」を参照) または `dbx` の `collector` サブコマンド (69 ページの「dbx の collector サブコマンドによるデータの収集」を参照) に指定できます。

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。実験グループ名 *group*`.erg` が指定された場合、実験名は *group*`.`*m*`.er` になり、実験名が指定された場合は、その実験名がデフォルト名よりも優先します。詳細は、78 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。

各実験は、UNIX コマンドでは簡単に操作できない 1 つのデータ構造体です。このため、実験をコピー、移動、削除するためのユーティリティが特別に用意されています。詳細は、165 ページの「実験の操作」を参照してください。

実験には、実験データの他に、プログラムが使用したロードオブジェクトが含まれます。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。

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。プロファイルパケットには、実験の種類 (時間ベースのプロファイリング、同期待ち監視、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)

に依存するデータと、プログラムの構造に依存するデータ (呼び出しスタック) が含まれます。実験の種類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。プロファイルパケットは、実験の LWP ごとに記録されます。

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リング実験の場合、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 150 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リング実験の場合は、LWP 1 つあたり毎秒 100K 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。このようにデータ転送速度が高速になるため、実験の記録先は、ネットワークに接続されたファイルシステムではなく、ローカルのファイルシステムにすることを推奨します。

実験データを確実に格納し、そのデータ量の多さのために、実験の記録中にプログラムが大きな影響を受けないようにするには、プロファイル間隔や同期待ちの監視しきい値、ハードウェアカウンタのオーバーフロー値などの、データ収集速度を制御するパラメータの選択に注意してください。

実験サイズを概算するにあたっては、使用する LWP 数と、プロファイリングを有効にした状態の予想実行時間を考慮してください。また、アーカイブファイルが使用するディスク領域のサイズも考慮します (前節を参照)。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。

標本コレクタは、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、そのデータを一時的に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。標本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。

プログラムからのデータ収集の制御

動的共有ライブラリ `libcollector.so` には、ユーザープログラム内でデータ収集の制御に利用可能な API ルーチンがいくつか含まれています。それらのルーチンは C 言語で書かれていて、次のように定義されています。

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

C または C++ からこれらの API を使用するには、次の文をインクルードします。

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

現在のところ、明示的な Fortran インタフェースはありません。Fortran プログラムから直接、このインタフェースを使用するには、API ルーチンを呼び出す各サブプログラムの先頭に次の指定を挿入します。

```
!$PRAGMA C(function-list)
```

function-list は、サブプログラム内で使用する API ルーチン名をコンマで区切ったリストです。プログラムのリンクでは、`-lcollector` を使用します。コンパイラの指令については『Fortran ユーザーズガイド』、C と Fortran 間のインタフェースについては『Fortran プログラミングガイド』を参照してください。

C のインクルードファイルには、データが収集されない場合に、本当の API ルーチンに対する呼び出しを省略するマクロが含まれており、その場合、ルーチンは動的には読み込まれません。Fortran では、これらの API ルーチンは、実際に呼び出されたときに動的に読み込まれ、その際のオーバーヘッドが発生します。パフォーマンスデータを収集しないときにこのオーバーヘッドが発生しないようにするには、API ルーチンと C のプログマ指令に対する呼び出しに対し、条件付きのコンパイルを行います。

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、標本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 ルーチン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。

以下では、データ収集に関係する 4 つの API ルーチンについて説明します。

collector_sample(char *name)

標本パケットを記録し、その標本に指定された名前を付けます。ただし、現在、パフォーマンスアナライザは、この名前を使用しません。Fortran から collector_sample を使用する場合は、引数として整数 "0" を渡します。この引数は、この C ルーチンによって NULL ポインタと解釈されます。

collector_pause()

実験へのデータの実際の書き込みを一時停止します。実験は開いたままです。

collector_resume()

実験へのデータの実際の書き込みを再開します。

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。

データ収集と解析のためのプログラムのコンパイル

使用されたコンパイラオプションに関係なく、コンパイルされたプログラムのデータを収集、解析することができますが、-g オプション (フロントエンドのインライン化を有効にするオプションで、C++ の場合は -g0) を使用しなかった場合は、パフォーマンスアナライザの一部の機能が利用できなくなります。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。

また、-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。

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。

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、標本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、標本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。

プログラムに関する制限事項

プログラムに関するパフォーマンスデータの収集では、利用できるシステムユーティリティにいくつか制限があります。以下に、標本コレクタがデータの収集に失敗する可能性がある状況をまとめます。

- プログラムが SIGPROF を使用するかマスクする場合、あるいは SIGPROF 用のハンドラをインストールする場合 - 時間ベースおよびハードウェアカウンタオーバーフローのプロファイリングに失敗する可能性があります。
- プログラムが SIGEMT を使用するかマスクする場合、あるいは SIGEMT 用のハンドラをインストールする場合 - 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に失敗する可能性があります。
- プログラムが libcpc(3) 内のエントリポイントを使用する場合 - 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に失敗する可能性があります。

- プログラムが ITIMER_PROF または ITIMER_REALPROF に setitimer(2) を使用する場合 - 時間ベースおよびハードウェアカウンタオーバーフローのプロファイリングに失敗する可能性があります。

collect コマンドによるデータの収集

collect コマンドを使用し、コマンド行から標本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、*program-arguments* はそのプログラムに対する引数です。

注 - プログラムのコンパイルで静的リンクを有効にした場合は、collect コマンドの実行に失敗し、エラーメッセージが返されます。

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リングが有効になり、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リング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、49 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。51 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リングに関する制限事項」も参照してください。

collect コマンドで、定期的な標本収集を行うことはできません。libcollector(3) API を使用した標本収集呼び出しがプログラムに含まれているか(56 ページの「プログラムからのデータ収集の制御」を参照)、-l オプションが設定されていない限り (62 ページの「実験制御関連のオプション」を参照)、標本ポイントはプロセスの開始時と終了時にのみ記録されます。

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、47 ページの「標本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リングが行われます。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リングが明示的に無効にされ、同期待ちの監視とハードウェアカウンタオーバーフローのプロファイリング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て終了します。

`-a`

アドレス空間データを収集します。標本ポイントでのみ収集されます (52 ページの「標本ポイント」を参照)。

注 - この機能は、IA ハードウェアでは利用できません。

`-h` *counter* [, *threshold* [, *counter2* [, *threshold2*]]]

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。

- 標準のカウンタ名
- `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。

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` コマンドを入力します。ハードウェアカウンタの一覧については、47 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。

threshold および *threshold2* には、オーバーフロー値を指定できます。これらの引数は次の値をとることができます。

- `h` - 指定したカウンタの高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 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。49 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リングに関する制限事項」も参照してください。

`-h` オプションと `-p` オプションを組み合わせることはできません。指定した場合、`collect` コマンドはエラーメッセージを出力して終了します。`-h` オプションが使用されると、デフォルトの `-p on` オプションは無効になります。

`-n`

標本コレクタを使用せずにターゲットを実行します。この場合、他のオプションはすべて無視されます。条件付きのデータ収集、たとえば、ジョブ内のすべての MPI プロセスではなく、一部のプロセスのデータを収集するためのスクリプトを簡略化するときに役立つオプションです。

`-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リングを無効にします。
- `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リングを有効にします。
- *value* - プロファイル間隔を指定した値に設定します (ミリ秒単位)。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リングについては、48 ページの「時間ベースのデータ」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 オプションと -p オプションを組み合わせることはできません。指定した場合、collect コマンドはエラーメッセージを出力して終了します。

-s *option*

同期待ち監視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監視を有効にします。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 - 同期待ちの監視を有効にし、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 - 同期待ちの監視を無効にします。
- on - デフォルトのしきい値 (実行時の測定で値を決定) で同期待ちの監視を有効にします。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。

実験制御関連のオプション

-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。

signal 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。

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 をプロセスに接続すると、dbx コマンドの `ignore PROF` によって、収集シグナルが確実に `collect` コマンドに渡されます。

`-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。

signal 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、標本コレクタは記録状態で起動します。それ以外の場合は、一時停止状態で標本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。

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ファイルを格納します。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*

この実験を *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*

記録する実験名として *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。実験名と標本コレクタにおける実験名の取り扱いについての詳細は、53 ページの「収集データの格納場所」を参照してください。

その他のオプション

-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。

Sun WorkShop 統合プログラミング環境からのデータの収集

注 - 「標本コレクタ」ウィンドウとその使用方法に関する情報は、オンラインヘルプに記載されています。ウィンドウのオンラインヘルプを表示するには、そのウィンドウで「ヘルプ」をクリックしてください。

Sun WorkShop 統合プログラミング環境からデータを収集するには、以下の操作を行います。

1. 「デバッグ」ウィンドウにプログラムを読み込みます。
2. 実行時検査機能がオフ (デフォルト) になっていることを確認します。

   メモリーの使用またはアクセス検査が有効になっている場合は、「セッション」タブまたは実行時検査ウィンドウにアイコンが表示されます。「検査」メニューに「無効」ではなく「有効」とある場合、実行時検査機能はオフです。「標本コレクタ」ウィンドウを開いたまま実行時検査を有効にしようとすると、エラーメッセージが表示されます。
3. 「デバッグ」ウィンドウのメニューバーから「ウィンドウ」➤「標本コレクタ」を選択します。
4. 「データ収集」ラジオボタンを使用し、データ収集を有効または無効にします。

   - 「オン」を選択すると、標本コレクタが有効になります。この場合、標本コレクタはプログラム実行終了後も動作を続け、以降、プログラムが実行されるたびに、新しい実験が作成されます。デフォルトでは、「標本コレクタ」ウィンドウを開いたとき、このボタンが選択されています。
   - 「オフ」を選択すると、標本コレクタが無効になります。「オン」を選択するか、「標本コレクタ」ウィンドウを再度開くまで、データの収集と格納は行われません。

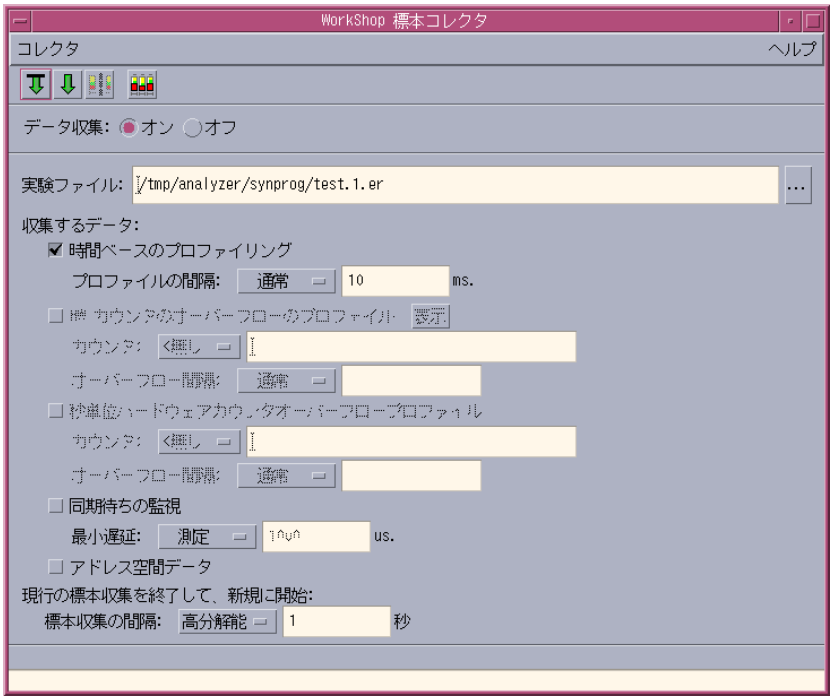


図 3-1　「標本コレクタ」ウィンドウ

5. 「実験ファイル」テキストボックスまたはブラウズボタンをクリックし、実験名を指定します。

   標本コレクタが提供するデフォルトの実験名は `test.1.er` です。これ以外の名前を使用する場合は、テキストボックスにその名前を入力するか、ブラウズボタンをクリックしてファイル選択ウィンドウを表示し、目的のディレクトリに移動して名前を選択します。実験名は、`.er` で終わる一意の名前である必要があります。

   実験名に *name*`.`*n*`.er` の形式の名前を使用した場合、以降の実験名では、整数 *n* の部分が自動的に 1 ずつインクリメントされます。たとえば、`test.1.er` の次は `test.2.er` になります。末尾が `.er` 以外の実験名を入力した場合は、警告メッセージが表示され、最後に有効であった名前がテキストボックスに再表示されます。実験名と標本コレクタにおける実験名の取り扱いについての詳細は、53 ページの「収集データの格納場所」を参照してください。

6. 時間ベースのプロファイル情報を収集するには、「時間ベースのプロファイリング」のチェックボックスが選択されている (デフォルト) ことを確認します。

プロファイル間隔を選択するには、「プロファイル間隔」リストボックスから以下のいずれかを選択します。

- 通常 - 10 ミリ秒のプロファイル間隔
- 高分解能 - 1 ミリ秒のプロファイル間隔
- カスタム - 独自にプロファイル間隔を設定できます (ミリ秒単位)。

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 (詳細は、48 ページの「時間ベースのデータ」を参照)。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リングについては、48 ページの「時間ベースのデータ」を参照してください。

7. **ハードウェアカウンタのオーバーフローに関するデータを収集するには、「HW カウンタのオーバーフローのプロファイル」チェックボックスをクリックし、カウンタとオーバーフロー値を選択します。**

   1 つ目のハードウェアカウンタを選択するには、次のいずれかを行います。

   - 「カウンタ」リストボックスから一般的なハードウェアカウンタの 1 つを選択する。
   - 「カウンタ」リストボックスから「その他」を選択し、「カウンタ」テキストボックスにカウンタ名を入力する。使用可能なカウンタの一覧を見るには、「表示」を選択します。「デバッグ」ウィンドウの出力区画にカウンタの一覧が表示されます。この一覧の形式については、49 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。

   注 - すべてのハードウェアカウンタはプラットフォームに依存します。このため、利用可能なカウンタの一覧の内容は、システムによって異なります。ハードウェアカウンタオーバーフローのプロファイリングをサポートしていないシステムの場合、このオプションは無効になります。

   オーバーフロー値を設定するには、「オーバーフロー間隔」オプションメニューから以下のいずれかを選択します。

   - 通常 - 通常値を選択します。
   - 高分解能 - 高分解能値を選択します。
   - カスタム - テキストボックスに独自の値を入力できます。

通常および高分解能値は、ハードウェアカウンタに依存します。これらの値は、ハードウェアカウンタリストで確認できます (49 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)。

1 つ目のハードウェアカウンタを選択すると、2 つ目のハードウェアカウンタ用のデータ入力が有効になります。2 つ目のハードウェアカウンタを選択するには、1 つ目のハードウェアカウンタで使用した手順と同じです。

大部分のハードウェアカウンタは、1 つのレジスタでのみカウントを行います。2 つ目のカウンタを選択した場合は、必ず、1 つ目のカウンタとは異なるレジスタでカウントが行われるようにします。このようにしなかった場合は、エラーメッセージが表示されます。ハードウェアカウンタリストには、レジスタ番号が示されます。

8. 同期待ちの監視データを収集するには、「同期待ちの監視」をクリックします。

   監視を開始するしきい値を指定するには、最小遅延のオプションメニューから次のいずれかを選択します。

   - 測定 - 実行時にしきい値が決定されます。詳細は、49 ページの「同期待ち監視データ」を参照してください。
   - 1000 us
   - 100 us
   - 全部 - しきい値をゼロに設定し、すべての同期待ちを監視します。
   - カスタム - テキストボックスに独自の値を入力できます (マイクロ秒単位)。

9. アドレス空間のメモリー割り当てに関する情報を収集するには、「アドレス空間データ」チェックボックスをクリックします。

   注 - この機能は、IA ハードウェアでは利用できません。

   ---

10. 標本収集の間隔を設定するには、標本収集の間隔のオプションメニューから次のいずれかを選択します。

    - 通常 - 10 秒間隔
    - 高分解能 - 1 秒間隔
    - カスタム - 独自に間隔を設定できます (秒単位)。

- 手動 - 「標本コレクタ」ウィンドウから「収集」➤「新規標本」を選択するか、「新規標本」ボタンをクリックすることによって標本ポイントを記録します。

注 - 非同期入出力ライブラリの `libaio.so` を使用する場合は、手動の標本収集を使用する必要があります。

---

11. 以下のいずれかを行うことによって、プログラムを実行してデータを収集します。

- 「標本コレクタ」ウィンドウから「収集」➤「開始」を選択する。
- 「標本コレクタ」ウィンドウまたは「デバッグ」ウィンドウのツールバーにある「開始」ボタンをクリックする (プログラムの先頭からデータの収集を開始する場合)。

- 「標本コレクタ」ウィンドウまたは「デバッグ」ウィンドウのツールバーにある「継続」ボタンをクリックする (一時停止したプログラムの実行を再開する場合)。

dbx の `collector` サブコマンドによるデータの収集

dbx から標本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、データの収集を有効にします。

```
(dbx) collector subcommand
```

collector に使用可能な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、標本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。

address_space { on | off }

アドレス空間データ (参照または修正されたページ) の収集を有効または無効にします。デフォルトは off です。

注 - この機能は、IA ハードウェアでは利用できません。

---

hwprofile *option*

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リング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リングを有効にします。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リングを無効にします。
- `list` - この一覧の形式については、49 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。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 つ目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。

option のデフォルト値は `off` です。

時間ベースのプロファイリングを有効にしていて、`hwprofile` サブコマンドが使用された場合は、警告メッセージが表示され、時間ベースのプロファイリングが無効になります。

51 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リングに関する制限事項」も参照してください。

`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` - 時間ベースのプロファイリングを有効にします。実験がアクティブな場合、このオプションは無視されて、警告メッセージが表示されます。
- `off` - 時間ベースのプロファイリングを無効にします。
- `timer` *value* - プロファイル間隔を指定した値に設定します (ミリ秒単位)。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リングについては、48 ページの「時間ベースのデータ」を参照してください。

option のデフォルト値は `on` です。

`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リングを有効にしている場合に、`profile` サブコマンドが使用された場合は、警告メッセージが表示され、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が無効になります。

`sample` *option*

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

- `periodic` - 定期的な標本収集を有効にします。必ず `period` オプションと併用する必要があります。
- `manual` - 手動の標本収集を有効にします。
- `period` *value* - 標本収集の間隔を *value* に設定します (秒単位)。

option のデフォルト値は、標本収集の間隔が 1 秒の `periodic` です。

注 - 非同期入出力ライブラリの `libaio.so` を使用する場合は、手動の標本収集を使用する必要があります。

`synctrace` *option*

同期待ちの監視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` - 同期待ちの監視を有効にします。
- `off` - 同期待ちの監視を無効にします。
- `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監視されます。デフォルト値は `calibrate` です。

option のデフォルト値は `off` です。

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無効の場合、このサブコマンドは無視されて警告が出されます。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、データ収集が無効の場合は、データ収集が有効になり、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。データの収集がアクティブな場合、このサブコマンドは無視されます。

出力関連のサブコマンド

`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。
- `filename` *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。実験名と標本コレクタにおける実験名の取り扱いについての詳細は、53 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。

情報関連のサブコマンド

`show`

標本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

`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、このサブコマンドは無視されて警告が出されます。

quit

`disable` と同じです。

動作中のプロセスからのデータの収集

標本コレクタでは、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 (CLI または GUI のどちらでも可) の制御下にある場合は、プログラムを一時停止し、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。

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。

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。

   - 「デバッグ」ウィンドウからの場合は、「デバッグ」➤「プロセスを接続」を選択し、ダイアログで接続する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、次のコマンドを使用します。

```
% 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。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。

- 「デバッグ」ウィンドウからの場合は、「ウィンドウ」➤「標本コレクタ」を選択して、ダイアログで標本収集のパラメータを設定してから、「実行」➤「継続」を選択するか、「継続」ボタンをクリックします。
- dbx からの場合は、collector コマンドを使用して標本収集の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。

- 「デバッグ」ウィンドウからの場合は、「実行」➤「プロセスを切り離す」を選択します。
- dbx からの場合は、次のコマンドを入力します。

```
(dbx) detach
```

同期待ちの監視データを収集する場合は、プログラムを実行する前に、標本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同期ルーチンにラッパーが提供されるためです。

このライブラリ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。ライブラリ名の設定には LD_PRELOAD 環境変数、ライブラリパスの設定には LD_LIBRARY_PATH を使用してください。

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、LD_LIBRARY_PATH64 環境変数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 3-2 に、これらの環境変数に設定する値をまとめます。

表 3-2 libcollector.so ライブラリを事前に読み込むための環境変数の設定

環境変数	値
LD_PRELOAD	libcollector.so
LD_LIBRARY_PATH	/opt/SUNWspro/lib
LD_LIBRARY_PATH64	/opt/SUNWspro/lib/v9

/opt/SUNWspro 以外のディレクトリに Sun WorkShop 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標本コレクタを有効にする必要があります。MPI ジョブ内のすべてのプロセスに dbx を接続すると、各プロセスと停止と再開が異なるタイミングで行われ、この時間差によって、MPI プロセス間の対話状態が変化し、収集するパフォーマンスデータに影響が出ることがあります。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。78 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。

MPI プログラムからのデータの収集

標本コレクタは、Sun MPI (Message Passing Interface) 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™ に付属しています。ClusterTools は、バージョン 3.1 またはその互換バージョンを使用してください。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。詳細は、Sun HPC 3.1 AnswerBook™ Collection に付属している Sun High-Performance Computing (HPC) ClusterTools 3.1 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 と標本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 MPI ジョブのデータの収集方法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、53 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、共通の 1 つの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。

実験名の標準形式である *experiment*.*n*.er を使用し、ローカルの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 という 4 つの単一プロセッサノードを持つマシンで MPI ジョブを実行し、このすべてのノードに /scratch というローカルディスクがあり、それらのディスクの *username* ディレクトリに実験を格納した場合、MPI ジョブによって作成される実験のフルパス名は以下のようになります。

```
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 ジョブの完了後に共通の場所に実験ファイルを移動する場合は、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、165 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、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が、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、54 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逆アセンブリコード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 (または同じパスとタイムスタンプを持つコピー) にアクセスする必要があります。

MPI の制御下での `collect` コマンドの実行

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

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

n は、MPI によって作成されるプロセス数です。このコマンドによって、*n* 個のインスタンスの `collect` が作成され、インスタンスごとに実験ファイルが 1 つ作成されます。実験ファイルの格納場所と格納方法については、53 ページの「収集データの格納場所」を参照してください。

MPI の実行ごとに、それぞれの実験セットが別々に格納されるようにするには、それらの MPI の実行ごとに `-g` オプションを使用し、1 つの実験グループを作成するようにします。このとき、実験グループは、関係するすべての MPI プロセスからアクセス可能なファイルシステムに格納します。実験グループを作成することによって、1 回の MPI 実行用の実験セットのアナライザへの読み込みが簡単になります。この他、MPI の実行ごとに `-d` オプションを使用し、異なるディレクトリを作成するという方法もあります。

MPI の制御下で dbx を起動することによるデータの収集

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

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

n は MPI によって作成されるプロセス数、*collection-script* はデータ収集の設定と開始に必要なコマンドからなる dbx スクリプトです。このコマンドによって、*n* 個のインスタンスの dbx が作成され、それらのインスタンスごとに 1 つの MPI プロセスに関する 1 つの実験ファイルが作成されます。実験名を指定しなかった場合は、実験に MPI ランクのラベルが付けられます。実験ファイルの格納場所と格納方法については、78 ページの「MPI 実験ファイルの格納」を参照してください。

標本収集スクリプトを利用し、プログラムで MPI_Comm_rank() を呼び出すことによって、実験ファイルに MPI ランクに基づく名前を付けることができます。たとえば、C プログラムの場合は、次の行を挿入します。

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

また、Fortran プログラムの場合は、次の行を挿入します。

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

たとえば、この呼び出しを 17 行目に挿入した場合、次のようなスクリプトを使用できます。

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

第4章

パフォーマンスアナライザ GUI を使用したプログラムのパフォーマンス解析

この章では、パフォーマンスアナライザとそのパフォーマンスメトリック、およびその使用方法について説明します。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。パフォーマンスアナライザを利用することによって、実験データを検査・操作し、プログラム実行時の問題点を発見してプログラムのパフォーマンスを解析、改善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。

- パフォーマンスメトリックの種類と目的
- プログラム構造へのメトリックの対応付け
- パフォーマンスアナライザの実行
- 実験の選択
- フィルタによる表示するデータの選別
- 関数やロードオブジェクトのメトリックの表示
- 関数の呼び出し元と呼び出し先メトリックの表示
- 注釈付きソースコードと逆アセンブリコードの表示
- マップファイルの作成と利用
- 標本情報の表示
- 実行統計情報の表示
- アドレス空間情報の表示
- 表示内容の印刷

パフォーマンスアナライザの使用法の概略と、パフォーマンスアナライザを使用したアプリケーションのチューニング例については、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 6 章を参照してください。

パフォーマンスメトリックの種類と目的

パフォーマンスアナライザは、標本コレクタの収集した raw データを読み込み、メトリック (計測データ) というプログラムのパフォーマンスを表す値 (単位量) に変換します。3 種類のプロファイルデータが 3 種類のメトリックに変換されます。クロックベースのデータはタイミングメトリック、同期待ちの監視データは同期メトリック、ハードウェアカウンタのオーバーフローデータはカウントメトリックにそれぞれ変換されます。

タイミングメトリック

クロックベースのデータからは以下のメトリックが得られます。

- **ユーザー CPU 時間** - LWP がユーザーモードで CPU を使用した時間
- **時計時間** - LWP 1 で費やした時間。一般的な「時間」です。
- **全 LWP 時間** - 全 LWP 時間の合計
- **システム CPU 時間** - LWP がカーネルモードで CPU を使用したか、トラップ状態にあった時間
- **CPU 待ち時間** - LWP が CPU 待ちに費やした時間
- **テキストページフォルト時間** - LWP がテキストページ待ちに費やした時間
- **データページフォルト時間** - LWP がデータページ待ちに費やした時間
- **その他の待ち時間** - LWP がロックまたはカーネルページ待ち、スリープに費やした時間、または停止していた時間

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。
- 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 (108 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。

同期遅延メトリック

同期待ちの監視データから得られるメトリックは以下のとおりです。

- **同期遅延イベント数** - 待ち時間が所定のしきい値を超えた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待ち時間の合計。

この情報から、関数またはロードオブジェクトが頻繁に待ち状態になっているかどうか、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。

カウントメトリック

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。

一般的なハードウェアカウンタとしては、命令キャッシュミス数、データキャッシュミス数、サイクル数、キャッシュストールサイクル数、浮動小数点演算数、発行または実行命令数をカウントするカウンタがあります。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。

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。あらゆる命令がそれぞれ 1 つのソースコード行に対応付けられ、その命令に割り当てられたメトリックも同じソースコード行に対応付けられます。この仕組みについての詳細は、第 6 章を参照してください。パフォーマンスアナライザにおけるソースコードおよび命令レベルのメトリックの表示方法については、106 ページの「注釈付きソースコードと逆アセンブリコードの表示」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である関数およびロードオブジェクトにも対応付けられます。呼び出しスタックには、プロファイルが取られたときに記録された命令アドレスに達するまでに行われた、一連の関数呼び出しに関する情報が含まれます。パフォーマンスアナライザは、この呼び出しスタックを使用し、プログラム内の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経路に関する情報を提供します。これに対し、関数の属性メトリックは、関数を通る特定の経路に関する情報を提供し、特定の関数呼び出しが原因になっているメトリックを示します。このマニュアルでは、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」と定義しています。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を合計すると、その関数の包括的メトリックになります。
- 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 4-1 は、部分的にですが、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。

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。

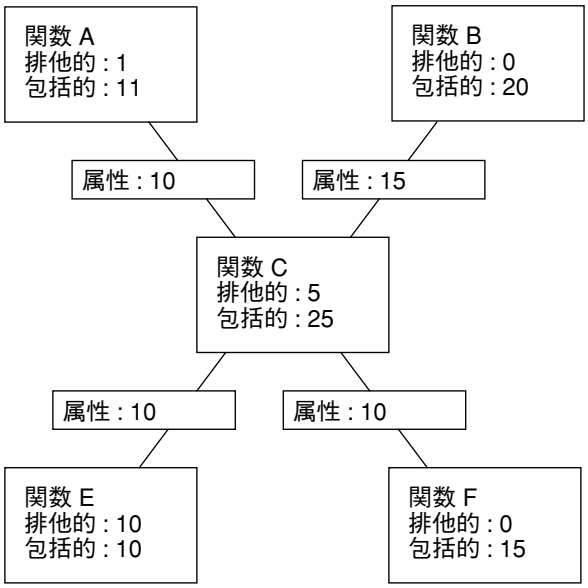


図 4-1　呼び出しツリーにおける排他的、包括的、属性メトリックの関係

関数 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 が 10 単位、関数 B が 15 単位、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全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。

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、コマンド行または Sun WorkShop 統合プログラミング環境のどちらからでも起動できます。

Sun WorkShop 統合プログラミング環境からパフォーマンスアナライザを起動する方法は、次の 3 つあります。

- メニューバーから「ツール」➤「アナライザ」を選択する。

  メニューが表示され、このメニューから、新しいアナライザウィンドウを開くか、特定の実験を読み込むか、実験リストを編集するかを選択できます。詳細は、Sun WorkShop オンラインヘルプの「メインウィンドウ」の「ツールメニュー」を参照してください。

- メインウィンドウのツールバーにある「解析」ボタンをクリックする。

パフォーマンスアナライザによって、最後に解析した実験が自動的に読み込まれます。それまでに実験が解析されたことがない場合は、「実験ファイルの読み込み」ダイアログボックスが表示され、実験を選択できます。

- 「標本コレクタ」ウィンドウのツールバーにある「解析」ボタンをクリックする。

パフォーマンスアナライザによって、最後に収集が行われた実験が自動的に読み込まれます。

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 [-s session-id] [-V] [experiment-name [, experiment-name2 … ]]
```

実験名については、53 ページの「収集データの格納場所」を参照してください。実験名が省略した場合は、パフォーマンスアナライザが起動したときに「実験ファイルの読み込み」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。実験名ではなく実験グループ名を指定することによって、実験グループを読み込むこともできます。

表 4-1 に、analyzer コマンドのオプションをまとめます。

表 4-1 analyzer コマンドのオプション

-s *session-name*	パフォーマンスアナライザのインスタンスにユーザー定義の識別子を割り当てます。
-V	パフォーマンスアナライザのバージョン番号を stdout に出力します。

下図のアナライザウィンドウは、パフォーマンスアナライザを起動したときに表示されるアナライザのメインウィンドウの例です。このウィンドウには、メインメニューバー、上部および下部のツールバー、2 つのパネルからなるデータ表示区画があります。

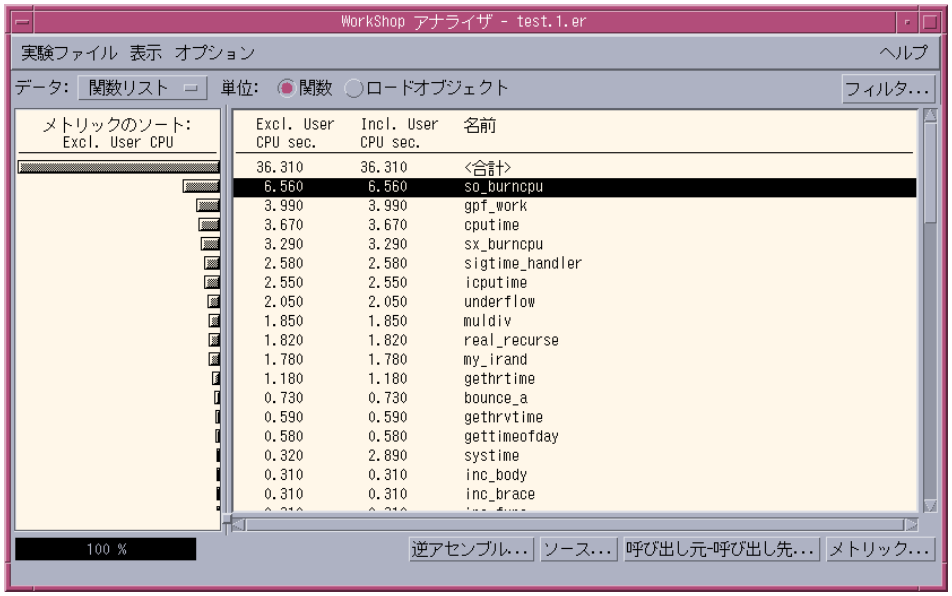


図 4-2　「アナライザ」ウィンドウ

パフォーマンスアナライザが表示可能なデータの種類は、大きく分けて、関数リスト、概要、アドレス空間、実行統計の 4 つがあります。これらの種類のデータについては、この後の節で説明します。どの種類のデータを表示するかは、上部のツールバーにある「データ」メニューから選択できます。データおよび機能の大半は、関数リストから使用することができます。デフォルトでは、パフォーマンスアナライザを起動すると、関数リストが表示されます。

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「表示」➤「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「実験ファイル」➤「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。

パフォーマンスアナライザを複数実行している場合は、それぞれのインタンスに異なるセッション名を割り当てない限り、それらインスタンスによって 1 つのエディタウィンドウが共有されます。エディタウィンドウは、注釈付きのソースまたは逆アセ

ンブリコードの表示に使用するウィンドウです。パフォーマンスアナライザのインスタンスに名前を付けることによって、実験ごとに異なるエディタウィンドウを使用し、注釈付きソースや逆アセンブリコードを比較することができます。パフォーマンスアナライザのインスタンスへのタグ付けは、`analyzer` コマンドを使用してのみ行うことができます。

パフォーマンスアナライザを終了するには、「アナライザ」ウィンドウのメニューバーから「実験ファイル」➤「終了」を選択します。

注 - 「パフォーマンスアナライザ」ウィンドウとその使用方法については、オンラインヘルプを参照してください。ウィンドウのオンラインヘルプを表示するには、そのウィンドウで「ヘルプ」をクリックしてください。

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。

パフォーマンスアナライザへの実験の読み込み

実験を読み込むと、パフォーマンスアナライザから現在の実験データのすべてが消去され、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。)

パフォーマンスアナライザへの実験または実験グループの読み込みは、以下の手順でいつでも行うことができます。

1. **「アナライザ」ウィンドウのメニューバーから「実験ファイル」➤「読み込み」を選択し、「実験ファイルの読み込み」ダイアログを開きます。**

   Sun WorkShop からパフォーマンスアナライザを起動したか、実験名を指定しないでコマンド行からパフォーマンスアナライザを起動した場合は、「実験ファイルの読み込み」ダイアログが自動的に開きます。

2. 「実験ファイルの読み込み」ダイアログのリストボックスから実験または実験グループ名をダブルクリックして選択するか、テキストボックスに名前を入力します。

   「ヘルプ」ボタンをクリックすると、このダイアログのナビゲート方法が表示されます。

パフォーマンスアナライザへの実験の追加

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に便利です。

パフォーマンスアナライザにすでに読み込まれている実験に実験を追加するには、以下の操作を行います。

1. 「アナライザ」ウィンドウのメニューバーから「実験ファイル」➤「追加」を選択し、「実験ファイルの追加」ダイアログを開きます。
2. 「実験ファイルの追加」ダイアログで、追加する実験ファイルをダブルクリックして選択するか、テキストボックスに実験名を入力します。

   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。

パフォーマンスアナライザからの実験の解除

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すべてのデータが消去され、メトリックが再計算されます。(実験ファイルそのものが、この消去の影響を受けることはありません。)

パフォーマンスアナライザから実験レコードを解除するには、以下の操作を行います。

1. 「実験ファイル」➤「解除」を選択し、「実験ファイルの解除」ダイアログを開きます。

2. リストボックスから、パフォーマンスアナライザから解除する実験ファイルをクリックして選択します。
3. ダイアログを閉じないで実験ファイルを解除するには、「適用」をクリックします。実験レコードを解除してダイアログを閉じるには、「了解」をクリックします。

   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。

   パフォーマンスアナライザから実験を解除できるのは、複数の実験ファイルが読み込まれている場合だけです。実験ファイルが 1 つしか読み込まれていない場合、「解除」コマンドは使用できません。

フィルタによる表示するデータの選別

問題があると考えられるプログラム部分に注目できれば、作業を効率的に進めることができます。パフォーマンスアナライザでは、複数の方法で実験情報にフィルタをかけることができます。

- 実験、標本、スレッド、LWP による選別
- ロードオブジェクトによる選別

実験、標本、スレッド、LWP の選択

メトリックを表示する実験、標本、スレッド、LWP を指定することによって、表示する情報を制限することができます。この場合、「関数リスト」や「概要」の表示には、指定した標本、スレッド、LWP のメトリックだけが表示されます。

表示する情報を制限するには、以下に示す手順を使用します。一部手順を省略したり、繰り返したりできます。

1. 「アナライザ」ウィンドウにある「フィルタ」ボタンをクリックします。

   「フィルタ」ダイアログが表示されます。

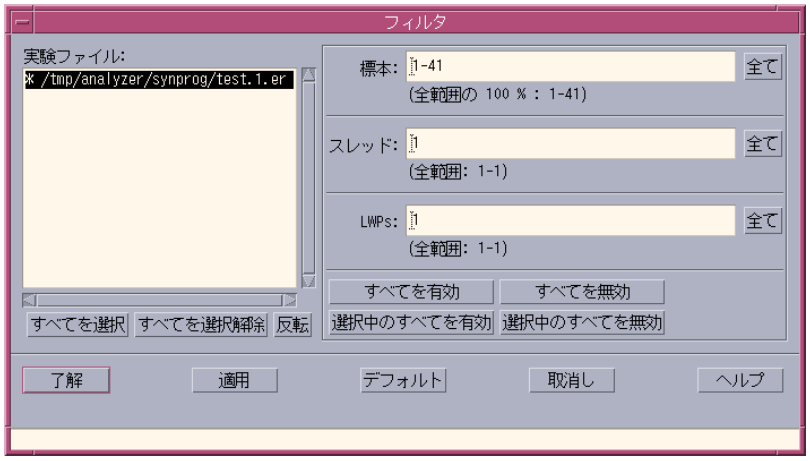


図 4-3　「フィルタ」ダイアログボックス

2. データを表示する標本、スレッド、LWP を指定するには、それぞれのテキストボックスに番号を入力するか (複数の番号を入力する場合はコンマで区切ります)、「すべて」をクリックします。

   2 つの番号をハイフンで区切って番号範囲を指定することもできます。標本、スレッド、LWP は任意の組み合わせでいくつでも指定できますが、スレッドおよび LWP は、アプリケーションにとって意味のある組み合わせにしてください。

3. 実験を選択するには、「実験ファイル」リストボックスとその下にある3つのボタンを利用します。

   - すべてを選択 - すべての実験を選択状態にします。
   - すべてを選択解除 - すべての実験を選択解除します。
   - 反転 - 選択されていない実験を選択状態にして、選択状態に実験を選択解除します。

4. 選択した標本、スレッド、LWP を実験に適用するには、テキストボックスの下にある 4 つのボタンを使用します。

   - すべてを有効 - 実験が選択されているかどうかに関係なく、指定された標本、スレッド、LWP をすべての実験に適用します。
   - 選択中のすべてを有効 -「実験ファイル」リストボックスで選択されている実験に、指定された標本、スレッド、LWP を適用します。

- すべてを無効 - 実験が選択されているかどうかに関係なく、すべての実験のデータ表示を無効にします。このオプションを選択した場合、データは表示されません。
- 選択中のすべてを無効 -「実験ファイル」リストボックスで選択されている実験に対するデータ表示を無効にします。標本、スレッド、LWP の指定は無視されます。

実験ごとに固有のスレッド、標本、LWP を指定することができます。実験に対して同じ設定内容を複数回指定した場合は、最後の設定内容が他のすべての設定内容に優先します。

5. 現在の設定内容を適用して別の設定を行う場合は、「適用」をクリックします。
6. 現在の設定内容を適用して「フィルタ」ダイアログを終了する場合は、「了解」をクリックします。

ロードオブジェクトの選択

多くの場合、パフォーマンスを解析するために、プログラムのすべてのロードオブジェクトに関する情報を表示する必要はありません。たとえば、プログラムファイルに関係するメトリックだけが必要で、システムライブラリに関係するメトリックは必要ない場合などがあります。パフォーマンスアナライザでは、「関数リスト」や「概要」にメトリックを表示するロードオブジェクトを指定することができます。

メトリックを表示するロードオブジェクトを選択するには、以下の操作を行います。

1. 「表示」➤「次の条件を含んだロードオブジェクトを選択」を選択し、「次の条件を含んだロードオブジェクトを選択」ダイアログを開きます。
2. リストボックスから表示しないファイルをクリックして選択解除します。表示するファイルが選択されていない場合は、クリックして選択します。リストのすべてのロードオブジェクトを選択または選択解除するには、「すべてを選択」および「すべてを選択解除」ボタンをクリックします。
3. 「了解」をクリックして現在の設定内容を適用し、「次の条件を含んだロードオブジェクトを選択」ダイアログを閉じます。

関数やロードオブジェクトのメトリックの表示

パフォーマンスアナライザを起動すると、デフォルトでは、「アナライザ」ウィンドウに関数リストが表示されます。関数リストには、関数およびロードオブジェクトのメトリックデータが表示されます。データ表示区画は、次の 2 つのパネルで構成されています。

- 左のパネル - データをソートしたメトリックのヒストグラムが表示されます。
- 右のパネル - 関数またはロードオブジェクトのメトリックテーブルが表示されます。テーブルの各行の右端が、その行に示されているデータが該当する関数またはロードオブジェクトの名前です。

関数とロードオブジェクトのデータ表示を切り換えるには、上部のツールバーにある「関数」および「ロードオブジェクト」ラジオボタンを使用します。

関数リストにデフォルトで表示されるメトリックは、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、ファイル名が `.er.rc` である必要があり、`er_print` コマンドを含めることができます。詳細は、132 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。システムのデフォルト値ファイルには、表示するメトリックとして以下のメトリックを指定できます (実験にこれらのメトリックが存在する場合)。

- 排他的ユーザー CPU 時間
- 包括的ユーザー CPU 時間
- 排他的ハードウェアカウンタオーバーフローのプロファイルメトリック
- 包括的ハードウェアカウンタオーバーフローのプロファイルメトリック
- 包括的同期待ち時間
- 包括的同期待ち回数

関数やロードオブジェクトのすべてのメトリックについて、「関数リストメトリック」ダイアログに表示する項目を選択したり、ソート基準を指定したりできます。

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。表示形式を選択するには、「オプション」➤「関数名の書式を設定」を選択します。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。

関数やロードオブジェクトのメトリックとソート順序の選択

プログラムのパフォーマンスが特定の問題の影響を受けていると考えられる場合は、その問題に関係しているメトリックだけが関数リストに表示されるように設定することができます。

- **関数リストに表示するメトリック、メトリックの単位、ソート順序、列の順序を変更するには、上部のツールバーにある「メトリック」ボタンをクリックします。**

「関数リストメトリック」ダイアログが表示されます。

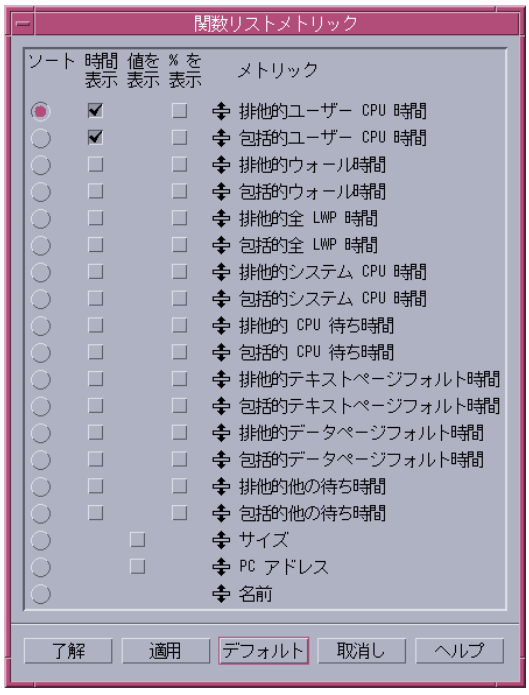


図 4-4　「関数リストメトリック」ダイアログ

このダイアログは、パフォーマンスアナライザに読み込まれた実験セットで選択可能なメトリックの一覧です。これらのメトリックを任意の組み合わせで表示することができます。また、サイズ (バイト単位) とプログラムカウンタのアドレスを表示することもできます。関数またはロードオブジェクトの名前は常に表示されます。

「アナライザ」ウィンドウのデータ表示区画に表示されるメトリック列の順序は、このダイアログの一覧内のメトリックの順序によって決まり、この一覧内のメトリックの順序は、当初、デフォルト値ファイルに基づいています。

すべてのメトリックが、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) の形式で表示することができます。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することができます。

1. 表示するメトリックとその単位を選択するには、「時間表示」、「値を表示」、「%を表示」列の適切なチェックボックスをクリックします。
2. ソート順序を指定するには、「ソート」列の適切なラジオボタンをクリックします。
3. 一覧内のメトリックの順序を変更するには、メトリック名の左横のアイコンをドラッグ & ドロップします。移動したメトリックは、ドロップ先のメトリックの上に表示されます。
4. 関数リストに新しい設定内容を適用して、「メトリック」ダイアログを閉じる場合は「了解」を、「メトリック」ダイアログを閉じないで関数リストに新しい設定内容を適用する場合は「適用」をクリックします。

関数やロードオブジェクトの概要メトリックの表示

「概要メトリック」ウィンドウを使用し、特定の関数またはオブジェクトのメトリックセット全体とその他の情報を、関数リストの一部としてではなく表形式で表示することができます。

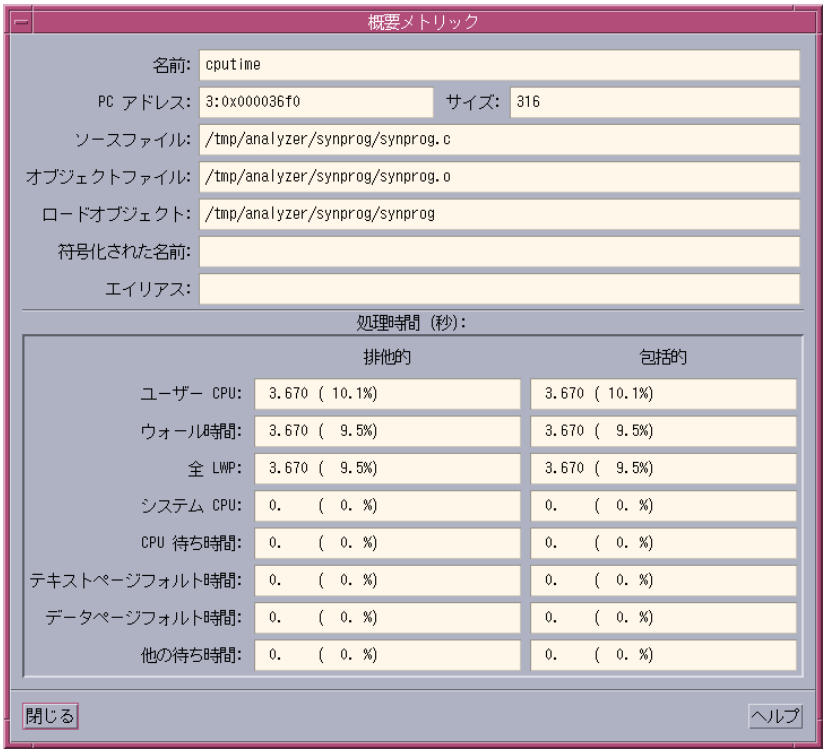


図 4-5　「概要メトリック」ウィンドウ

関数またはロードオブジェクトの概要メトリックを表示するには、以下の操作を行います。

1. 「単位」のラジオボタンを使用し、「関数」と「ロードオブジェクト」のどちらを表示するのかを選択します。
2. 関数リストから関数またはロードオブジェクトをクリックして選択します。
3. 「表示」➤「概要メトリックを表示」を選択し、「概要メトリック」ウィンドウを開きます。

「概要メトリック」ウィンドウは 2 つの区画から構成されています。上の区画には、選択された関数またはロードオブジェクトの名前、アドレス、サイズが表示され、関数の場合は、その関数のコードが存在するソースファイル、オブジェクトファイル、ロードオブジェクトも表示されます。下の区画は、実験ファイルに記録されている、

選択された関数またはロードオブジェクトのすべてのメトリック (排他的および包括的) の一覧です。この表示が、関数リストで選択されているメトリックの影響を受けることはありません。

注 - 「概要メトリック」ウィンドウのデータは、全体をクリップボードにコピーして、テキストエディタにペーストすることができます。

関数やロードオブジェクトの検索

パフォーマンスアナライザには、検索ツールが用意されており、このツールを使って、関数リストから特定の関数またはロードオブジェクトを探すことができます。

特定の関数またはロードオブジェクトを検索するには、以下の操作を行います。

1. メニューバーから「表示」➤「検索」を選択し、「検索」ダイアログを表示します。

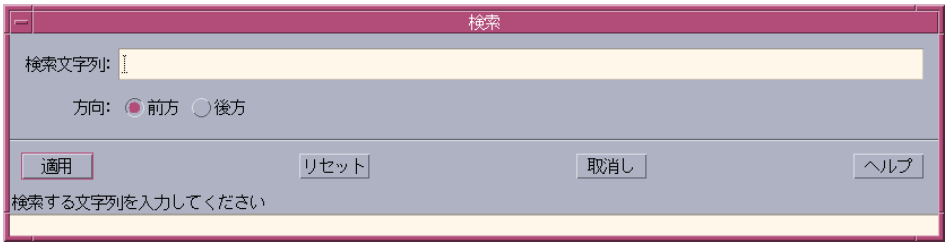


図 4-6　「検索」ダイアログ

2. 「検索文字列」フィールドに検索する文字列を入力します。

注 - パフォーマンスアナライザの検索機能では、UNIX の正規表現を利用できます。たとえば、`c` を任意の 1 文字とした場合、`c*` は、`c` の後に任意の個数の文字が続く文字列ではなく、任意の個数の `c` を示します。UNIX の正規表現についての詳細は、`regexp(5)` のマニュアルページを参照してください。

---

3. 「方向」のラジオボタンを使用して検索方向を指定できます。デフォルトの検索方向は「前方」です。

4. 「適用」をクリックします。

一致する文字列が見つかると、関数リスト内のその関数のデータ行が強調表示され、その関数が区画の先頭に来ます。

5. 検索文字列に一致する他の関数名を検索するには、再度「適用」をクリックします。

   関数リストの末尾に達すると、リストの先頭に戻って検索が行われます。

6. 「検索文字列」テキストボックスの文字列を前回見つかった文字列に戻すには、「リセット」をクリックします。

関数の呼び出し元と呼び出し先メトリックの表示

特定の関数の呼び出し元と呼び出し先のメトリックを表示するには、以下の操作を行います。

1. 関数リストから関数をクリックして選択します。
2. 下部のツールバーにある「呼び出し元-呼び出し先」ボタンをクリックします。

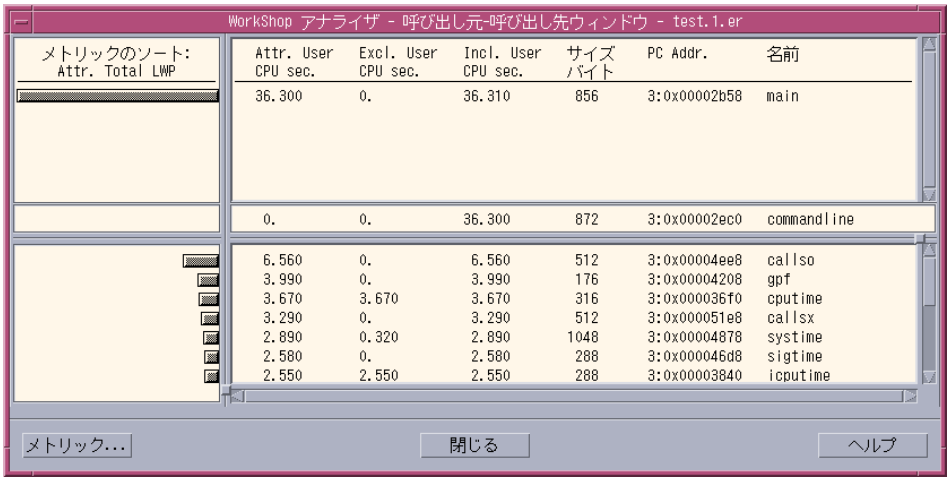


図 4-7　「呼び出し元-呼び出し先」ウィンドウ

「呼び出し元-呼び出し先」ウィンドウでは、選択した関数の情報が中央の区画、関数の呼び出し元の情報が上の区画、関数の呼び出し先の情報が下の区画にそれぞれ表示されます。これらの区画はそれぞれ 2 つのパネルに分かれています。

- 左のパネル - データをソートしたメトリックのヒストグラムが表示されます。
- 右のパネル - 関数のメトリックテーブルで、各行の右端に、その行に示されているデータが該当する関数の名前が表示されます。

「呼び出し元-呼び出し先」ウィンドウには、特定の関数の排他的、包括的、属性の各メトリックを表示することができます。標本コレクタによってデータが収集されている場合は、関数の呼び出し元と呼び出し先も表示できます。これらのメトリックはそれぞれ、時間またはイベント発生回数のいずれかと百分率の形式で表示できます。循環型のハードウェアカウンタのメトリックは、時間とイベント発生回数、百分率の形式で表示できます。属性メトリックの場合は、呼び出し元と呼び出し先の両方について、属性メトリックが関数の包括的メトリックに占める割合を示す百分率値が表示されます。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。

デフォルトのメトリックは、関数リストに含まれているメトリックから取られます。実験ファイルに存在する場合は、デフォルトのメトリックとして次のメトリックが表示されます。

- 属性ユーザー CPU 時間
- 排他的ユーザー CPU 時間
- 包括的ユーザー CPU 時間
- 属性ハードウェアカウンタオーバーフローのプロファイルメトリック
- 排他的ハードウェアカウンタオーバーフローのプロファイルメトリック
- 包括的ハードウェアカウンタオーバーフローのプロファイルメトリック
- 属性同期待ち時間
- 包括的同期待ち時間
- 属性同期待ち回数
- 包括的同期待ち回数

各関数のメトリックは、そのメトリックに従って秒単位の時間または回数の形式で表示されます。行は、デフォルト値リスト内の先頭の属性メトリックを基準にソートされます。

呼び出し元区画または呼び出し先区画いずれかの関数をクリックすることによって、プログラム構造内をナビゲートすることができます。このとき表示は、新たに選択された関数が中央に来るように変更されます。排他的、包括的、属性の各メトリックを観察することによって、特定のメトリックで大きな割合を占めている関数を特定することができます。

再帰関数呼び出しの場合、関数は、それ自身が呼び出し元にも、呼び出し先にもなることができます。「呼び出し元-呼び出し先」ウィンドウでは、再帰関数は、呼び出し先ではなく、それ自身の呼び出し元として表示されます。

「呼び出し元-呼び出し先」ウィンドウにおけるメトリックとソート順序の選択

「呼び出し元-呼び出し先」ウィンドウには、排他的および包括的メトリックだけでなく属性メトリックも表示するため、「呼び出し元-呼び出し先」ウィンドウには、メトリックとソート順序を選択するための専用のダイアログが用意されています。「呼び出し元-呼び出し先」ウィンドウに表示するデータとそのソート順序を指定するには、その「呼び出し元-呼び出し先」ウィンドウにある「メトリック」をクリックし、「呼び出し先-呼び出し元のメトリック」ダイアログを表示します。

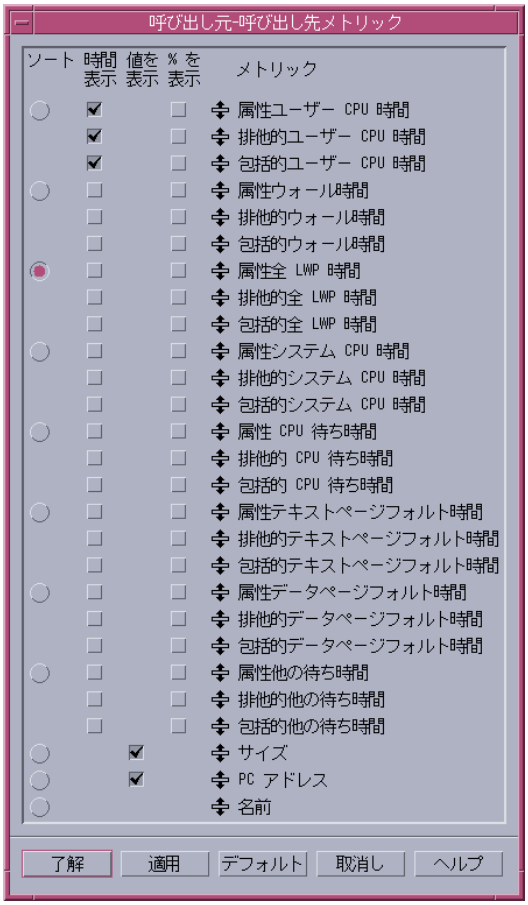


図 4-8　「呼び出し元-呼び出し先のメトリック」ダイアログ

このダイアログは、パフォーマンスアナライザに読み込まれた実験セットでサポートされているメトリックの一覧です。これらのメトリックを任意の組み合わせで表示することができます。また、次の情報を表示することもできます。

- サイズ (バイト単位)
- プログラムカウンタ (PC) のアドレス

関数名は常に表示されます。

「呼び出し元-呼び出し先」ウィンドウのデータ表示区画内のメトリック列の順序は、このダイアログの一覧内のメトリックの順序によって決まります。

- 表示するメトリックとその単位を選択するには、「時間表示」、「値を表示」、「%を表示」列の適切なチェックボックスをクリックします。
- ソート順序を指定するには、「ソート」列の適切なラジオボタンをクリックします。

  どのメトリックも、属性値に基づいてのみメトリックをソートすることができます。
- 一覧内のメトリックの順序を変更するには、メトリック名の左横のアイコンをドラッグ & ドロップします。移動したメトリックは、ドロップ先のメトリックの上に表示されます。
- 「呼び出し元-呼び出し先」ウィンドウに新しい設定内容を適用し、「メトリック」ダイアログを閉じる場合は「了解」を、「メトリック」ダイアログを閉じないでウィンドウに新しい設定内容を適用する場合は「適用」をクリックします。

注釈付きソースコードと逆アセンブリコードの表示

パフォーマンス上の問題の原因になっている関数を特定したら、パフォーマンスメトリックからなる注釈付きソースコードまたは逆アセンブリコードを表示し、その問題の原因になっている実際の行または命令を見つけることができます。

関数の注釈付きソースコードや逆アセンブリコードを表示するには、以下の操作を行います。

1. 関数リストから問題のある関数を含む行をクリックして選択します。

2. 「アナライザ」ウィンドウ下部のツールバーにある「ソース」または「逆アセンブル」をクリックします。

   テキストエディタのウィンドウが開き、選択した関数のコードが表示されます。各コード行のパフォーマンスメトリックが、その行の左側に表示されます。テキストエディタを選択するには、「オプション」➤「テキストエディタオプション」を選択します。

   注釈に表示されるメトリックは、ソースコードまたは逆アセンブリコードを表示したときに関数リストに表示されていたメトリックです。表示するメトリックを変更するには、「関数リストのメトリック」ダイアログで関数リストのメトリックを変更してから、注釈付きソースコードまたは逆アセンブリコードを再表示します。

   選択された関数のエントリポイントと、各列の最大メトリック値に対する割合で指定された値より大きいメトリック値を含む行が強調表示されます。

   プログラムのソースが存在する場合に、「逆アセンブル」をクリックして注釈付き逆アセンブリコードを生成すると、ソースコードと逆アセンブリコードが交互に表示されます。

   ソースコードの検索場所に関する情報は、プログラムが使用するロードオブジェクトに格納されています。実験が完了すると、他のものとともに、ロードオブジェクトごとに、そのオブジェクトへのパスと最終修正日時を示すタイムスタンプからなるアーカイブファイルが作成され、注釈付きソースコードまたは逆アセンブリコードの表示を要求すると、これらの情報がチェックされます。実験ファイルだけを別のコンピュータにコピーすると、注釈付きのソースコードや逆アセンブリコード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 (または同じパスとタイムスタンプを持つコピー) にアクセスできる必要があります。

   注釈付きソースコードの表示要求があると、パフォーマンスアナライザは、実行可能ファイルと実験ファイルに記録されている絶対パス下に、選択された関数を含むファイルが存在しているか調べ、存在しない場合は、現在の作業用ディレクトリに同じベース名を持つファイルがないかを調べます。ソースファイルを移動していたり、別のファイルシステム上で実験ファイルを記録した場合、注釈付きのソースコードを表示するには、現在のディレクトリから実際のソースの格納場所へのシンボリックリンクを作成します。

   ソースおよび逆アセンブリコードともに、コンパイラのコメントとそのコメントが関連付けられたソースコードが交互に表示されます。表示するコンパイラのコメントの種類は、`er_print` コマンドを含むデフォルト値ファイルで設定できます。コンパイ

ラのコメントの種類を選択するためのコマンドににについては、126 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」、デフォルト値ファイルについては、132 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。

注釈付きソースコードと逆アセンブリコードについては、第 6 章で詳細を説明しています。表6-2 に、注釈に表示されるメトリック値をまとめます。

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、実験ファイルの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、ルーチン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、以下の操作を行います。

1. 標本コレクタを使用してパフォーマンスデータを収集します (64 ページの「Sun WorkShop 統合プログラミング環境からのデータの収集」を参照)。

   マップファイル内のルーチンの順序は、関数リスト内の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ルーチン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。

2. 生成した実験ファイルをパフォーマンスアナライザに読み込みます (90 ページの「パフォーマンスアナライザの実行」を参照)。

3. 「メトリック」ダイアログを使用して、ルーチンの順序を決めるのに使用するソート基準メトリックを選択します。

4. 「実験ファイル」➤「マップファイル作成」を選択します。「マップファイル作成」ダイアログが表示されます。

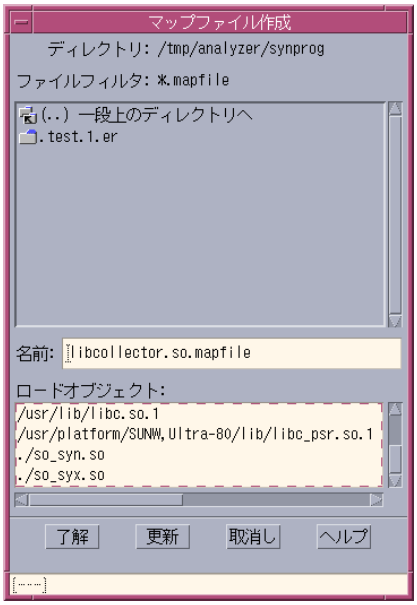


図 4-9　「マップファイル作成」ダイアログ

5. 「名前」テキストボックスに、作成するマップファイルを指定します。
6. 「ロードオブジェクトの選択」リストボックスで、マップファイルを作成するロードオブジェクト (通常はプログラムセグメント) を選択します。
7. 「了解」をクリックします。

マップファイルを使用してプログラム内のルーチンの順序を変更するには、以下の操作をします。

1. `-xF` オプションを使用してプログラムをコンパイルします。このオプションによって、コンパイラは、個別に再配置可能な関数を生成します。
2. `-M` オプションを使用してプログラムをリンクします。

C プログラムの場合は、以下のコマンドを使用します。

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

C++ プログラムの場合は、以下のコマンドを使用します。

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

Fortran プログラムの場合は、以下のコマンドを使用します。

```
% f95 -xF -c source-file-list
% f95 -M mapfile-name -o program-name object-file-list
```

以下の警告メッセージが表示された場合は、非共有オブジェクトやライブラリファイルなどの静的にリンクされたファイルすべてについて、-xF オプションを指定してコンパイルしたかどうかを調べます。

```
ld: warning: mapfile: text: .text% function-name: object-file-name:
Entrance criteria not met named-file, function-name, has not been
compiled with the -xF option.
```

標本情報の表示

概要表示から標本に関する情報を調べることができます。

- **標本の概要を表示するには、「データ」リストボックスから「概要」を選択します。**

  概要表示には、プログラムの一部または全実行期間中のプロセス時間に関する情報が表示されます。概要表示は、2 つの区画で構成されています。

  - 左の区画 - さまざまなプロセス状態で費やされた時間の割合を示すグラフが表示されます。これらの百分率値は、選択された 1 つまたは特定の範囲の標本にまたがってとられた平均値です。
  - 右の区画 - 選択されたすべての標本について、さまざまなプロセス状態で費やされた時間を示す一連のグラフが表示されます。各グラフは、1 つの標本収集間隔で標本コレクタが収集した標本情報を示します。標本の上にその ID 番号、標本の下に時間が表示されます。

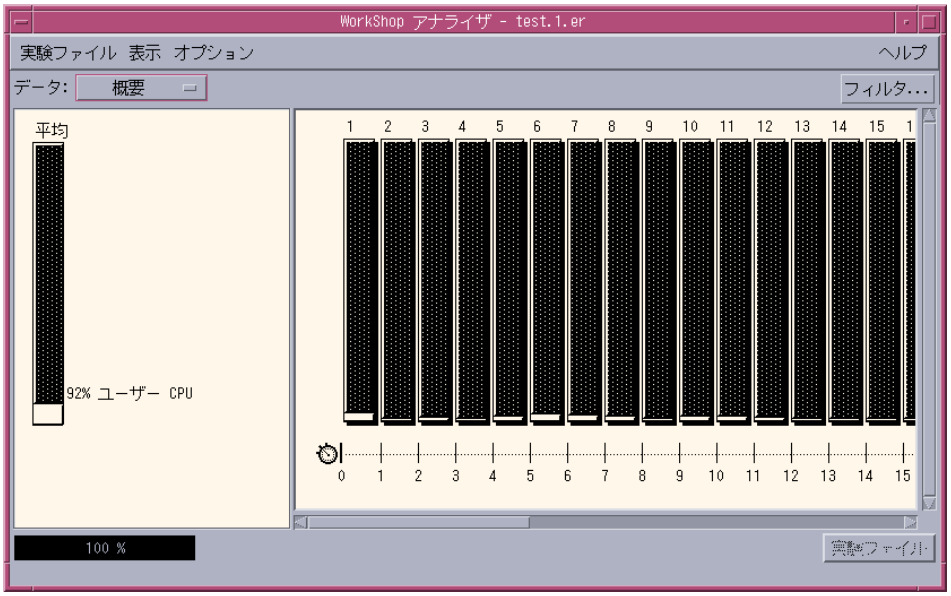


図 4-10 概要表示

デフォルトでは、概要表示の標本は固定幅で表示されます。つまり、標本のグラフは、標本収集間隔が同じ長さであるかどうかに関係なく、すべて同一幅で表示されます。「オプション」➤「概要のカラムサイズを指定」を使用し、標本収集間隔の長さに比例した幅で標本が表示されるように設定することもできます。

概要情報は、実験ごとに個別に表示されます。概要情報を表示する実験を選択するには、下部のツールバーにある「実験」ボタンを使用します。

プロセスの状態メトリックが小さすぎて標本のグラフで確認できないといった場合は、概要表示している標本に関するさらに詳細な解析情報を表示することができます。特定の標本の詳細情報を表示するには、その標本を選択する必要があります。このためには、「フィルタ」ボタンを使用します。表示するフィルタの選別方法については、95 ページの「実験、標本、スレッド、LWP の選択」を参照してください。デフォルトでは、すべての標本が選択状態になっています。

注 - 「フィルタ」ダイアログを使用して標本を選択した場合は、右の表示区画の標本の後ろに影が表示されます。

- 標本の詳細情報を表示するには、メニューバーから「表示」➤「標本の詳細を表示」を選択します。

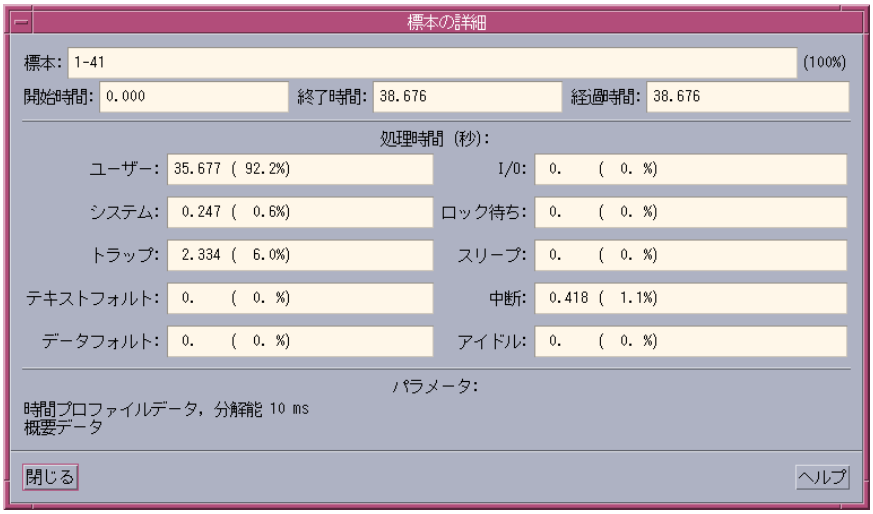


図 4-11 「標本の詳細」ウィンドウ

以下のメトリックを含む「標本の詳細」ウィンドウが表示されます。

- 標本の ID
- 選択された全標本が占める割合
- 標本収集の開始時刻と終了時刻および長さ (秒単位)
- プロセスの状態と各状態で費やされた時間 (秒数と選択された全標本の全体のメトリックに占める割合) のリスト
- 標本コレクタが実験ファイルに記録したデータの種類とデータが記録されたパラメータの一覧

実行統計情報の表示

実行統計表示は、選択した標本について集計されたさまざまなシステム統計情報の一覧です。標本の選択方法については、95 ページの「実験、標本、スレッド、LWP の選択」を参照してください。

- 実行統計情報を表示するには、「データ」リストボックスから「実行統計」を選択します。

注 - 「実行統計」ウィンドウのデータは、全体をクリップボードにコピーして、テキストエディタにペーストすることができます。

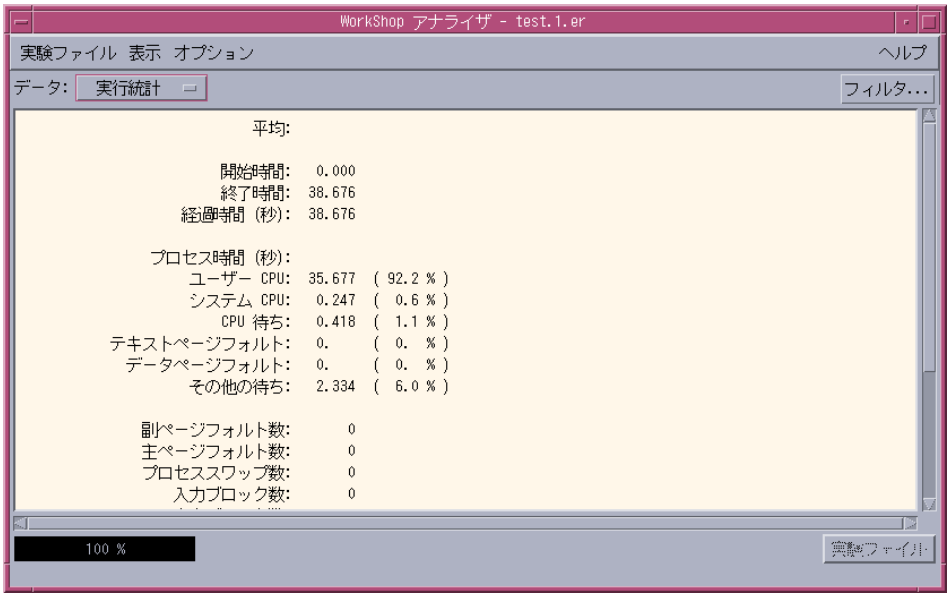


図 4-12 実行統計表示

アドレス空間情報の表示

アドレス空間情報は、標本コレクタで実験レコードを生成するときにアドレス空間データを選択した場合にのみ表示できます。選択していない場合は、パフォーマンスアナライザによって、アドレス空間情報がないという意味のメッセージが表示されます。この機能は、IA ハードウェアでは利用できません。

- **アドレス空間情報を表示するには、「データ」オプションリストから「アドレス空間」を選択します。**

  アドレス空間表示は、2 つの区画で構成されています。

  - 左の区画 - 右側のグラフ表示の説明が表示されます。
  - 右の区画 - プログラムのアドレス空間がグラフィカルに表示されます。

デフォルトでは、ページ単位で表示されます (「単位」のラジオボタンの「ページ」を選択した場合でも、この表示になります)。各マス目は、アドレス空間の 1 ページを示します。マス目の模様は、プログラムがページをどのように処理しているかを示します。

- 変更 (書き込み)
- 参照 (読み取り)
- 参照なし

図 4-13 アドレス空間表示

アドレス空間をセグメント表示するには、上部のツールバーにある「セグメント」ラジオボタンをクリックします。右の区画に、プログラムが使用していているメモリーブロックが処理の区別なしで表示されます。

ページまたはセグメントの詳細情報を表示するには、以下の操作を行います。

1. 「単位」のラジオボタンを使用し、ページ表示またはセグメント表示のいずれかを選択します。
2. 右の区画からページまたはセグメントをクリックして選択します。

   右の区画からページまたはセグメントを選択すると、ページまたはセグメントの後ろに影が表示されます。

3. 「アナライザ」ウィンドウのメニューバーから、「表示」➤「ページ属性を表示」または「表示」➤「セグメント属性を表示」を選択します。

   以下の情報からなる「ページ属性」ウィンドウまたは「セグメント属性」ウィンドウが表示されます。

   - ページまたはセグメントのアドレス
   - ページまたはセグメントのサイズ (バイト単位)
   - セグメント名 (判明している場合)
   - 関数のリスト (ページまたはセグメントに関数がある場合)

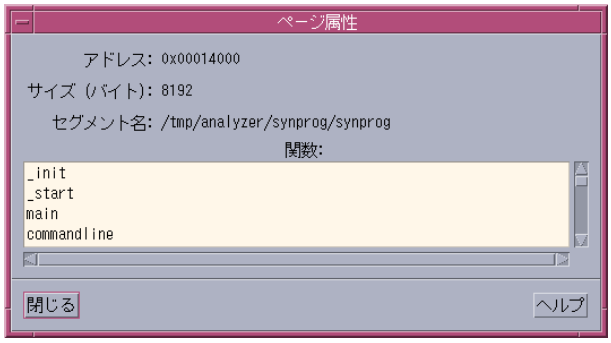


図 4-14 「ページ属性」ウィンドウ

表示内容の印刷

パフォーマンスアナライザの表示内容をテキスト形式で印刷するには、以下のようにします。

1. 「アナライザ」ウィンドウのメニューバーから「実験ファイル」➤「印刷」を選択し、「印刷」ダイアログを開きます。
2. 「出力先」のラジオボタンを使用し、プリンタまたはファイルのどちらに出力するかを選択します。

   - プリンタに印刷する場合は、「プリンタ」テキストボックスにプリンタ名を指定します。

- ファイルに出力する場合は、「ファイル」テキストボックスにファイル名を入力するか、「ブラウズ」ボタンをクリックして「印刷形式でファイルに保存」ダイアログを表示し、保存先のファイルを指定します。

3. 「印刷」ボタンをクリックします。

注 - 概要の場合は、グラフィック表示をテキスト形式に変換したものではなく、実験の各標本の統計情報のリストが出力されます。

第5章

er_prin コマンド行インタフェースによるプログラムのパフォーマンス解析

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、標本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
- 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

標本コレクタの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 4 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 | - ] experiment1 [ experiment2 … ]
```

表 5-1 に、er_print のオプションをまとめます。

表 5-1　er_print コマンドのオプション

オプション	説明
-	端末から入力された 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。

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。この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) とメトリック表示形式文字列 (visibility)、メトリック名文字列 (name) の 3 つの部分から構成されます。これらは、空白を入力せずに次のように続けて指定します。

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

メトリックタイプとメトリック表示形式文字列は、タイプ文字と表示形式文字を使用して指定します。

指定可能なメトリックタイプ文字を、表 5-2 にまとめます。複数のタイプ文字からなるメトリックキーワードは展開され、メトリックキーワードリストになります。たとえば、ie.user は、展開されて i.user:e.user になります。

表 5-2　メトリックタイプ文字

文字	説明
e	排他的メトリック値を表示します。
i	包括的メトリック値を表示します。
a	属性メトリック値を表示します (呼び出し元-呼び出し先メトリックのみ)

指定可能なメトリック表示形式文字を、表 5-3 にまとめます。表示形式文字列を構成する文字の順序は重要ではありません。対応するメトリックの表示順序が、この指定順序の影響を受けることはありません。たとえば、i%.user と i.%user は、ともに i.user:i%user と解釈されます。

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

表 5-3 メトリック表示形式文字

文字	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 5-4 に、クロックベースのメトリックと同期遅延メトリック、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、49 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 5-4　メトリック名文字列

文字列	説明
user	ユーザー CPU 時間
wall	ウォール時間
total	全 LWP 時間
system	システム CPU 時間
wait	CPU 待ち時間
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
sync	同期待ち時間
syncn	同期待ち回数
cycles	CPU サイクル数 (レジスタ 0 でカウント)
cycles1	CPU サイクル数 (レジスタ 1 でカウント)
insts	発行命令数 (レジスタ 0 でカウント)
insts1	発行命令数 (レジスタ 1 でカウント)

注 - wait メトリック文字列の意味が変わり、wait は CPU 待ち時間を意味します。新しい文字列の owait がその他のすべてのシステム待ち時間を意味します。

表 5-4 に示す名前文字列の他に、デフォルトのメトリックリストでのみ使用可能な 2 つの名前文字列 (任意のハードウェアカウンタ名を意味する hwc と、任意のメトリック名文字列を意味する any) があります。

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。

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` コマンド使用して制限できます (133 ページの「出力関連のコマンド」を参照)。

関数の概要メトリックについては、100 ページの「関数やロードオブジェクトの概要メトリックの表示」を参照してください。

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。出力される行数は、`limit` コマンドを使用して制限できます (133 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` コマンドを使用して変更できます。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。

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、119 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

現在選択されているメトリックでロードオブジェクトリストを出力します。出力する行数は、limit コマンドを使用して制限できます (133 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。

osummary

ロードオブジェクトリスト内のすべてのロードオブジェクトについて、それぞれに概要メトリックパネルを出力します。出力するパネル数は、limit コマンドを使用して制限できます (133 ページの「出力関連のコマンド」を参照)。

ロードオブジェクトの概要メトリックについては、100 ページの「関数やロードオブジェクトの概要メトリックの表示」を参照してください。

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。*metric-keyword* は、119 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 (133 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。

```
Excl. User  Incl. User  Attr. User  名前
CPU sec.    CPU sec.    CPU sec.
0.          0.010       0.010       _doprnt
0.          0.          0.          _xflsbuf
0.          0.010       0.         *_realbufend
0.          0.620       0.          _rw_rdlock
0.          0.010       0.010       rw_unlock
```

この例では、`_realbufend` が選択されている関数です。この関数は、`_doprnt` と `_xflsbuf` によって呼び出され、`_rw_rdlock` と `_rw_unlock` を呼び出します。

cmetrics *metric-list*

呼び出し元 - 呼び出し先の一群のメトリックを指定します。*metric-list* は、下記の例で示しているように、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、119 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。*metric-keyword* は、119 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。*class-list* は、コロンで区切ったクラスのリストで、以下のメッセージクラスを指定できます。また、強調表示しきい値を指定することもできます。

- `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。
- t[hreshold] = *nn* - ソースまたは逆アセンブリコード行の強調表示しきい値を設定します。ファイル内のソース行または逆アセンブリ命令に関するメトリック値が、その最大値の *nn*% 以上の場合に、該当する行が強調表示されます。

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。

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。*class-list* は、コロンで区切ったクラスのリストで、注釈付きソースコードのクラスのほかに以下のクラスを指定できます。

- h[ex] - 命令の 16 進値を表示します。

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、その直前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-selectionz*

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*

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。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、コンマを二重引用符で囲む必要があります。

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。

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

解析対象として選択されているロードオブジェクトを一覧表示します。

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
=== ================================================
はい /tmp/var/synprog/synprog
はい /opt/SUNWspro/WS6U2/lib/dbxruntime/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に優先し、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に優先します。

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 および dcc 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、119 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。メトリックリストの構文と使用方法については、119 ページの「メトリックリスト」で説明しています。

同様に、実験が読み込まれている場合、呼び出し元 - 呼び出し先リストのデフォルトのソート基準メトリックは、`dsort` メトリックリスト内の、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対応する属性メトリックになります。

`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

`address_space` *experiment-ID*

指定した実験のアドレス空間データを表示します。*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* を省略した場合は、最初に読み込まれた実験のデータが表示されます。

`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* を省略した場合は、最初に読み込まれた実験の情報が表示されます。

`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の概要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* を省略した場合は、最初に読み込まれた実験のデータが表示されます。

`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。*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* を省略した場合は、最初に読み込まれた実験のデータが表示されます。

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。

第6章

パフォーマンスアナライザとそのデータの内容

パフォーマンスアナライザは、標本コレクタの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ピング
- 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、高精度のタイムスタンプ、スレッド ID、LWP ID が含まれます。パフォーマンスアナライザでは、これら 3 つの情報 (時刻、スレッド、LWP) に基づいてメトリックを選別表示することができます。また、各イベントでは、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状態の、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状態のカウント値は 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 6-1 に、各マイクロステート状態とメトリックの対応関係をまとめます。

表 6-1 カーネルのマイクロステート状態とメトリックの対応関係

カーネルのマイクロステート状態	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	その他の待ち時間
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の間、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状態に帰せられます。最初のプロファイルパケットが記録されたとき LWP がシステム CPU マイクロステート状態である可能性があり、その場合、システム CPU 時間メトリックは実際より大きくなります。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。この時間は、しばしばユーザー CPU マイクロステート状態で費やされるため、ユーザー CPU 時間のメトリックが実際より小さくなります。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 のスケジューリングは、プロファイル間隔より短い時間の尺度で行われます。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状態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 マイクロステート状態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。

- 命令の実行間隔は、プロファイル間隔よりかなり短い時間です。このため、命令に時間を正しく関連付けられないことがあります。

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% の精度になります。CPU 時間の場合は、`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、ルーチン、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「実行統計」ウィンドウに表示されるマイクロステートタイミングは、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、マイクロステート状態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、集計されたタイミングメトリックのおおよその精度を知ることができます。

同期待ちの監視

標本コレクタは、スレッドライブラリ (`libthread.so`) 内の関数の呼び出しまたは MPI ブロック化ルーチンの呼び出しを監視することによって、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 (監視対象の呼び出しの始まりと終わり) の高精度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。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。同期待ち監視データは、許可時に実験ファイルに記録されます。

プログラムが結合スレッドを使用している場合は、その遅延の原因となったイベントが完了しない限り、待ちスレッドがスケジューリングされている LWP が他の作業を行うことはできません。この待ち時間は、同期待ち時間とその他の待ち時間の両方に反映されます。

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、その他の待ち時間はゼロになりますが、同期待ち時間はゼロにはなりません。これは、その時間が、スレッドが動作している LWP ではなく、特定のスレッドに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。

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、164 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。

- 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。
- 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_record_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。

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)。

パフォーマンスデータの排他的メトリックは、呼び出しスタック内のリーフ PC が含まれている関数が原因とされます。また、包括的メトリックの場合は、スタック上の各 PC (リーフ PC を含む) が含まれている関数が原因とされます。

ほとんどの場合、記録された呼び出しスタック内のすべての 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、136 ページの「プログラム構造へのアドレスのマッピング」を参照してください。

シングルスレッド実行と関数の呼び出し

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禁止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。当初、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当てます。PLT アドレスは呼び出しスタックに現れることがあります (159 ページの「<未知> 関数」を参照)。

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、システムルーチン `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ルーチンの `sigacthandler()` の呼び出し元、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、割り込まれたルーチンの包括的メトリックとして表示されます。

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としては、たとえば、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状態はユーザー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。マイクロステート状態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ルーチンとしては、たとえば `gethrtime` や `gethrvtime` があります。これらのルーチンではオーバーヘッドを伴うため、マイクロステート状態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、TLB (translation lookaside buffer) ミスやレジスタウィンドウのスピルおよびフィルなどです。

どちらの場合も、消費時間は ユーザー CPU 時間として記録されますが、システムモードに切り替えられているため、ハードウェアカウンタは無効になります。このため、これらのトラップの処理に費やされた時間は、時間ベースの実験のユーザー CPU 時間と、ハードウェアカウンタ実験の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

特定のルーチンがその最後で他のルーチンを呼び出す場合、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と、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、ルーチン A が ルーチン 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) 上の単一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ルーチンを呼び出し、そのルーチンに、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()` というルーチン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、スレッド作成ルーチン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、それら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。

オペレーティングシステムは、実行時に CPU への LWP の割り当てを制御し、スレッドライブラリは LWP に対するスレッドのスケジューリングを制御します。たとえば、スレッドが mutex_lock などによって同期が阻まれている場合、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、その時間がその他の待ち時間に加算されることはありません。

並列実行とコンパイラ生成の本体関数

コードに Sun、Cray、OpenMP のいずれかの並列化指令が含まれている場合は、並列実行用のコンパイルを行うことができます (OpenMP は Sun WorkShop 6 update 2 の Fortran 95 および C コンパイラで使用可能な機能です) 。並列化戦略と OpenMP 指令の詳細については、『Fortran プログラミングガイド』と『C ユーザーズガイド』の並列化と OpenMP に関する章を参照するか、OpenMP 規格に関する 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

並列構造を検出した場合、コンパイラは、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ルーチン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ルーチンは、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ルーチンに渡されます。

- 並列構造が以下のいずれかで区切られている場合、並列構造はマイクロタスクライブラリのルーチンである __mt_MasterFunction_() への呼び出しに置き換えられます。
  - 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 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。

- `c$omp PARALLEL` 構造に、ワークシェアリング用の `c$omp DO` または `c$omp SECTIONS` 指令が含まれている場合、それぞれのワークシェアリング構造は、マイクロタスクライブラリルーチンの `__mt_Worksharing_()` への呼び出しに置き換えられ、それぞれに新しい本体関数が 1 つ作成されます。

このときコンパイラは、以下の形式の名前を本体関数に割り当てます。

```
_$1$mf_string1_$namelength$functionname$linenumber$string2
```

- *string1* は並列構造の種類を示します。
- *namelength* は、*functionname* の文字数を示します。
- *functionname* は、構造の抽出された関数の名前で、Fortran 関数の場合、最後の文字は下線 (_) になります。
- *linenumber* は、元のソースでの構造の先頭行の行番号です。
- *string2* は、ソースファイル名に関係する文字列です。

データを解析しやすくするために、パフォーマンスアナライザは、コンパイラ生成名に加えて、本体関数に分かりやすい名前を割り当てます。

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ルーチンの `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ルーチンの `__mt_SlaveFunction_()` を実行します。

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_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 の場合は、__mt_run_my_job_() が別の複数のルーチン (ループの性質によって異なる) を呼び出し、それらのルーチン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。このルーチンには、__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、__mt_run_my_job_() は、worksharing section の場合は直接、また worksharing do または for の場合は、ループ依存ルーチン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。

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

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

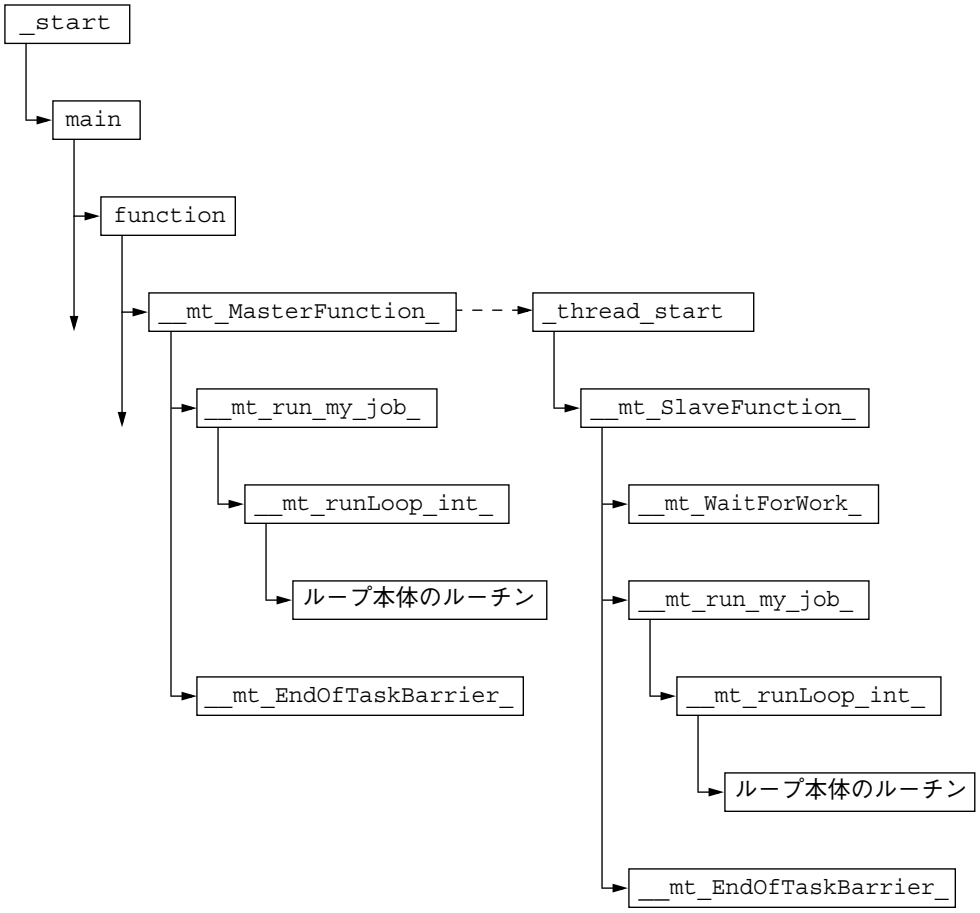


図 6-1　Parallel Do 構造を含むマルチスレッドプログラムの呼び出しツリー

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

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

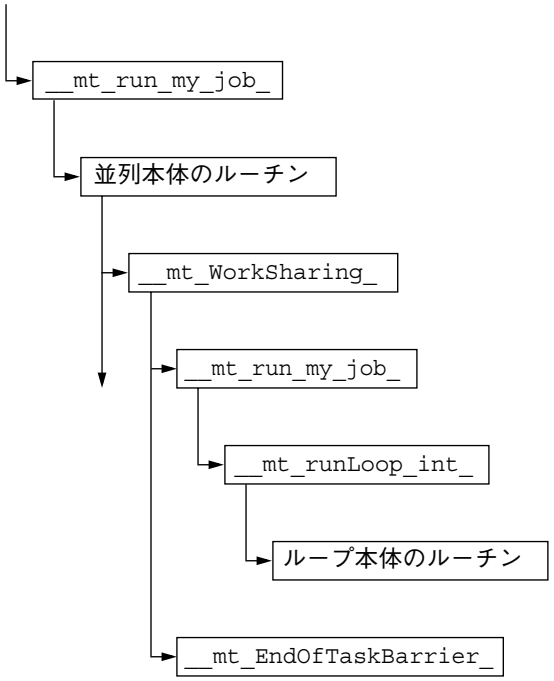


図 6-2　Worksharing Do 構造を含む並列領域の呼び出しツリー

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ルーチン (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、本体関数と本体関数を呼び出すマイクロタスクライブラリ関数の間に元の関数への呼び出しを挿入し、バリア関数 (`__mt_EndOfTaskBarrier_()`) と `__mt_MasterFunction_()`、`__mt_SlaveFunction_()`、`__mt_WorkSharing_()` のいずれかの間に、元の関数および本体関数への呼び出しを挿入します。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、本体ルーチン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ルーチンではなく、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、元の関数がマイクロタスクルーチンの呼び出し先として現れます。こうした呼び出しは無視してください。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 (90 ページの「関数レベルのメトリックに再帰が及ぼす影響」を参照)。

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。これが、ビジー待機と呼ばれる状態です。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。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。

プログラム構造へのアドレスのマッピング

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) にマッピングします。ここでは、これらのマッピングについて説明します。

プロセスイメージ

プログラムを実行すると、そのプログラムの実行可能ファイルからプロセスがインスタンス化されます。プロセスのアドレス空間には、実行可能な命令を表すテキストが存在する領域や、通常は実行されないデータが存在する領域などの多数の領域があります。通常、呼び出しスタックに記録される PC は、プログラムのいずれかのテキストセグメント内のアドレスに対応しています。

プロセスの先頭テキストセクションは、実行可能ファイルそのものから生成されます。先頭以外のテキストセクションは、プロセスの開始時に実行可能ファイルとともに読み込まれたか、プロセスによって動的に読み込まれた、共有オブジェクトに対応しています。呼び出しスタック内の PC 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、集合的に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、必ず、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++ 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、関数にマッピング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、まったく同じマッピング情報が使用されます。関数の多くは、プログラムのソースモデルに直接対応します。以降の節では、一部の、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。「概要メトリック」ウィンドウで、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。「概要メトリック」ウィンドウでは、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。「概要メトリック」ウィンドウでは、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。
- プログラムで、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。標本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、その場合も、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ルーチン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、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。

インライン化された関数

インライン化された関数はソースで関数と定義されているコードであり、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、148 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、149 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。ページングおよび命令キャッシュの動作を改善するために、こうしたコードはアドレス空間の別の場所に移動され、以下の形式の名前を持つ関数になります。

```
_$1$outline string1$namelength$functionname$linenumber$string2
```

- *string1* は、アウトライン関数の特定のセクションに関係する文字列です。
- *namelength* は、*functionname* の文字数を示します。
- *functionname* は、構造が抽出された関数の名前です。

- *linenumber* は、元のソース内のセクションの先頭行の行番号です。
- *string2* は、コンパイラの内部名に関係する文字列です。

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、メインルーチンからそのアウトライン部分への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。

<未知> 関数

PC が既知の関数にマッピング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ピングされます。

PC が <未知> にマッピングされるのは、次のような場合です。

- PC がロードオブジェクト内のプログラムリンケージテーブル (PLT) に対応している。こうした状況は、ロードオブジェクト内の関数が、別の共有オブジェクト内の関数を呼び出すときに必ず発生します。実際の呼び出しは、最初に PLT 内 の 3 命令シーケンスに移り、その後、実際の呼び出し先に移ります。
- 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。通常、データを実行することはできないため、データアドレスが呼び出しスタックに現れることはありません。自己を書き換えるプログラム、またはルーチンが動的にコンパイルされるプログラムは、こうしたコードを実行する前にプログラムデータ空間に命令を書き込みます。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。
- 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。

<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。

注釈付きコードリスト

パフォーマンスアナライザの注釈付きソースコードおよび逆アセンブリコード機能は、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 をソース行にマッピングすることによって作成されます。注釈付きソースファイルを作成するにあたり、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ピング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。

コンパイル処理では、要求される最適化レベルに応じて多くの段階があり、変換によって命令とソース行のマッピングに混乱が生じることがあります。最適化によっては、ソース行の情報が完全に失われたり、混乱が生じたりすることがあります。コンパイラは、さまざまな発見手法によって命令のソース行を追跡しますが、こうした手法は絶対ではありません。

表 6-2 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 6-2　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。
`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、監視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、行が実行されなかったことを意味するわけではなく、単にそのメトリックがプロファイルに統計データとして現れず、その行のスレッド同期呼び出しで、しきい値を超える遅延がなかったことを意味します。
`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ピング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、マッピング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

一般的な最適化では、同じ式が複数の場所に出現することが検出され、その式のコードを一カ所に集めることでパフォーマンスの向上が図られます。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。

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、行番号のマッピング情報およびソースファイルが存在するかどうか、また注釈付き逆アセンブリが要求されている関数のオブジェクトモジュールが既知かどうかによって決まります。

- オブジェクトモジュールが既知ではない場合は、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、ソースと逆アセンブリコードが交互に表示されます。
- コンパイラによってオブジェクトコードにコメントが挿入されている場合は、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。命令の行にはメトリックが表示されますが、挿入されたソースまたはコメントには表示されません。表示可能なメトリックは、表 6-2 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、行番号は単純で、ソースおよび逆アセンブルされた命令の交互表示は自然なものになります。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* にあったもの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。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

命令はグループ単位で読み込まれて、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)。

ハードウェアカウンタオーバーフローの関連付け

オーバーフローで生成されたシグナル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。しばしば、記録された PC の少し前の命令に本当の PC があることがあり、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、本当の 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。

第7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、標本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、標本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。移動先に *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-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、注釈付きの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。アナライザの表示については、106 ページの「注釈付きソースコードと逆アセンブリコードの表示」を参照してください。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、126 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、132 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。

-h

逆アセンブリコードの命令を 16 進数形式で表示します。デフォルトでは、16 進数形式で表示されません。逆アセンブリコードが要求されていない場合、このオプションは無視されます。

-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。*filename* がハイフン (-) の場合は、stdout に出力されます。デフォルトでは、リストはデフォルトのテキストエディタに表示されます。

-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アナライザや er_print コマンドを初めて使用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 が実行されることはありません。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 を手動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、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、アナライザまたは er_print で表示できます。

-F

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

-V

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。prof と gprof は、Solaris 2.6、7、8 (SPARC および Intel プラットフォーム版) に付属しているプロファイルツールです。tcov は、Forte™ Developer 製品に付属しているコードカバレージツールです。

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、標本コレクタおよびパフォーマンスアナライザを使用すると、プログラムが時間を消費する部分に関するより詳細で正確な情報を得ることができます。これらのツールの使用方法については、第 3 章と第 4 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、各ルーチン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 - このルーチンによって費やされる時間の、総 CPU 時間に対する割合
- Seconds - このルーチンが占める総 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 - このルーチンが呼び出される回数
- msecs/call - このルーチンが呼び出されたときに費やされる平均時間 (ミリ秒単位)
- Name - ルーチン名

   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() ルーチンによって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。15 ページの「gprof の誤った推論」を参照してください。

prof 同様、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、個々の呼び出し元 - 呼び出し先の間で制御が受け渡される回数もカウントします。

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 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、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. gprof を実行してプロファイルレポートを作成します。

   gprof 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 stdout 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ルーチン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。この意味で "0.07% of 14.74 seconds" は、14.74 ミリ秒の CPU 時間を表す 1 つの標本は実行全体の 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、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 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。tcov は、基本的なブロックに関する概要情報も提供します。時間ベースのデータは生成しません。

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、index.assist.tcov の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。

```
            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、共有ライブラリに tcov プロファイルサブルーチン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 サブルーチンが使用されます。

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、tmp/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、/tmp/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サブルーチンを呼び出します。そして、プログラムの終了時、これらのサブルーチンは、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。この場合は、数秒後にエラーメッセージが表示され、既存の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。

プロファイルサブルーチンは、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 実行時ルーチンによって報告されるエラー

ここでは、tcov 実行時ルーチン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: Could not write coverage data file 'coverage-data-file-name'
because 'system-error-message-string'.
```

- 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。また、このディレクトリは、プロファイルバケツともいいます。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 (187 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の、-xprofile=collect および -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

O

P

S

T

あ

い

う

え

お

か

き

こ

さ

し

す

せ

そ

た

ち

て

と

に

は

ひ

ふ

へ

ほ

ま

み

め

よ

ら

ろ