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. 806-4835-01
2000 年 6 月 Revision A

原典： *Analyzing Program Performance With Sun WorkShop*
Part No: 806-3562-10
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™ 統合プログラミング環境の基本的なプログラム開発機能について説明します。このマニュアルは C、C++、Fortran による開発経験を持つプログラマーで、Solaris™ および UNIX® に関する実用的な知識を持ち、Sun WorkShop の主な開発機能について理解することを目的としたアプリケーション開発者を対象にしています。

マルチプラットフォーム対応

この Sun WorkShop リリースは、Solaris 2.6、7、および 8 のオペレーティング環境 (SPARC™ プラットフォームおよび Intel プラットフォーム) をサポートしています。

注 – Intel アーキテクチャとは、Pentium、Pentium Pro、Pentium II プロセッサおよび、これらと互換性のある AMD および Cyrix 製のマイクロプロセッサチップを含む、Intel 8086 マイクロプロセッサチップ群を意味しています。このマニュアルでは、これらすべてのプラットフォームアーキテクチャを総称して Intel アーキテクチャと呼んでいます。

Sun WorkShop 開発ツールへのアクセス方法

Sun WorkShop 製品コンポーネントとマニュアルページは標準ディレクトリ /usr/bin および /usr/share/man にはインストールされません。そのため PATH および MANPATH 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ります。

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以下を実行します。

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、この節の指示に従って、PATH 環境変数を設定してください。

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以下を実行します。

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、この節の指示に従って MANPATH 環境変数を設定してください。

注 – この節に記載されている情報は Sun WorkShop 6 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 ソフトウェアが /opt ディレクトリにインストールされていない場合は、システム管理者に連絡してください。

PATH 変数および MANPATH 変数は、C シェルを使用している場合はホームディレクトリの下の .cshrc ファイルに設定する必要があります。Bourne シェルか Korn シェルを使用している場合は、ホームディレクトリの下の .profile ファイルに設定する必要があります。

- Sun WorkShop コマンドを使用するには、 PATH 変数に以下を追加してください。

  /opt/SUNWspro/bin

- man コマンドで、Sun WorkShop マニュアルページにアクセスするには、MANPATH 変数に以下を追加してください。

  /opt/SUNWspro/man

  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 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。

内容の紹介

このマニュアルは次の章と付録から構成されています。

第 1 章「パフォーマンスプロファイリングおよび解析ツールの概要」では、パフォーマンスプロファイリングツールを紹介し、その機能と使用方法を簡単に説明します。

第 2 章「チュートリアル：標本コレクタおよび標本アナライザの使用方法」では、Sampling Collector と Sampling Analyzer を使用して 3 つのサンプルプログラムのパフォーマンスを最大にする方法を、チュートリアル形式で説明します。

第 3 章「標本コレクタリファレンス」では、Sampling Collector を使用してプログラムの実行についての情報を収集する方法について説明します。

第 4 章「標本アナライザリファレンス」では、Sampling Analyzer を使用してプログラムのパフォーマンスを最大にする方法について説明します。

第 5 章「er_print リファレンス」では、er_print ユーティリティの使用方法について説明します。

第 6 章「上級項目: 標本アナライザとデータ」では、Sampling Analyzer によって表示される、データの最適化の影響に関する高度な問題について説明します。

第 7 章「ループ解析ツール」では、コンパイラによって並列化されたプログラムループの解析を行う、LoopReport と LoopTool について説明します。

付録 A「従来のプロファイルツール」では、従来のプロファイリングツールである `prof`、`gprof`、`tcov` について説明します。これらのツールで、プログラム中の使用度が高い部分を特定し、プログラムのテスト回数を決定することができます。

書体と記号について

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ます。

表 P-1　このマニュアルで使用している書体と記号

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』

表 P-1　このマニュアルで使用している書体と記号 (続き)

書体または記号	意味	例
「 」	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」

シェルプロンプトについて

シェルプロンプトの例を以下に示します。

表 P-2　シェルプロンプト

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)	`#`

関連マニュアル

以下の方法で、関連マニュアルにアクセスすることができます。

- インターネットの docs.sun.com の Web サイトからアクセスできます。特定の本のタイトルで検索するか、主題、マニュアルコレクションまたは製品別にブラウズすることができます。

  http://docs.sun.com

- ローカルシステムまたはローカルネットワークにインストールされた Sun WorkShop 製品からアクセスできます。Sun WorkShop 6 HTML 文書 (マニュアル、オンラインヘルプ、マニュアルページ、各コンポーネントの README ファイル、リリースノート) が、インストールした Sun WorkShop 6 製品から参照可能です。HTML 文書にアクセスするには、次のいずれかを実行します。

  - Sun WorkShop または Sun WorkShop™ TeamWare ウィンドウで、「ヘルプ」➤「オンラインマニュアルについて」を選択します。
  - Netscape™ Communicator 4.0 またはその互換バージョンのブラウザで、以下のファイルを開きます。

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

  参照できる Sun WorkShop 6 HTML 文書の一覧がブラウザに表示されます。一覧にあるマニュアルを開くには、マニュアルのタイトルをクリックしてください。

表 P-3 は、Sun WorkShop 6 関連マニュアルをマニュアルコレクション別に一覧にしたものです。

表 P-3 マニュアルコレクション別 Sun WorkShop 6 関連マニュアル

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
Forte Developer 6 / Sun WorkShop 6 リリースマニュアル	Sun WorkShop 6 マニュアルの概要	Sun WorkShop 6 で使用可能なマニュアルとそのアクセス方法について説明しています。
	Sun WorkShop の新機能	Sun WorkShop 6 の現在のリリースと以前のリリースでの新機能についての情報を記載しています。
	Sun WorkShop 6 リリースノート	インストールの詳細と Sun WorkShop 6 最終リリースの直前に判明した情報を記載しています。このマニュアルはコンポーネントごとの README ファイルにある情報を補足するものです。

表 P-3　マニュアルコレクション別 Sun WorkShop 6 関連マニュアル (続き)

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
Forte Developer 6 / Sun WorkShop 6	プログラムのパフォーマンス解析	新しい標本コレクタと標本アナラザの使い方について説明しています (上級者向けのプロファイリング事例と説明付き)。コマンド行解析ツール `er_print`、ループツール、ループレポートユーティリティおよび UNIX プロファイルツール `prof`、`gprof`、`tcov` についての情報も含んでいます。
	dbx コマンドによるデバッグ	dbx コマンドを使ってプログラムをデバッグする方法について説明しています。参考情報として、同じデバッグ処理を Sun WorkShop デバッグウィンドウを使って実行する方法も記載しています。
	Sun WorkShop の概要	Sun WorkShop 統合プログラミング環境の基本的なプログラム開発機能について説明しています。
Forte C 6 / Sun WorkShop 6 Compilers C	C ユーザーズガイド	C コンパイラオプション、サン固有の機能 (プラグマ、`lint` ツール、並列化、64 ビットオペレーティングシステムへの移行および ANSI/ISO 準拠 C) について説明しています。
Forte C++ 6 / Sun WorkShop 6 Compilers C++	C++ ライブラリ・リファレンス	C++ ライブラリについて説明しています。C++ 標準ライブラリ、Tools.h++ クラスライブラリ、Sun WorkShop Memory Monitor、`Iostream` および複素数の情報も含みます。
	C++ 移行ガイド	コードを本バージョンの Sun WorkShop C++ コンパイラに移行する方法について説明しています。

表 P-3　マニュアルコレクション別 Sun WorkShop 6 関連マニュアル (続き)

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
	C++ プログラミングガイド	新しい機能を使ってより効率的なプログラムを記述する方法について説明しています。テンプレート、例外処理、実行時の型識別、キャスト演算、パフォーマンス、およびマルチスレッド対応のプログラムに関する情報も記載されています。
	C++ ユーザーズガイド	コマンド行オプションとコンパイラの使い方についての情報を記載しています。
	Sun WorkShop Memory Monitor ユーザーズガイド	C および C++ のメモリー管理で生じた問題を Sun WorkShop Memory Monitor で解決する方法について説明しています。このマニュアルはインストールした製品 (`/opt/SUNWspro/docs/ja/index.html`) からのみ参照可能で、`docs.sun.com` Web サイトで参照することはできません。
Forte for High Performance Computing 6 / Sun WorkShop 6 Compilers Fortran 77/95	Fortran ライブラリ・リファレンス	Fortran コンパイラによって提供されるライブラリルーチンの詳細について説明しています。
	Fortran プログラミングガイド	入出力、ライブラリ、プログラム分析、デバッグおよびパフォーマンスに関連する内容を記述しています。
	Fortran ユーザーズガイド	コマンド行オプションとコンパイラの使い方についての情報を記載しています。
	FORTRAN 77 言語リファレンス	Fortran 77 言語の包括的な参照情報を記載しています。
	Fortran 95 区間演算プログラミングリファレンス	Fortran 95 コンパイラによってサポートされる組み込み `INTERVAL` データについて説明しています。
Forte TeamWare 6 / Sun WorkShop TeamWare 6	Sun WorkShop TeamWare ユーザーズガイド	Sun WorkShop TeamWare コード管理ツールの使用方法について説明しています。

表 P-3　マニュアルコレクション別 Sun WorkShop 6 関連マニュアル (続き)

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
Forte Developer 6/ Sun WorkShop Visual 6	Sun WorkShop Visual ユーザーズガイド	C++ と Java™ の GUI (グラフィカルユーザーインタフェース) を Sun WorkShop Visual を使用して作成する方法について説明しています。このマニュアルには、旧リリース (Sun WorkShop Visual 5.0) から変更のない機能が記載されています。
	Sun WorkShop Visual の新機能	Sun WorkShop Visual 6.0 で追加または変更された機能について説明しています。
Forte / Sun Performance Library 6	Sun Performance Library Reference (英語のみ)	コンピュータによる線形代数および高速フーリエ変換を実行するサブルーチンと関数の最適化ライブラリについて説明しています。
	Sun Performance Library User's Guide (英語のみ)	線形代数で発生した問題の解決に使用されるサブルーチンと関数のコレクションである Sun Performance Library のサン固有の機能の使用方法について説明しています。
数値計算ガイド	数値計算ガイド	浮動小数点演算における数値の精度に関する問題について説明しています。
標準ライブラリ 2	Standard C++ Library Class Reference (英語のみ)	標準 C++ の詳細について説明しています。
	標準 C++ ライブラリ・ユーザーズガイド	標準 C++ ライブラリの使用方法について説明しています。
Tools.h++ 7	Tools.h++ 7.0 ユーザーズガイド	Tools.h++ クラスライブラリの詳細について説明しています。
	Tools.h++ 7.0 クラスライブラリ・リファレンスマニュアル	C++ クラスを使用して、プログラム効率を向上させる方法について説明しています。

表 P-4 は、docs.sun.com の Web サイトからアクセスできる Solaris 関連マニュアルの一覧です。

表 P-4 Solaris 関連マニュアル

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 ソフトウェア開発	リンカーとライブラリ	Solaris リンクエディタと実行時リンカーの操作およびそれらが操作するオブジェクトについて説明しています。
	プログラミングユーティリティ	Solaris オペレーティング環境で使用可能な特殊組み込みプログラミングツールに関する開発者向けの情報を記載しています。

第1章

パフォーマンスプロファイリングおよび解析ツールの概要

高性能なアプリケーションの開発には、コンパイラのさまざまな機能、最適化されたルーチンのライブラリ、およびコードの解析と特定を行うためのツールを組み合わせて使用することが必要です。このマニュアルでは、コードの解析と特定に役立つツールについて説明します。

このマニュアルでは、主に標本コレクタおよび標本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し、分析するために組で使用するツールです。

- 標本コレクタは、パフォーマンスデータ (呼び出しスタックの統計プロファイル、スレッド同期遅延イベント、ハードウェアカウンタオーバフロープロファイル、アドレス空間データ、オペレーティングシステムの要約情報) を収集し、実験ファイルに格納します。標本コレクタの詳細については、第 3 章を参照してください。
- 標本アナライザでは、標本コレクタが記録したデータが表示され、ユーザーがその情報を調べることができます。標本アナライザは、データを処理し、パフォーマンスに関するさまざまなメトリックを、関数レベル、呼び出し側と呼び出された側のレベル、ソース行レベル、逆アセンブリ命令レベル、およびプログラムレベルで表示します。標本アナライザの詳細については、第 4 章を参照してください。

  標本アナライザは、マップファイルを作成することによって、使用しているアプリケーションのパフォーマンスを正しく調整するのにも役立ちます。マップファイルを使用すると、アプリケーションのアドレス空間に関数を読み込む順序を改善することができます。

パフォーマンスの調整は開発者の主な仕事ではないにせよ、標本コレクタと標本アナライザは、ソフトウェア開発者が使用することを想定して設計されています。

標本コレクタおよび標本アナライザと同等の機能は、コマンド行で利用できます。

- `dbx` は、標本コレクタと同様のデータ収集機能を持っています。42 ページの「dbx で標本コレクタ下のプロセスを起動」 を参照してください。
- コマンド行ユーティリティ `er_print` は、さまざまな標本アナライザ表示のを ASCII 形式で出力し、コマンド行のサンプリングアナライザとして機能します。詳細については、 第 5 章を参照してください。

標本コレクタおよび標本アナライザは、次に示す Sun Workshop™ 製品に含まれています。

- Sun WorkShop Professional™ C
- Sun Visual WorkShop™ C++
- Sun Performance WorkShop™ Fortran
- Sun WorkShop™ University Edition

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。

- ループツールおよびループレポート

  ループツールは、自動的に並行化されたプログラムのパフォーマンスの調整をサポートするループ解析ツールです。ループレポート は、ループツールのコマンド行バージョンです。詳細については、第 7 章 を参照してください。

- `prof`、`gprof` および `tcov`

  `prof` および `gprof` は、プロファイルデータを生成するための従来からのツールであり、SPARC™ プラットフォーム版および Intel プラットフォーム版の Solaris™ バージョン 2.6、7 および 8 に含まれています。

  `tcov` は、コードカバレッジツールで、Sun Workshop に含まれています。

  `prof`、`gprof` および `tcov` の詳細については、付録 A を参照してください。

第2章

チュートリアル：標本コレクタおよび標本アナライザの使用方法

この章では、次に示す 3 つのサンプルプログラムのパフォーマンスを通じて、標本コレクタおよび標本アナライザを使用する方法を示します。

- 例 1：synprog - さまざまなプログラミング構造と、`gprof` の誤った推論の例を示すサンプルプログラム
- 例 2：omptest - OpenMP の並行化機能を使用する Fortran プログラム
- 例 3：mttest - 明示的なマルチスレッド機能を使用するプログラム

これらの例で、各プログラムは、次に示す同じパフォーマンスの問題を検討します。

- パフォーマンスを改善するには、プログラムのどこを修正したらよいか

  このレベルのプログラム解析は、より高いレベルのアルゴリズムの問題に関係しています。コンパイルされたコードはコンパイラができる範囲ですでに最適化されています。ここでは、プログラムをより効率的に実行するには、プログラムのアルゴリズム自体をどのように見直したらよいかを検討してみます。

- 自分のプログラムはどのような資源を使用しているか

  たとえば、自分のプログラムはどのくらいの CPU を使用しているか。

- これらの資源は、自分のプログラム内のどの箇所でもっとも使用されているか

  たとえば、プログラムの大部分の時間を費やしている特定の関数はあるか。また、その関数のどの行や命令が、時間の消費に大きく寄与しているか。

- 実行プロセスにおいて、問題となっている行や命令にプログラムがどのようにたどり着いているか。

資源の大部分を費やしているプログラム内の箇所がわかったら、Analyzer を使用すると、さまざまな方法でそのコードを調べることができるので、その理由を判断することができます。

注 – これらのすべての例で示される時間は、特定の実験で収集されたデータに基づくものです。異なる環境で行われた実験で収集されたデータからは、異なる数値が得られます。

これらの各デモプログラムのソースコードは、配布媒体に含まれています。このチュートリアルを開始する前に、次の作業が必要です。

1. **使用するパスに** Sun WorkShop **ディレクトリが追加されていることを確認してください。**

```
/opt/SUNWspro/bin
```

2. **いくつかのデモディレクトリに含まれているファイルを自分の作業領域にコピーし、**make **を行なって、デモプログラムを作成します。**

例 1：synprog

synprog プログラムには、興味深いパフォーマンス特性を示す、いくつかのプログラミング構造 (単純なメトリック解析、再帰、動的にリンクされた共用オブジェクトの読み込みと読み込み解除など) が組み込まれています。synprog プログラムは、標本コレクタと標本アナライザの操作演習に適しています。

synprog は、いわゆる「gprof の誤った推論」を引き起こします。標準 UNIX パフォーマンスツールである gprof コマンドは、synprog プログラムの CPU 時間のほとんどを自身で消費する関数 (さらに呼び出した関数が消費する時間は含まない) を正しく特定しますが、その関数を呼び出す関数を誤って報告します。

synprog のコピー

Sun WorkShop のインストールプログラムによって、synprog ソースファイルは、次のディレクトリにインストールされます。

```
/installation_directory/SUNWspro/WS6/examples/analyzer/synprog
```

デフォルトインストールの *installation_directory* は、/opt になります。

チュートリアルのこの部分を開始するにあたり、作業ディレクトリを作成し、synprog ソースファイルと Makefile をこの作業ディレクトリにコピーします。

```
cp -r installation_directory ~/synprog
```

synprog の作成

synprog プログラムを作成する前に次の作業を実施します。

- **テキストエディタで** Makefile **を開きます。**

  Makefile には、環境変数 ARCH と OFLAGS の代替設定値が含まれています。

  - ARCH については、デフォルト設定値をそのまま使用できます。このデフォルト設定値は、SPARC 7、8、9 の各プラットフォームおよび Intel プラットフォーム上で使用できます。

注 – SPARC のデフォルトアーキテクチャ (-xarch コンパイラフラグ) は、古いシステムに対応するよう、v7 です。一方、Intel のデフォルトアーキテクチャは、ia32 です。ほとんどの新しい SPARC マシンは、v8 をサポートします。使用しているマシンが v8 をサポートしているのであれば、デフォルトの部分をコメントにし、ARCH=-xarch=v8 または ARCH=-xarch=v9 (v9 マシンの場合) という行のコメントを外します。デフォルトを使用すると、整数の乗算命令および除算命令を使用するのではなく、libc.so .mul および .div というルーチンを呼び出すコードが生成されます。また、これらの算術演算に要した時間が <Unknown> 関数に示されます。<未知> 関数の詳細については、107 ページを参照してください。

- Makefile には、OFLAGS の設定値が 2 つ含まれていますが、OFLAGSは、プログラムの最適化に影響します。synprog は、デフォルト設定値 (コマンド行オプションとして、-g、-xF、-v、-V を使用する) で作成し、その synprog 上で標本コレクタを実行し、結果をアナライザで表示することができます。他の設定値を使用してこのプロセスを繰り返すと、異なる設定値によって、コンパイラのコードの最適化方法にどのような影響があるかを把握できます。OFLAGS で設定するさまざまなコンパイラオプションについては、『*C ユーザーズガイド*』を参照してください。

注 – この節に示すサンプルプログラムは、最適化を行わずにコンパイルされています。

---

synprog を作成するには、次の手順に従います。

1. **Makefile を保存し、エディタを閉じます。**
2. **コマンドプロンプトで** make **と入力します。**

synprog に関するデータの収集

synprog に関するパフォーマンスデータを収集するには、次の手順に従います。

1. **次のように入力して、**Sun WorkShop **を起動します。**

```
% workshop
```

2. **「デバッグ」ボタンをクリックし、「デバッグ」ウィンドウを開きます。**

3. synprog **を「デバッグ」ウィンドウに読み込みます。**
   a. **「デバッグ」メニューから「デバッグ」➤「新規プログラム」を選択します。**
   b. **「新規プログラムデバッグ」ダイアログの「名前」フィールドに、**synprog **のパスを入力するか、またはリストボックスを使用して** synprog **までナビゲートします。**
   c. **「了解」をクリックします。**

4. **「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、「標本コレクタ」ウィンドウを開きます。**

   次のようになっているはずです。

   - データ収集が「一度の実行のみ」になっている
   - 実験レコードファイルのデフォルトパスとファイル名が「実験ファイル」フィールドに表示されている
   - 「収集するデータ」チェックボックスで、デフォルトの「時間ベースのプロファイリング」だけが、収集対象として選択されている

   この実験では、時間ベースのプロファイルデータだけを収集し、デフォルトのサンプリング間隔を使用します。したがって、デフォルト設定値を使用します。

5. **「開始」ボタンをクリックします。**

`synprog` は、「デバッグ」ウィンドウで実行され、標本コレクタが時間ベースのプロファイルデータを収集し、デフォルトの実験レコードファイルである `test.1.er` に格納します。

`synprog` パフォーマンスメトリックの解析

アナライザを開き、`test.1.er` をアナライザに読み込むには、次の操作を行います。

1. Sun WorkShop **メインウィンドウまたは「標本コレクタ」ウィンドウにあるツールバーの「解析」ボタンをクリックします。**

2. **「実験ファイルの読み込み」ダイアログで、`test.1.er` と入力し、「了解」をクリックします。**

   アナライザを起動し、コマンド行からを `test.1.er` を読み込むには、次の操作を行います。

- **次のように入力します。**

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

「アナライザ」ウィンドウには、synprog の関数一覧が表示されます。関数一覧には、次に示すデフォルトの時間ベースのプロファイリングメトリックが表示されます。

- 排他ユーザー CPU 時間 (関数自体に費やされた時間) (秒数)
- 包含ユーザー CPU 時間 (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時間)

関数一覧は、専用ユーザー CPU 時間でソートされます。

単純なメトリック解析

まず、cputime() および icputime() という非常に単純な関数の実行時間を見ていましょう。どちらの関数にも for ループが含まれており、変数 x が 1 ずつインクリメントされています。ただし、cputime() の場合、x は浮動小数点であり、icputime() の場合は整数です。

1. **「関数リスト」表示で cputime() と icputime() を見つけます。**

   これら 2 つの関数の専用ユーザー CPU 時間を見ると、cputime() のほうが、icputime() に比べ、はるかに実行に時間がかかっていることに気づくはずです。そこで、別の解析機能を使用して、この理由が何かを探ってみます。

2. **「関数リスト」表示で cputime() をクリックして選択します。**

3. **ウィンドウ下部にある「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。このとき、テキストエディタのウィンドウの幅を広げる必要があるかもしれません。

   実行時間の大部分がループ行と、x がインクリメントされる行で費やされているのがわかります。

```
for(j=0; j<1000000; j++) {
        x = x + 1.0;
}
```

4. **今度は「関数リスト」表示で「icputime()」を選択し、「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 がインクリメントされる行を見てみます。

```
for(j=0; j<1000000; j++) {
        x = x + 1;
}
```

   この icputime() 内のこの行は、対応する cputime() 内の行ほど実行に時間がかかっていません。なお、ループ行自体にかかっている時間は、cputime() 関数とほとんど同じです。

   それでは、これらの関数の注釈付き逆アセンブリコードを見て、このようになっている理由を調べます。

5. **「関数リスト」表示で cputime() を選択し、ウィンドウ下部にある「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。ゼロでない測定値が見つかるまでスクロールダウンします。x がインクリメントされているソースコードの行を見つけます。

   x と 1 の読み込みと加算にはほとんど時間が費やされていませんが、fstod 命令の実行に大部分の時間が費やされているのがわかります。この命令は、x の値を単精度浮動小数点型の値から倍精度浮動小数点型の値に変換し、同じく倍精度浮動小数点型の値として定義されている 1 だけインクリメントできるようにしています。後に、x を単精度浮動小数点型の値に戻す fdtos 指令で、やや CPU 時間をとられています。これら 2 つの演算で、費やされる CPU 時間の約 3/4 を占めています。

6. **「関数リスト」表示で icputime() を選択し、「逆アセンブル」をクリックします。**

   テキストエディタに icputime() の注釈付き逆アセンブリコードが表示されます。x がインクリメントされているソースコード行のパフォーマンスメトリックが見つかるまで、スクロールダウンします。

   関係する操作は、単純な読み込み、加算、格納だけであり、これらに要する時間は、浮動小数点を使用した加算に要する時間の約 1/3 ですんでいます。その理由は、変換が不要だからです。ここでは、値 1 をレジスタにロードする必要もありません。値 1 は直接 x に加算できます。

再帰の効果

synprog プログラムには、再帰呼び出しシーケンスに関する 2 つの例が含まれています。

- 関数 recurse() は、直接的な再帰を示します。recurse() が real_recurse() を呼び出すと、real_recurse() は、テスト条件が満たされるまで自分自身を呼び出します。テスト条件が満たされた時点で、ユーザー CPU 時間をとる処理を実行します。その後、real_recurse() を呼び出し、それが正常終了するという操作を何度か繰り返し、最終的には制御が recurse() に戻ります。

- bounce() という関数は、間接的な再帰を示します。この関数は、bounce_a() という関数を呼び出し、テスト条件が満たされているかどうかをチェックします。テスト条件が満たされていないと、bounce_b() を呼び出します。bounce_b() は、今度 bounce_a を呼び出します。このシーケンスは、bounce_a() でテスト条件が満たされるまで続きます。bounce_a() でテスト条件が満たされると、bounce_a() は、ユーザー CPU 時間をとる何らかのタスクを実行します。その後、bounce_a() と bounce_b() を呼び出し、それらが正常に終了するというプロセスを何度か繰り返しながら、最終的には制御フローが bounce() に戻ります。

いずれの場合も、排他メトリックは、実際に処理が実行される関数に属します。この例では、real_recurse() と bounce_a() が実際にタスクが実行される関数に相当します。これらの排他メトリックは、最終的な関数を呼び出すあらゆる関数に包含メトリックとして渡されます。

まず、recurse() および real_recurse() のメトリックを見ていきます。

1. **「関数リスト」表示で関数 recurse() を見つけ、クリックして選択します。**

   recurse() 関数には、包含ユーザー CPU 時間が表示されますが、その専用ユーザー CPU 時間がゼロになっていることに注意してください。これは、recurse() が real_recurse() の呼び出し以外に何も行なっていないからです。

注 – 場合によっては、recurse() に小さなユーザー排他 CPU 時間値が表示されることがあります。これは、プロファイル実験が統計的な性質をもっており、synprog 上で行う実験が recurse() 関数の消費したごく短かな時間を記録している可能性があるからです。しかし、この排他時間は、包含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、「呼び出し元-呼び出し先」ウィンドウを開きます。このウィンドウには、`recurse()` が関数 `real_recurse()` を呼び出している部分が表示されます。**

3. **`real_recurse()` をクリックして選択します。**

   「呼び出し元-呼び出し先」ウィンドウには、次の情報が表示されます。

   - `recurse()` および `real_recurse()` は両方とも、`real_recurse()` の呼び出し側として呼び出し側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()` が自分自身を再帰的に呼び出すことからも見当がつきます。
   - 表示を見やすくするために、`real_recurse()` が、自分自身によって呼び出される関数として、呼び出され側区画に表示されることはありません。
   - 実際に時間が消費される `real_recurse()` に対しては、排他メトリックと包含メトリックが記録されます。この時間は、`recurse()` の包含メトリックに加算されます。
   - `real_recurse()` については、呼び出し側区画に排他メトリックも表示されます。ある関数によって排他メトリックが生成される場合、Analyzer 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、その関数の排他メトリックを表示します。

   今度は、間接再帰シーケンスではどのようになるかを見てみましょう。

1. **「関数リスト」表示で関数 `bounce()` を見つけ、クリックして選択します。**

   `bounce()` 関数には、包含ユーザー CPU 時間が表示されますが、その排他ユーザー CPU 時間はゼロであることに注意してください。これは、`bounce()` は、`bounce_a()` の呼び出し以外に何も行わないからです。

2. **「呼び出し元-呼び出し先」をクリックし、「呼び出し元-呼び出し先」ウィンドウを開きます。このウィンドウでは、`bounce()` が関数 `bounce_a()` を呼び出している部分だけが表示されます。**

3. **`bounce_a()` をクリックして選択します。**

   「呼び出し元-呼び出し先」ウィンドウには、次の情報が表示されます。

   - `bounce()` および `bounce_b` は両方とも、`bounce_a()` の呼び出し側として、呼び出し側区画に表示されます。

- bounce_b() は、呼び出され側区画にも表示されます。ある関数が自分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され側区画に表示されます。

- bounce_a() には、排他メトリックと包含メトリックの両方が表示されます。bounce_a() は、実際のユーザー CPU 時間が費やされる場所です。これらのメトリックは、bounce_a() を呼び出す関数の包含メトリックにも加算されます。

4. **bounce_b() をクリックして選択します。**

   bounce_b() は、bounce_a() の呼び出し側、および bounce_a() によって呼び出される側の両方として表示されます。bounce_a() の排他メトリックおよび包含メトリックが、呼び出し側区画と呼び出され側区画の両方に表示されます。これは、ある関数によって排他メトリックが生成される場合、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からです。

gprof の誤った推論

ここでは、synprog に対する gprof の誤った推論を、アナライザがどのように解決していくかを見ていきます。

gpf_work() という関数を選択します。これは、synprog がもっとも多く時間を費やす関数の 1 つです。

しかし、この例では、プログラムがどこに時間を費やしているかを知ることよりも、なぜそのようになっているかを考えるほうが重要です。そのために、gpf_work() を呼び出す関数と、それらの関数がどのように gpf_work() を呼び出すかを見てみましょう。

- **「アナライザ」ウィンドウ下部にある「呼び出し元-呼び出し先」をクリックし、「呼び出し元-呼び出し先」ウィンドウを開きます。**

  「呼び出し元-呼び出し先」ウィンドウは、水平方向の 3 つの区画に分かれています。

  - 中央の区画には、選択した関数に関連したデータが表示されます。今回は gpf_work() です。

  - 上の区画には、選択した関数を呼び出すすべての関数に関連したデータが表示されます。この場合は、gpf_b() と gpf_a() です。

- 下の区画には、選択した関数によって呼び出されるすべての関数に関連したデータが表示されます。この場合は、この区画は空になります。なぜなら、gpf_work() は一切他の関数を呼び出さないからです。

呼び出し側の区画を見ると、gpf_work が gpf_b() と gpf_a() という 2 つの関数によって呼び出されているのがわかります。アナライザによれば、gpf_work() の大部分の時間は gpf_b() からの呼び出しで費やされており、gpf_a() からの呼び出しではあまり費やされていません。そこで、gpf_b() からの呼び出しが gpf_a() からの呼び出しに比べ、gpf_work で 10 倍の時間が費やされている理由を知るために、これらの呼び出し側を調べてみる必要があります。

1. **呼び出し側区画で gpf_a() をクリックして選択します。**

   gpf_a() が現在選択されている関数なので、これが中央の区画に移動します。その呼び出し側は上の呼び出し側区画に表示されます。呼び出され側の gpf_work() は、下の呼び出され側区画に表示されます。

2. **メインの「アナライザ」ウィンドウ (gpf_a() が現在選択されている関数として表示されている) に戻り、「ソース」ボタンをクリックしてテキストエディタを開き、gpf_a() の注釈付きソースコードを表示します。**

3. **テキストエディタでスクロールを行い、gpf_a() と gpf_b() の両方のコードを表示します。**

   コードから、gpf_a() が引数に 1 を設定して gpf_work() を 10 回呼び出しているのがわかります。一方、gpf_b() は、gpf_work() を 1 回しか呼び出していませんが、引数に 10 を設定しています。gpf_a() と gpf_b() からの引数は、gpf_work() の仮引数 amt に渡されます。

   今度は、gpf_work() のコードを調べ、gpf_work() の呼び出し方がなぜ違いを生むのかを調べてみましょう。

- **テキストエディタ内で画面をスクロールし、gpf_work() のコードを表示します。**

  imax = 4 * amt * amt という行は、続く for ループの上限を設定しています。この行から考えて、gpf_work() に要する時間がその引数の 2 乗に依存しているのがわかります。引数に 10 を指定した関数からの 1 回の呼び出し (繰り返し回数が 400 回) は、引数に 1 を指定した関数からの 10 回の呼び出し (4 回の繰り返しが 10 回) に比べ、約 10 倍の時間がかかります。

  これは、gprof にどのような関係があるのでしょうか。「gprof の誤った推論」は、呼び出された関数が渡された引数をどのように使用するかを考慮することなく、関数の呼び出し回数に基づいて、その関数に要する時間を見積っていることにあります。

したがって、synprog の解析の場合、gprof であれば、gpf_a() からの呼び出しに、gpf_b() からの呼び出しの 10 倍の時間を当てることになるでしょう。これが gprof の誤った推論です。

関数の CPU 時間が、引数の累乗に依存していればいるほど、このようなゆがみは大きくなります。たとえば、次数が 3 乗される行列乗算であれば、さらに敏感に反映されるでしょう。

動的にリンクされた共用オブジェクトの読み込み

synprog ディレクトリには、2 つの動的にリンクされた共用オブジェクト so_syn.so と so_syx.so があります。実行中に、synprog は、まず so_syn.so を読み込み、それに含まれる関数の so_burncpu() を呼び出します。次に、so_syn.so の読み込みを解除し、so_syx.so をたまたま同じアドレスに読み込み、so_syx.so に含まれる関数の sx_burncpu() を呼び出します。次に、so_syx.so の読み込みを解除せずに、再度 so_syn.so を別のアドレスに読み込み、so_burncpu() を呼び出します。so_syn.so を別のアドレスに読み込んだのは、最初に読み込んだアドレスが別の共用オブジェクトによって使用されているからです。

関数 so_burncpu() および sx_burncpu() は、そのソースコードからわかるようにまったく同じ操作を行います。したがって、実行時に同じユーザー CPU 時間を費やすはずです。

共用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場所を選択します。

このやや手の込んだ練習問題は、同一の関数が異なる実行時点で異なるアドレスから呼び出され得ること、異なる関数が同一のアドレスから呼び出され得ること、およびアナライザがこのような動作を正しく処理し、関数の存在するアドレスに関係なく、その関数のデータを集計していることを示しています。

1. **「関数リスト」表示で sx_burncpu() をクリックして選択します。**
2. **「表示」➤「概要メトリックを表示」を選択します。**

   sx_burncpu() について「概要メトリック」ウィンドウが表示されます。sx_burncpu() のユーザー CPU 時間に注目してください。

3. **次に、so_burncpu() をクリックして選択します。so_burncpu() の概要データが「概要メトリック」ウィンドウに表示されます。**

   so_burncpu() と sx_burncpu() とは処理内容が同じですが、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 倍かかっています。これは、so_burncpu() が 2 回実行されているからです。このように、アナライザは、プログラムの実行中に同じ関数が別のアドレスに読み込まれたとしても、同じ関数が実行されたことを認識し、そのデータを集計します。

例 2：omptest

注 – omptest は、OpenMP 指示が含まれた Fortran プログラムであり、SPARC プラットフォーム上でコンパイルするには Fortran 95 コンパイラが必要です。

omptest プログラムは、OpenMP の並列化機能を使用しており、並列分散の効率をテストするように設計されています。omptest の解析では、次の問題を扱います。

- 並列化の妨げとなるものは何か
- どのようにして負荷をスレッド全体に均等に分散させるか
- メモリー競合およびバス競合によるコストはどのくらいか

注 – この例では、最低でも CPU が 4 台あるマシン上で omptest を実行しているものと仮定しています。

omptest のコピー

Sun WorkShop のインストールプログラムによって、omptest ソースファイルは、次のディレクトリにインストールされます。

```
/installation_directory/SUNWspro/WS6/examples/analyzer/omptest
```

デフォルトインストールの *installation_directory* は、/opt になります。

このチュートリアルのここの部分を開始するにあたって、作業ディレクトリを作成し、omptest ソースファイルと Makefile をこの作業ディレクトリにコピーします。

```
% cp -r installation_directory ~/omptest
```

omptest の作成

omptest プログラムを作成する前に次の作業を実施します。

- **テキストエディタで** Makefile **を開きます。**

  Makefile には、環境変数 ARCH と OFLAGS の代替設定値が含まれています。

  - ARCH については、デフォルト設定値をそのまま使用できます。このデフォルト設定値は、SPARC 7、8、9 プラットフォーム上で使用できます。

注 – SPARC の場合、xarch コンパイラフラグのデフォルトアーキテクチャは、古いシステムに対応するよう、v7 です。omptest の Makefile のデフォルト設定値は、ARCH=-xarch=v8 です。ほとんどの新しい SPARC マシンは、v8 をサポートします。v7 というデフォルトを使用すると、整数の乗算命令および除算命令を使用するのではなく、libc.so .mul および .div というルーチンを呼び出すコードが生成されます。また、これらの算術演算に要した時間が <未知> 関数に示されます。<未知> 関数については、107 ページを参照してください。

  - Makefile には、OFLAGS の設定値が 2 つ含まれていますが、OFLAGS は、最適化と並行化に影響します。omptest は、デフォルト設定値 (コマンド行オプションとして、-g -O3 -mp=openmp -explicitpar -depend -stackvar -loopinfo -v -V) を使用して作成し、その omptest 上で標本コレクタを実行し、結果を標本アナライザで表示することができます。他の設定値を使用してこのプロセスを繰り返すと、異なる設定値によってコンパイラのコードの最適化と並列化にどのような影響があるかを把握できます。OFLAGS 設定値のさまざまなコンパイラオプションについては、『Fortran ユーザーズガイド』を参照してください。

注 – 並行化 (-mp=openmp -explicitpar) を指定しない場合、コンパイラは、OpenMP 指令を解釈せずに、逐次実行するプログラムをコンパイルします。

---

omptest を作成するには、次の操作を行います。

1. Makefile を保存し、エディタを閉じます。
2. コマンドプロンプトで `make` と入力します。

`omptest` に関するデータの収集

デモンストレーションのこの部分では、4 CPU 以上のシステム上で `omptest` を実行しなければなりません。Sun WorkShop をマシン上にインストールしなければなりません。これによって、標本コレクタを実行して、パフォーマンスデータを収集できます。

注 – 並行化戦略および OpenMP 指令の背景については、『Fortran プログラミングガイド』の並行化と OpenMP の章を参照してください。

---

1. 次のように入力して、Sun WorkShop を起動します。

```
% workshop
```

2. 「デバッグ」ボタンをクリックし、「デバッグ」ウィンドウを開きます。

3. `omptest` を「デバッグ」ウィンドウに読み込みます。
   a. 「デバッグ」メニューから「デバッグ」➤「新規プログラム」を選択します。
   b. 「新規プログラムデバッグ」ダイアログの「名前」フィールドに、`omptest` のパスを入力するか、またはリストボックスを使用して `omptest` までナビゲートします。
   c. 「了解」をクリックします。
4. 環境変数 `PARALLEL` に値 `4` を設定します。
   a. 「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」ダイアログを開きます。
   b. 「環境変数」をクリックし、「環境変数」ダイアログを開きます。

c. **「名前」フィールドに** `PARALLEL` **と入力し、「値」フィールドに** `4` **を入力します。**

d. **「環境変数」ダイアログと「実行時の引数の編集」ダイアログを、それぞれ「了解」をクリックして閉じます。**

5. **「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、「標本コレクタ」ウィンドウを開きます。**

   次のようになっているはずです。

   - データ収集が「一度の実行のみ」になっている
   - 実験レコードファイルのデフォルトパスとファイル名が「実験ファイル」フィールドに表示されている。実験レコードファイルの名前は、`omptest.1.er` に変更してください。
   - 「収集するデータ」チェックボックスで、デフォルトの「時間ベースのプロファイリング」だけが、収集対象のデータとして選択されている

     この実験では、時間ベースのプロファイルデータだけを収集して、デフォルトのサンプリング間隔を使用します。したがって、デフォルトの設定を使用します。

6. **「開始」ボタンをクリックします。**

   

   `omptest` は、「デバッグ」ウィンドウで実行され、標本コレクタが時間ベースのプロファイルデータを収集し、デフォルトの実験レコードファイルである `omptest.1.er` に格納します。

7. `PARALLEL` **に** 2 **を設定して、手順** 4 **から手順** 6 **を繰り返し、パフォーマンス情報を** `omptest.2.er` **に保存します。**

`omptest` パフォーマンスメトリックの解析

並行化戦略を解析するために、4 つの関数を調べ、それらを 4 台の CPU で実行する場合と 2 台の CPU で実行する場合とで、動作にどのような違いが生ずるかを比較します。

- `psec_()` は、`PARALLEL SECTION` ルーチン
- `pdo_()` は、`PARALLEL DO` ルーチン
- `critsum_()` は、`CIRTICAL SECTION` ルーチン

- redsum_() は、REDUCTION ルーチン

これらのルーチンはすべて、OpenMT 指令を omptest ソースコードに挿入した結果、コンパイル時に生成されたルーチンです。psec_() と pdo_() の組と、critsum_() と redsum_() の組は、効率的な並行化のための対照的な戦略を示しています。

注 – 並列コードの動作は、予期できないことがあります。このコードの動作の詳細については、98 ページの「並列実行およびコンパイラ生成の本体関数」、および 106 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。

まず、4 台の CPU と 2 台の CPU 上で実行される psec_() と pdo_() のパフォーマンスを見てみましょう。

- **2 つの「アナライザ」ウィンドウを開きます。一方のウィンドウには、omptest.1.er を読み込み、もう一方のウィンドウには、omptest.2 er を読み込みます。これらのウィンドウに表示されるメトリックを比較できるように、これらのウィンドウを位置付けます。**

  各「アナライザ」ウィンドウで次の作業を行います。

1. **「関数リスト」表示で psec_() のデータを見つけ、その行をクリックして選択します。**
2. **「表示」➤「概要メトリックを表示」を選択します。**

   「**概要メトリック**」ウィンドウが表示され、psec_() のメトリックが表示されます。
3. **ユーザー CPU 時間、待ち時間、および総 LWP の包含メトリックを調べます。**

   2 CPU で実行した場合、psec_() のユーザー CPU 時間は約 8.9 秒、待ち時間は約 4.8 秒、総 LWP 時間は約 9.6 秒です。待ち時間と、ユーザー CPU 時間または総 LWP との比率は約 1 対 2 です。これは、比較的並列化の効率がよいことを物語っています。

   一方、4 CPU で実行した場合、psec_() のユーザー CPU 時間は、2 CPU で実行した場合とほとんど同じ (8.4 秒) ですが、待ち時間と総 LWP 時間がともに長くなっています (それぞれ、6.12 秒と 11.55 秒)。psec() の場合、4 CPU マシンで実行した方が時間がかかっています。psec_() PARALLEL SECTION 構造には 2 つのセクションしかないため、それらを実行するには、スレッドが 2 つだけ必要です。したがって、使用可能な 4 台の CPU のうち、同時に使用されているのは 2 台だけです。なお、4 台の CPU マシン上でのパフォーマンスがわずかに劣っているのは、4 台の CPU 間でスレッドをスケジュールするという負荷がかかるためです。

4. **関数 pdo_() までスクロールし、その行をクリックして選択します。**

5. **「アナライザ」のメニューから「表示」➤「概要メトリックを表示」を選択します。**

   「概要メトリック」ウィンドウが表示され、pdo_() のメトリックが表示されます。

6. **ユーザー CPU 時間、待ち時間、および総 LWP の包含メトリックを調べます。**

   pdo_() のユーザー CPU 時間は、psec_() とほぼ同じです (2 CPU で実行した場合は 8.4 秒、4 CPU で実行した場合は約 8.6 秒)。しかし、待ち時間対ユーザー CPU 時間の比率は、2 CPU の場合で約 1 対 2 ですが、4 CPU の場合は、約 1 対 4 になっています。このことは、利用できる CPU の数を考慮し、ループを適切にスケジュールすることで、pdo_() の並行化戦略において CPU の数により良く比例したパフォーマンスを得ることができることを意味します。

   今度は、critsec_() と reduc_() のルーチンの相対的な効率を見てみましょう。この場合、注釈付きのソースコードを見ると、各並行化戦略が、一組のdo ループに埋め込まれた同じ代入文をどの程度効率的に処理しているかがわかります。なお、この do ループの目的は、3 つの 2 次元配列の内容を合計することです。

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

1. **「関数リスト」表示で critsum_() のデータを見つけ、その行をクリックして選択します。**

2. **「アナライザ」ウィンドウの下部にある「ソース」ボタンをクリックします。**

   critsum_() の生成元となるソースコードが注釈付きでテキストエディタに表示されます。

3. **包含ユーザー CPU 時間を調べます。**

   約 13 秒という大変大きな数値になっています。この包含ユーザー CPU 時間が非常に大きいのは、critsum_() が重要領域の並行化戦略を使用しているからです。すなわち、加算演算は 4 台すべての CPU に分散されますが。t という値を sum に合計できるのは、同時に 1 台の CPU だけです。これは、並行化を有効に活用した戦略とは言えません。

4. **テキストエディタを閉じて、redsum_() のデータまでスクロールします。その行をクリックして選択します。**

5. **「アナライザ」ウィンドウの下部にある「ソース」をクリックします。**

   `redsum_()` の生成元となるソースコードが注釈付きでテキストエディタに表示されます。

6. **包含ユーザー CPU 時間を調べます。**

   ここでは、約 1.7 秒と大幅に値が縮小されています。これは、`redsum_()` が削減戦略を使用しているからです。削減戦略によって、`(a(j,i)+b(j,i)+c(j,i))/k` の部分合計の計算が複数の CPUに分散して行われます。その後、これらの中間値が `sum` に加算されます。この戦略によって、利用可能な CPU の有効活用が行われています。

例 3：`mttest`

`mttest` プログラムは、クライアント-サーバ環境にあるサーバをエミュレートします。ここでは、クライアントが要求をキューに入れ、サーバが複数のスレッドを使用して、それらにサービスを提供します。このとき、明示的にスレッド機能を使用します。`mttest` で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競合と、実行時にキャッシュ機能を使用する効果を示します。

`mttest` のコピー

Sun WorkShop のインストールプログラムによって、`mttest` ソースファイルは、次のディレクトリにインストールされます。

```
/installation_directory/SUNWspro/WS6/examples/analyzer/mttest
```

デフォルトインストールの *installation_directory* は、`/opt` になります。

このチュートリアルのここの部分を開始するにあたって、作業ディレクトリを作成し、`mttest` ソースファイルと Makefile をこの作業ディレクトリにコピーします。

```
% cp -r installation_directory ~/mttest
```

mttest の作成

mttest プログラムを作成する前に次の作業を実施します。

- **テキストエディタで** Makefile **を開きます。**

  Makefile には、環境変数 ARCH、OFLAGS、THREADS および FLAG の代替設定値が含まれています。

  - ARCH については、デフォルト設定値をそのまま使用できます。このデフォルト設定値は、SPARC 7、8、9 の各プラットフォームと、Intel プラットフォーム上で使用できます。

注 – SPARC のデフォルトアーキテクチャ (-xarch コンパイラフラグ) は、古いシステムに対応するよう、v7 です。一方、Intel のデフォルトアーキテクチャは、ia32 です。ほとんどの新しい SPARC マシンは、v8 をサポートします。使用しているマシンが v8 をサポートしているのであれば、デフォルトの部分をコメントにし、ARCH=-xarch=v8 または ARCH=-xarch=v9 (v9 マシンの場合) という行のコメントを外します。デフォルトを使用すると、整数の乗算命令および除算命令を使用するのではなく、libc.so .mul および .div というルーチンを呼び出すコードが生成されます。また、これらの算術演算に要した時間が <未知> 関数に示されます。<未知> 関数の詳細については、107 ページを参照してください。

  - Makefile には、OFLAGS の設定値が 2 つ含まれていますが、OFLAGS は、プログラムの最適化と並行化に影響します。Omptest は、デフォルト設定値 (コマンド行オプションとして、-g、-xF、-v、-V を使用する) で作成し、その omptest 上で標本コレクタを実行し、結果をアナライザで表示することができます。他の設定値を使用してこのプロセスを繰り返すと、異なる設定値によって、コンパイラのコードの最適化方法と並列化方法にどのような影響があるかを把握できます。OFLAGS 設定値のさまざまなコンパイラオプションについては、『C ユーザーズガイド』を参照してください。

  - THREADS および FLAG については、ターゲットアプリケーションにもっとも近い設定値をコメントにし、それ以外の設定値のコメントを解除します。THREADS、SOLARIS および POSIX のいずれかを設定します。これによって、どちらのスレッド標準用にプログラムがコンパイルされるのかが決まります。FLAGS の設定値は、BOUND および UNBOUND であり、プログラムが結合スレッドまたは非結合スレッドのどちらを使用するかを決定します。

mttest を作成するには、次の操作を行います。

1. Makefile を保存し、エディタを閉じます。
2. コマンドプロンプトに対し make と入力します。

mttest に関するデータの収集と解析

ここで作成した実行可能プログラム mttest は、明示的なマルチスレッド機能を使用するようにコンパイルされ、複数の CPU が搭載されたマシンまたは CPU が 1 台搭載されたマシン上でマルチスレッドプログラムとして実行します。CPU が 1 台搭載されたマシン上で mttest を実行すると、パフォーマンスメトリックに関する興味深い相違点と類似点がわかります。

次の 2 つの節「mttest に関するデータ収集 (4 CPU システム)」および 「mttest パフォーマンスメトリックの解析 (4 CPU システム)」では、4 CPU マシン上での mttest のパフォーマンスに注目してください。さらに続く 2 つの節 27 ページの「mttest に関するデータ収集 (1 CPU システム)」および 29 ページの「mttest パフォーマンスメトリックの解析 (1 CPU システム)」では、1 CPU マシン上での同じパフォーマンスメトリック群に注目してください。

mttest に関するデータ収集 (4 CPU システム)

デモンストレーションのこの部分では、4 CPU システム上で mttest を実行させなければなりません。標本コレクタを実行して、パフォーマンスデータを収集するには、Sun WorkShop をマシン上にインストールしておく必要があります。

1. 次のように入力して、Sun WorkShop を起動します。

```
% workshop
```

2. 「デバッグ」ボタンをクリックし、「デバッグ」ウィンドウを開きます。

3. mttest を「デバッグ」ウィンドウに読み込みます。

   a. 「デバッグ」メニューから「デバッグ」➤「新規プログラム」を選択します。

b. **「新規プログラムデバッグ」ダイアログの「名前」フィールドに、`mttest` のパスを入力するか、リストボックスを使用して `mttest` までナビゲートします。**

c. **「了解」をクリックします。**

4. **「デバッグ」ウィンドウメニューから「ウィンドウ」➤「標本コレクタ」を選択し、「標本コレクタ」ウィンドウを開きます。**

   次のようになっているはずです。

   - データ収集が「一度の実行のみ」になっている
   - 実験レコードファイルのデフォルトパスとファイル名が「実験ファイル」フィールドに表示されている。この実験レコードファイルの名前は、`mttest.1.er` に変更してください。
   - 「収集するデータ」チェックボックスで、デフォルトの「時間ベースのプロファイリング」だけが、収集対象のデータとして選択されている

5. **この実験では、時間ベースのプロファイリングに加え、同期待ちの監視情報も収集します。そのため「同期待ちの監視」チェックボックスを選択します。**

6. **「開始」ボタンをクリックします。**

   

   `mttest` は、「デバッグ」ウィンドウで実行され、標本コレクタが時間ベースのプロファイルデータおよび同期待ちの監視データを収集し、それを実験レコードファイル `mttest.1.er` に格納します。

`mttest` パフォーマンスメトリックの解析 (4 CPU システム)

アナライザを開き、`mttest.1.er` を読み込むには、次の操作を行います。

1. Sun WorkShop **メインウィンドウまたは「標本コレクタ」ウィンドウのツールバー上の「解析」ボタンをクリックします。**

   

2. **「実験ファイルの読み込み」ダイアログで `mttest.1.er` と入力し、「了解」をクリックします。**

アナライザを起動し、コマンド行からを mttest.1.er を読み込むには、次の操作を行います。

- **次のように入力します。**

```
% analyzer mttest.1.er
```

「アナライザ」ウィンドウには、mttest の「関数リスト」が表示されます。

1. **「関数リスト」表示で lock_local() および lock_global() のデータまでスクロールダウンします。**

   これらの関数の包含ユーザー CPU 時間は同じです (約 3.2 秒)。これは、両方の機能が同じ量のタスクを実行していることを示しています。

   ただし、lock_global() は、同期待ちに多くの時間 (約 5秒) を費やしていますが、lock_local() は、同期待ちには時間を費やしていません。これらの関数の注釈付きソースコードを表示してみると、その理由がわかります。

2. **lock_global() のデータ行をクリックし、それを選択します。**

3. **「アナライザ」ウィンドウの下部にある「ソース」ボタンをクリックします。**

   テキストエディタに lock_global() 関数のソースコードが注釈付きで表示されます。このコードから、lock_global() がデータに大域ロックをかけているのがわかります。

```
mutex_lock(&global_lock);
```

   大域ロックのために、実行中の全スレッドでデータへのアクセスのための競合が発生しています。同時にアクセスできるのは 1 つのスレッドだけです。残りのスレッドがデータにアクセスするためには、作業中のスレッドがロックを解くまで待たなければなりません。

4. **lock_local() のデータ行をクリックし、それを選択します。**

5. **「アナライザ」ウィンドウの下部にある「ソース」ボタンをクリックします。**

   テキストエディタに lock_local() 関数のソースコードが注釈付きで表示されます。このコードから、lock_local() が特定のスレッドの作業ブロック内にあるデータにだけロックをかけているのがわかります。

   ```
   mutex_lock(&(array->lock));
   ```

   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。lock_local() の待ち時間は、ゼロ秒です。

6. **「アナライザ」ウィンドウの「関数リスト」表示に戻り、関数 computeA() および computeB() のデータまでスクロールします。**

7. **computeA() のデータ行をクリックして選択します。**

8. **「アナライザ」ウィンドウの下部にある「ソース」ボタンをクリックします。**

   テキストエディタが開き、computeA() のソースコードが注釈付きで表示されます。

9. **テキストエディタをスクロールダウンし、computeA() および computeB() のソースコードを表示します。**

   これらの関数のコードは実質的に同じ (ループのところで変数に 1 を加算している) ですが、computeB() は、包含ユーザー CPU 時間のほぼ 10 秒をここで費やし、computeA() は、約 3.3 秒しか費やしていません。このような相違点の理由を見つけるために、computeA() と computeB() を呼び出しているコードを調べてみる必要があります。

10. **「アナライザ」ウィンドウの「関数リスト」表示に戻り、computeA() のデータ行をクリックして選択します。**

11. **ウィンドウの下部にある「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機能が、上の区画にその呼び出し側が表示されます。

12. **呼び出し側の lock_none() をクリックして選択します。**

13. **「アナライザ」ウィンドウの「関数リスト」表示に戻ります。**

    ここで、lock_none() が現在選択されている関数であることに注意してください。

14. **「アナライザ」ウィンドウの下部にある「ソース」ボタンをクリックします。**

    テキストエディタが開き、`lock_none()` のソースコードが注釈付きで表示されます。

15. **スクロールダウンし、`computeB()` を呼び出している `cache_trash()` という関数のコードを表示します。**

16. **`computeA()` および `computeB()` の呼び出し方を比較します。**

    `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]`) を引数として使用して呼び出されています。これは、他のスレッドと競合することなく、直接読み取りと書き込みを行うことができます。

    `computeB()` は、メモリー内で連続した語を占め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ています。あるスレッドがメモリー内にあるこれらのアドレスの 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しなければなりません。なぜなら、そのデータはすでに古くなっているからです。後にスレッドの 1 つがプログラムでそのデータを必要とした場合は、メモリーからデータキャッシュにコピーし直さなければなりません。これは非常に時間のかかる操作です。キャッシュ上にデータが無い、すなわち、データキャッシュ上で使用可能でないデータにアクセスしようとすることによって、多くの CPU 時間が無駄に費やされます。これが、`computeB()` が `computeA()` に比べ 3 倍もの CPU 時間がかかっている理由です。

17. **「アナライザ」ウィンドウの「関数リスト」表示に戻り、`computeB()` の行をクリックして選択します。**

18. **ウィンドウの下部にある「逆アセンブル」をクリックします。**

    テキストエディタが開き、`computeB()` の逆アセンブリコードが注釈付きで表示されます。ユーザー CPU 時間 (7 秒を超える) の大部分が `fadd` 命令に費やされているのがわかります。fadd 命令は、キャッシュ上にないレジスタの読み込みを待っています。

`mttest` に関するデータ収集 (1 CPU システム)

デモンストレーションのこの部分では、1 CPU システム上で `mttest` を実行しなければなりません。まず、標本コレクタを実行して、パフォーマンスデータを収集するには、Sun WorkShop をマシン上にインストールしておく必要があります。

1. **次のように入力して、Sun WorkShop を起動します。**

```
% workshop
```

2. **「デバッグ」ボタンをクリックし、「デバッグ」ウィンドウを開きます。**

3. **`mttest` を「デバッグ」メニューに読み込みます。**

   a. **「デバッグ」メニューから「デバッグ」➤「新規プログラム」を選択します。**

   b. **「新規プログラムデバッグ」ダイアログの「名前」フィールドに、`mttest` のパスを入力するか、リストボックスを使用して `mttest` までナビゲートします。**

   c. **「了解」をクリックします。**

4. **「デバッグ」ウィンドウメニューバーから「ウィンドウ」➤「標本コレクタ」を選択し、「標本コレクタ」ウィンドウを開きます。**

   次のようになっているはずです。

   - データ収集が「一度の実行のみ」になっている
   - 実験レコードファイルのデフォルトパスとファイル名が「実験ファイル」フィールドに表示されている。この実験レコードファイルの名前は、`mttest.2.er` に変更してください。
   - 「収集するデータ」チェックボックスで、デフォルトの「時間ベースのプロファイリング」だけが、収集対象のデータとして選択されている

5. **この実験では、時間ベースのプロファイリングに加え、同期待ちの監視情報も収集します。そのため「同期待ちの監視」チェックボックスを選択します。**

6. **「開始」ボタンをクリックします。**

`mttest` は、「デバッグ」ウィンドウで実行され、標本コレクタが時間ベースのプロファイルデータおよび同期待ちの監視データを収集し、それをデフォルトの実験レコードファイル `mttest.2.er` に格納します。

mttest パフォーマンスメトリックの解析 (1 CPU システム)

アナライザを開き、mttest.2.er を読み込むには、次の操作を行います。

1. **次のように入力します。**

```
% analyzer mttest.2.er
```

「アナライザ」ウィンドウには、mttest の関数一覧が表示されます。

2. **「関数リスト」表示で lock_local() および lock_global() のデータまでスクロールダウンします。**

4 CPU システムの場合と同様に、これらの関数の包含ユーザー CPU 時間は同じです (約 10 秒)。このことは、両方の関数のタスク量が同じであることを示しています。

ただし、lock_global() の総 LWP 時間は、実際に lock_local() より少なく (25 秒対 37秒) なっています。それぞれのロックシステムのスレッドに CPU を割り当てる方法の違いによって、このようなことが起こり得ます。lock_global() で設定された大域ロックを使用すると、各スレッドは、順番に処理が完了するまで実行することができます。つまり、最初のスレッドが 2.5 秒間実行し終了するまでの間、他のスレッドは待ち状態になります。次に、最初のスレッドを待っていた 2 番目のスレッドが、2.5 秒間実行します。この時点で総 LWP 時間は 5 秒になります。3 番目のスレッドは、最初の 2 つのスレッドの終了を 5 秒待たされた後に実行され、総 LWP 時間は 7.5 秒になります。4 番目のスレッドは、7.5 秒待たされた後に実行され、総 LWP 時間は 10 秒になります。一方、lock_local() の局所ロックの場合は、CPU 上の各スレッドは、少しずつ実行するようにスケジュールされ、すべてのスレッドが完了するまで、この処理が繰り返されます。4 つのすべてのスレッドは、10 秒という実行時間のほぼ 3/4 が待ち状態にあります。

lock_global() は、多くの時間を同期待ちに費やしています (約 15 秒：2.5 + 5 + 7.5) が、lock_local() には、同期待ちの時間がありません。lock_local() は、特定のスレッドの作業ブロック内にあるデータにだけロックをかけます。lock_local() の処理は、4 CPU システムの場合と同様に、競合や同期待ちによる無駄時間なしに行われます。lock_local() の同期待ち時間はゼロ秒です。

3. **「アナライザ」ウィンドウの「関数リスト」表示に戻り、関数 computeA() および computeB() のデータまでスクロールします。**

4. **computeA() のデータ行をクリックして選択します。**

5. **「アナライザ」ウィンドウの下部にある「ソース」ボタンをクリックします。**

   テキストエディタが開き、`computeA()` のソースコードが注釈付きで表示されます。

6. **テキストエディタをスクロールダウンし、`computeA()` および `computeB()` のソースコードを表示します。**

   1 CPU システムの場合、これらの関数の包含 CPU 時間はほぼ同じです。

7. **「アナライザ」ウィンドウの「関数リスト」表示に戻り、`computeA()` のデータ行をクリックして選択します。**

8. **ウィンドウの下部にある「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 `computeA()` が、上の区画にその呼び出し側が表示されます。

9. **呼び出し側である `cache_trash()` をクリックして選択します。**

10. **「アナライザ」ウィンドウの「関数リスト」表示に戻ります。**

    現在、`cache_trash()` が選択されています。

11. **ウィンドウの下部にある「ソース」ボタンをクリックします。**

    テキストエディタが開き、`cache_trash()` のソースコードが注釈付きでが表示されます。

12. **スクロールアップし、`computeA()` を呼び出している `lock_none()` という関数のコードを表示します。**

13. **`computeA()` および `computeB()` の呼び出し方を比較します。**

    `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。これは、他のスレッドと競合することなく、直接読み取りと書き込みを行うことができます。

    `computeB()` は、一続き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ています。この一連の倍精度浮動小数点数型の値は、メモリー内で連続した語を占め、データキャッシュを介してアクセスされます。複数のスレッドが複数の CPU 上で実行されているときに、あるスレッドがメモリー内にあるこれらのアドレスの 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しなければなりません。なぜなら、そのデータはすでに古くなっているからです。後にスレッドの 1 つがプログラム

でそのデータを必要とした場合は、メモリーからデータキャッシュにコピーし直さなければなりません。これは非常に時間のかかる操作です。キャッシュ上でのデータ消失によって、多くの CPU 時間が無駄に費やされます。

しかし、実行中のスレッドが 1 つだけで、かつ他のスレッドがメモリーに書き込みを行なっていない場合は、実行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。キャッシュにデータが無いためにメモリーからコピーすることもありません。したがって、1 CPU マシン上の `computeB()` のパフォーマンスは、`computeA()` のパフォーマンスと同様に良好になります。

第3章

標本コレクタリファレンス

この章では、標本コレクタを紹介し、その使用方法を説明します。以下の各項目を取り上げます。

- 標本コレクタが収集するデータ
- Sun WorkShopでのパフォーマンスデータの収集
- dbx で標本コレクタ下のプロセスを起動
- 実行中プロセスへの接続
- MPI を使用するプログラム

標本コレクタは、ターゲットアプリケーションおよびそのアプリケーションが実行されているカーネルのパフォーマンスデータを収集し、そのデータを実験レコードファイルに書き込みます。本書では、アプリケーションの実行中にデータを収集すること、または収集されたデータを、**実験**と呼ぶことがあります。

特に指定しない限り、標本コレクタで生成された実験レコードファイルには、*.n*`.er` という拡張子が付きます。*n* は 1 以上の整数です。デフォルトの実験ファイル名は、`test`.*n*`.er` になります。実験レコードファイルに *filename*`.1.er` という形式を使用する場合、標本コレクタは、以降の実験レコードファイル名を自動的にインクリメントします。たとえば、`my_test.1.er` に続いて、`my_test.2.er`、`my_test.3.er` となります。

注意 – 実験レコードフィルの削除に `rm` ユーティリティを使用しないでください。実際の実験情報は、*.filename.n*`.er` という隠れディレクトリに格納されており、このディレクトリは `rm` によって削除されないためです。実験レコードファイルと隠れディレクトリを削除するには、パフォーマンスツールユーティリティ

の er_rm を使用します。これは、標本コレクタとアナライザとともに提供されています。er_rm の使用方法については、er_rm のマニュアルページを参照してください。

標本コレクタが収集するデータ

標本コレクタは実験中に、パフォーマンスデータを記録し、そのデータを**一定間隔ごとの標本**として編成します。標本コレクタは、以下の状況で標本を終了し、次の標本に移ります。

- ブレークポイントを見つけたとき (Sun WorkShop デバッガでのブレークポイントの設定については『Sun WorkShop の概要』を参照)
- サンプリング間隔を設定した場合は、設定間隔が経過とき
- 手動サンプリングを選択した場合は、「コレクタ」➤「新規標本」を選択したとき、または「新規標本」ボタンをクリックしたとき

各標本に記録されるデータは、カーネルからの微小状態記録アカウンティング情報とカーネル内で保持されているその他のさまざまな統計情報とからなります。

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、関数レベルのメトリック (測定結果) が含まれていません。ただし、標本収集の合間に関数レベルのメトリックが記録された場合、標本コレクタはこれらの関数メトリックをそれが収集されたサンプリング間隔に関連付けます。

標本コレクタは、次に示す関数レベルの情報を収集できます。

- 時間ベースのプロファイルデータ
- スレッド同期待ちの監視
- ハードウェアカウンタオーバフロープロファイリング

排他メトリック、包含メトリック、寄与メトリック

標本コレクタは、関数および読み込みオブジェクトに対する、排他メトリック、包含メトリック、寄与 (属性) メトリックを収集します。

- 排他データは、関数自身に費やされている時間に適用されます。

- 包含データは、関数自身に費やされている時間と、その関数が呼び出した任意の関数に費やされている時間に適用されます。呼び出された関数の時間は、その呼び出す側の関数から呼び出された場合にのみ加算されます。
- ある関数の寄与データは、その関数から呼び出された関数で発生したメトリックと、そのような呼び出しの結果、呼び出された関数が新たに呼び出した関数で発生したメトリックの合計で。寄与メトリックには、次の条件が適用されます。
  - ある関数を呼び出した関数の寄与メトリックとは、呼び出された関数で発生したメトリック、およびそのような呼び出しの結果、呼び出された関数が呼び出した任意の関数 (複数も可) で発生したメトリックを指します。
  - 呼び出し側の寄与メトリックは、呼び出された関数がその呼び出し側の包含メトリックに寄与分に相当します。
  - ある関数のすべての呼び出し側の寄与メトリック合計は、その呼び出された関数の包含メトリックと同じ値です。
  - ある関数に呼び出された関数の寄与メトリックは、呼び出された関数の、呼び出し側の関数から呼び出された結果としての、包含メトリックの一部となります。
  - 呼び出された関数の寄与メトリックと包含メトリックとの差は、呼び出された関数の包含メトリックのうち、その呼び出し関数以外から呼び出された結果発生したものに相当します。
  - ある関数の包含メトリックは、その排他メトリックに、その関数から呼び出された関数の全寄与メトリックの合計を加算したものになります。

時間ベースのプロファイルデータ

時間ベースのプロファイリングは、次のメトリックをサポートするための情報を記録します。

- **ユーザー** CPU **時間** - アプリケーションが CPU 上で動作している時間
- **総** LWP **時間** - 全 LWP (軽量プロセス) の実行時間の合計
- **時計時間** - スレッド 1 で経過した LWP 時間
- **システム** CPU **時間** - オペレーティングシステムが費やした総 CPU 時間 (LWP がトラップ状態にある総 CPU 時間)
- **システム待ち時間** - CPU、ロックまたはカーネルページ待ちで経過した LWP 時間 (休眠や停止に費やされた時間)

- **テキストページフォルト時間** - テキストページ待ちで経過した LWP 時間
- **データページフォルト時間** - データページ待ちで経過した LWP 時間

この情報は、アナライザの「関数リスト」表示と「呼び出し元-呼び出し先」ウィンドウに表示されます (52 ページの「関数およびロードオブジェクトの測定結果の検査」を参照)。この情報は、「概要メトリック」ウィンドウおよび注釈付きのソースと逆アセンブリコードにも表示されます。

注 – マルチプロセッサを使用した実験では、時計時間以外の時間は、プロセス内の全 LWP について合計されます。総時間は、時計時間に、プロセス内に含まれる平均 LWP 数を掛けた値に等しくなります。各レコードには、タイムスタンプ、およびごく短時間に実行していた LWP とスレッドの ID が含まれます。

時間ベースのプロファイリングは、次のような疑問を解消するのに役立ちます。

- アプリケーションは利用可能な資源のどのくらいを消費しているか
- どの関数が資源の大半を消費しているか
- どのソース行および逆アセンブリ命令が資源の大半を消費しているか
- プログラムはどのようにして実行のこの地点に到着したか

スレッド同期待ちの監視

マルチスレッドプログラムでは、スレッド同期待ちの監視機能は、スレッドライブラリ内にあるスレッド同期ルーチンによる待ち時間を追跡しています。リアルタイム遅延が、ある一定のユーザ定義しきい値を超えると、その呼び出しと待ち時間 (秒数) に関するイベントが記録されます。

各レコードには、タイムスタンプと、クロックスタンプ時に実行中であったスレッドと LWP の ID が含まれます。同期遅延情報は、次のメトリックをサポートしています。

- **同期遅延イベント** - 待ち時間が規定のしきい値を超過した場合の、同期ルーチンの呼び出しの回数
- **同期待ち行列** - 規定のしきい値を超過した待ち時間の合計

この情報は、標本アナライザの「関数リスト」表示と「呼び出し元-呼び出し先」ウィンドウに表示されます (52 ページの「関数およびロードオブジェクトの測定結果の検査」を参照)。この情報は、「概要メトリック」ウィンドウおよび注釈付きのソースと逆アセンブリコードにも表示されます。

ハードウェアカウンタのオーバフロープロファイル

ハードウェアカウンタのオーバフローのプロファイルは、LWP が動作している CPU の指定のハードウェアカウンタがオーバフローを起こしたときに、各 LWP の呼び出しスタックを記録します。

標本コレクタを使用すると、オーバフローが監視されているカウンタの種類を選択し、それにオーバフロー値を設定することができます。通常、カウンタは、命令キャッシュの消失、データキャッシュの消失、サイクル、発行または実行された命令などを追跡します。

注 – ハードウェアカウンタのオーバフロープロファイル機能が使用できるのは、SPARC (UltraSPARC III) マシンおよび Intel (Pentium II およびその互換製品) で実行される Solaris 8 だけです。その他のマシン上では使用できません。

ハードウェアカウンタのオーバフローのプロファイルは、カウンタメトリックをサポートするデータを生成します。

大域情報

プログラムに関する大域情報には、次のようなデータが含まれます。

- **実行統計** - ページフォルトおよび I/O データ、コンテキスト切り替え、およびさまざまなページ存在 (ワーキングセットおよびページング) 統計情報が含まれます。この情報は、標本アナライザの「実行統計」表示に表示されます (77 ページの「実行の統計情報の検査」を参照)。
- **アドレス空間データ (任意)** - アプリケーションのアドレス空間の各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からなります。この情報は、標本アナライザの「アドレス空間」表示に表示されます (75 ページの「アドレス空間情報の検査」を参照)。

Sun WorkShop**でのパフォーマンスデータの収集**

データを収集するにあたって、次の作業が必要です。

- プログラムを「デバッグ」ウィンドウに読み込みます (Sun WorkShop の起動方法および「デバッグ」ウィンドウへのアクセス方法については、『Sun WorkShop の概要』を参照)。
- ランタイムチェック機能がオフ (デフォルト) になっていることを確認します。

データ収集は、次の 2 段階で行います。

1. 収集したいデータの種類と、データを格納したい場所を指定します。
2. 標本コレクタを実行します。

収集したいデータの種類を指定するには、次の操作を行います。

1. WorkShop **メインウィンドウから「ウィンドウ」➤「標本コレクタ」を選択します。**

   「標本コレクタ」ウィンドウが表示されます。

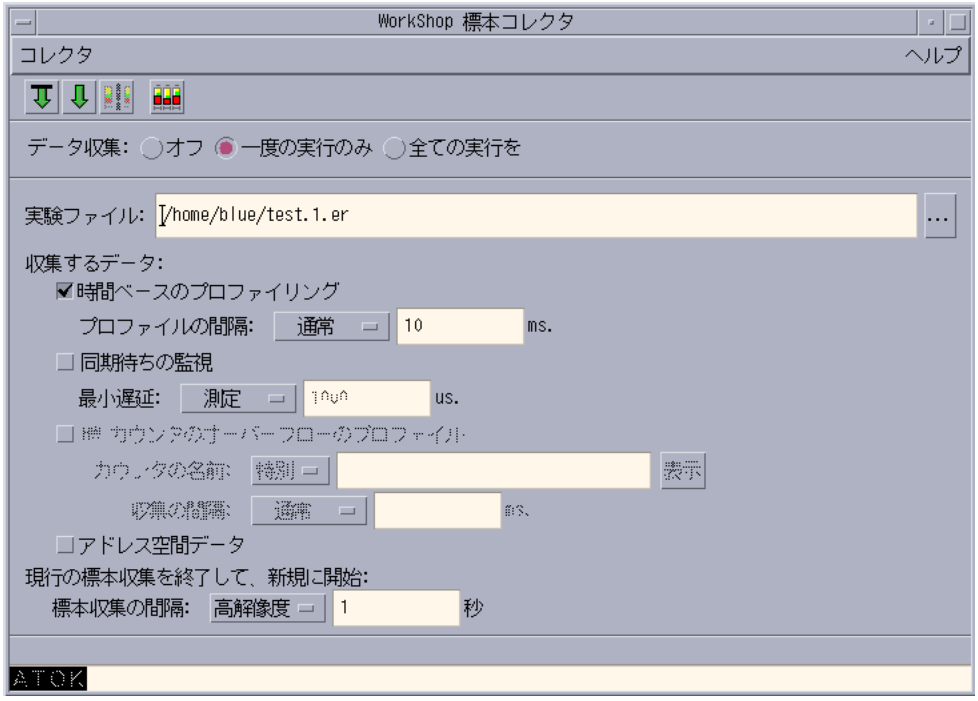

図 3-1　「標本コレクタ」ウィンドウ

2. **「データ収集」ラジオボタンを使用して、1 回の実行または全ての実行でデータを収集するかのいずれかを指定します。**

   - 「一度の実行のみ」を選択した場合は、プログラムを 1 回実行した後、標本コレクタを無効にして、その実行データを実験レコードファイルに格納します。
   - 「すべての実行」を選択した場合は、プログラムの実行が終了しても、標本コレクタは動作状態のままです。以降の実行ごとに、標本コレクタは、新しい実験レコードファイルを作成し、その実行データを格納します。
   - 「オフ」を選択した場合は、これ以外の「データ収集」ラジオボタンを押すまで標本コレクタを無効にして、データの収集と格納を行いません。

3. **「実験ファイル」フィールドに、データを格納したい実験レコードファイルのパスとファイル名を指定します。**

   標本コレクタが指定するデフォルトの実験レコードファイル名は、`test.1.er` です。別の名前を付けたい場合は、そのパス (デフォルトディレクトリに格納したくない場合) とファイル名を入力します。

   実験レコードファイル名に `.1.er` という接尾辞を使用すると、標本コレクタは、以降の実験レコードファイルの名前を自動的に 1 つずつ加算します。たとえば、`test.1.er` の次は、`test.2.er` になります。

4. **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」チェックボックス選択されていることを確認します (デフォルトで選択)。**

   「通常」プロファイル間隔 (10ミリ秒) をそのまま使用することも、リストボックスから「高解像度」 (1 ミリ秒) または「カスタム」を選択することもできます。「カスタム」を選択した場合は、独自の間隔をミリ秒単位で指定します。

   高解像度プロファイルは、どのような実行に対しても、通常プロファイルと比較して 10 倍のデータを記録します。高解像度プロファイルをサポートするには、オペレーティングシステムが高解像度クロックルーチンを実行していなければなりません。高解像度ルーチンを指定するには、`/etc/system` というファイルに次の行を追加して、リブートします。

```
set hires_tick=1
```

注 – 高解像度プロファイルをサポートしていないオペレーティングシステム上でこの機能を設定しようとすると、標本コレクタは、警告メッセージを出力し、サポートされている解像度のうち、もっとも高いものを使用します。カスタム設定値がシステムでサポートされている解像度の倍数でない場合は、その解像度にもっとも近い倍数に丸められ、標本コレクタが警告メッセージを出力します。

---

5. **スレッド同期待ち回数および待ち時間についての情報を収集するには、「同期待ちの監視」チェックボックスを選択します。**

   監視が開始するしきい値を指定するには、次の操作を行います。

   - デフォルトの「測定」(しきい値は実行時に決定されます) をそのまま使用することができます。
   - 「最小遅延」リストボックスから、次のいずれかのしきい値を選択できます。

- 「1000 us.」
- 「100 us.」
- 「すべて」(待ち時間に関係なく、すべての同期待ちが監視される)
- 「カスタム」(独自のしきい値をマイクロ秒で設定する)

6. **ハードウェアカウンタオーバフローに関する情報を収集するには、「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選択します。**
7. **「カウンタの名前」からカウンタの種類を選択し、「表示」をクリックしてそのカテゴリで使用できる全カウンタの一覧を表示します。ユーザーが認識できるカウンタ名のうち、選択したものが「カウンタの名前」フィールドに表示されます。**
8. **あるオーバフローイベントと次のオーバフローイベントとの間で加算される数値を指定するには、「収集の間隔」からデフォルトの「通常」(選択したカウンタによって値が異なる) を選択するか、または「カスタム」を選択して「収集の間隔」フィールドに値を入力します。**

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。したがって、利用可能なカウンタの一覧は、システムごとに異なります。一部のシステムでは、ハードウェアカウンタのオーバフロープロファイルをサポートしていません。そのようなシステムでは、このオプションが使用不可になっています。

---

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選択します。**
10. **標本収集の間隔については、デフォルトの高解像度収集間隔 (1秒間隔) をそのまま使用するか、「標本収集の間隔」リストボックスから「通常」(10 秒間隔)、「カスタム」(独自の間隔を秒数で設定) または「手動」を選択することができます。なお、「手動」を選択した場合は、「標本コレクタ」ウィンドウの「コレクタ」➤「新規標本」を選択するか、または「新規標本」ボタンをクリックして現在の標本の終了と次の標本の開始を指定します。**

以上で、データ収集を開始できる状態になりました。標本コレクタを実行するには、「標本コレクタ」ウィンドウで、次の作業を行います。

- **「コレクタ」➤「開始」を選択するか、または「開始」ボタンをクリックします。**

dbx で標本コレクタ下のプロセスを起動

標本コレクタは、「デバッグ」ウィンドウからだけでなく dbx からも起動できます。

1. **dbx でプログラムを起動するには、次のように入力します。**

```
% dbx program_name
```

2. **(dbx) のプロンプトが表示されるまで、スペースバーを押します。**

3. **標本コレクタコマンドにさまざまな引数を指定してデータを収集し、実験レコードを生成します。**

```
(dbx)collector argument
```

表 3-1 に、collector コマンドの引数を示します。

表 3-1 collector コマンドの引数

引数	機能
{ enable | enable_once | disable }	データ収集を動作状態または非動作状態にする • モードが enable の場合、データ収集は、現在の実行と以降のすべての実行で行われる • モードが enable_once の場合、現在の実行についてはデータが収集されるが、その実行が終了すると、モードは disable にリセットされる • モードが disable の場合、パフォーマンスデータは収集されない
profile { on | off }	プロファイルデータの収集をオンまたはオフにする。デフォルトは**オン**
profile timer *value*	プロファイルタイマ間隔に、ミリ秒単位で指定した *value* を設定する。デフォルト値は、10 ms
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) の収集をオンまたはオフにする。デフォルトは**オフ**
synctrace { on | off }	スレッド同期待ちの監視データの収集をオンまたはオフにする。デフォルトは**オフ**
synctrace threshold *value*	同期遅延トレース機能のしきい値を、ミリ秒単位で指定した *value* に従って設定する。*value* には次のいずれかの値を設定できる • calibrate：実行時に決定されるキャリブレートされたしきい値を使用する • *number*：ミリ秒単位で指定した *number* のしきい値を使用する。*number* に**ゼロ**を設定すると、コレクタは、待ち時間に関係なくすべてのイベントをトレースする デフォルト設定値は calibrate

表 3-1 `collector` コマンドの引数 (続き)

引数	機能
`hwprofile` { `on` | `off` }	ハードウェアカウンタのオーバフロープロファイル機能をオンまたは**オフ**にする。デフォルトはオフ。ハードウェアカウンタのオーバフロープロファイル機能をサポートしていないシステム上でこの機能をオンにしようとすると、dbx がエラーメッセージを返す
`hwprofile list`	利用可能なカウンタ名の一覧が返される。各カウンタには、通常間隔と高解像度用間隔の値が示される。ハードウェアカウンタのオーバフロープロファイル機能がシステム上でサポートされていない場合は、dbx がエラーメッセージを返す
`hwprofile counter` *name interval*	ハードウェアカウンタのプロファイルにイベント**名** *(name)* を、そのオーバフロー間隔に *interval* をそれぞれ設定する。*name* のデフォルトは、通常のプロファイル間隔における `cycles` である。
`status`	読み込まれた実験の状態を報告する
`show`	あらゆるコレクタ制御の現在の設定値を示す
`close`	現在の実験を閉じる
`quit`	現在の実行のデータ収集を終了する
`sample` { `periodic` | `manual` }	標本収集モードに `periodic` (定期的) または `manual`(手動) のどちらかを設定する。なお、`periodic` を指定した場合は、`sample period` 引数に値を設定しなければならない
`sample period` *value*	標本収集頻度に、ミリ秒で指定した *value* を設定する
`store directory` *directory_name*	実験レコードファイルが格納されるディレクトリに、*directory_name* を設定する
`store filename` *file_name*	出力実験ファイル名に *file_name* を設定する

利用可能な `collector` コマンド引数の一覧を取得するには、`dbx` プロンプトに対し次のように入力し、Enter キーを押します。

```
(dbx) help collector
```

実行中プロセスへの接続

標本コレクタを使用すると、実行中プロセスに接続し、そのプロセスのパフォーマンスデータを収集できます。

スレッド同期待ちの監視データを収集したい場合は、実行可能プログラムを起動する前に、`libcollector.so` というライブラリを読み込みます。これによって、実際のルーチンそのものではなく、同期ルーチンを囲むコレクタのラッパーが参照されます。プロファイルデータまたはハードウェアカウンタのオーバフロープロファイルだけを収集する場合は、コレクタライブラリを事前に読み込んでおく必要はありませんが、読み込んでも差し支えありません。

`libcollector.so` を事前にロードするには、次の操作を行います。

- **表 3-2 に示すように、環境変数 `LD_PRELOAD` が `libcollector.so` を指すように設定します。**

表 3-2 `LD_RELOAD` の設定コマンド

プラットフォーム	コマンドシーケンス
csh	`setenv LD_PRELOAD` *install_directory*`/SUNWspro/WS6/lib/dbxruntime/libcollector.so`
sh、ksh	`LD_PRELOAD=`*install_directory*`/SUNWspro/WS6/lib/dbxruntime/libcollector.so` `export LD_PRELOAD`
csh 上の SPARC-V9 実行ファイル	`setenv LD_PRELOAD` *install_directory*`/SUNWspro/WS6/lib/v9/dbxruntime/libcollector.so`
sh、ksh 上の SPARC-V9 実行ファイル	`LD_PRELOAD=`*install_directory*`/SUNWspro/WS6/lib/v9/dbxruntime/libcollector.so` `export LD_PRELOAD`

install_directory は、配布ソフトウェアが含まれているディレクトリです (通常は `/opt/`)。

注 – 実行が終了したら、LD_PRELOAD の設定値を削除し、同じシェルから起動される他のすべてのプログラムに影響が出ないようにします。

実行可能プログラムに接続し、データを収集するには、次の操作を行います。

1. **プログラムを起動します。**
2. **プログラムの** PID **を決定し、その** PID **に** dbx **を接続させます。**
   - プログラムがバックグランドで実行中の場合、その PID はシェルによって標準出力に出力されます。
   - 次のようにして、プログラムの PID を調べることができます。

```
% ps -ef | grep program_name
```

3. **データ収集機能をオンにします。**
   a. **データ収集は、**collector **コマンドを使用して** dbx **から直接起動するか、または「標本コレクタ」ウィンドウから起動します。**
   b. cont **コマンドを使用して、**dbx **から対象のプロセスを再開します。**

注 – データ収集機能をオンにせずに、dbx から実行可能プログラムを起動した場合は、dbx からターゲットプロセスを一時停止させてから、前述の命令を実行して、プロセスの実行中にデータ収集を開始することができます。

MPI を使用するプログラム

Sun Cluster Runtime Environment (CRE) コマンドである mprun を使用して、並行ジョブを起動している場合、標本コレクタは、Sun Message Passing Interface (MPI) を使用するマルチプロセスプログラムのパフォーマンスデータを収集することができます。ClusterTools 3.1 またはそれと互換性のあるバージョンを使用します。詳細については、Sun HCP ClusterTools 3.1 のマニュアルを参照してください。

MPI ジョブのデータを収集するには、dbx から MPI プロセスを 起動するか、または dbx を各プロセスに別個に接続します。たとえば、次のようなコマンドを使用して MPI ジョブを実行するとします。

```
% mprun -np 2 a.out [program-arguments]
```

このコマンドは、次のように指定することもできます。

```
% mprun -np 2 dbx a.out < collection.script
```

collection.script は dbx スクリプトで、次の段落で説明します。

この例を実行すると、実行可能オブジェクトの a.out から 2 つの MPI プロセスが実行され、test.*M*.er と test.*N*.er という 2 つの実験が作成されます。なお、*M* と *N* は、2 つの MIP プロセスの PID です。

作成される実験名が一意になるように、collection.script というファイルによって保証しなければなりません。実験名が一意でないと、実験を生成する dbx インスタンスが同時に実行されるため、複数の dbx インスタンスが同じ名前で実験を作成しようとします。ファイル名を確実に一意にする 1 つの方法は、MPI プロセスのプロセス ID が含まれたファイル名を各 dbx インスタンスが使用するように、collection.script で指定することです。

```
stop in main
run [program_arguments]
collector enable
collector store filename test.$[getpid()].er
cont
quit
```

また、名前の一部に MPI ランクが含まれた実験を作成することもできます。それには、dbx スクリプトを若干変更します。このスクリプトでは、MPI_Comm_rank () の呼び出し直後に、ターゲットプログラムを停止し、ランクを使用して実験ディレクトリを指定します。たとえば、MPI プログラムの 17 行目に次の文が記述されているものとします。

- C プログラムの場合

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

- Fortran プログラムの場合

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

collection.script を次のように変更します。

```
stop at 18
run [program_arguments]
rank=$[me]
collector enable
collector store filename test.$rank.er
cont
quit
```

このような修正を行うことによって、mprun が作成する実験には、その実験が対応する MPI プロセスのランクが含まれた名前が付けられます。

MPI プロセスから収集されたデータを調べるには、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 プロセスのデータが集計されます。詳細については、50 ページの「標本アナライザの起動および実験の読み込み」および 78 ページの「標本アナライザに実験を追加」を参照してください。

er_print を使用してもデータを出力できます。er_print は、コマンド行から複数の実験を受け付けます。詳細については、第 5 章「er_print リファレンス」参照してください。

第4章

標本アナライザリファレンス

この章では、標本アナライザおよびその使い方について、以下の項目を説明します。

- 標本アナライザの起動および実験の読み込み
- 標本アナライザウィンドウ
- 関数およびロードオブジェクトの測定結果の検査
- 関数の呼び出し元と呼び出し先の測定結果の検査
- 注釈付きソースコードおよび逆アセンブリコードの調査
- フィルタ情報
- マップファイルの作成および使用
- データオプションリストで他のデータを表示
- 標本アナライザに実験を追加
- 標本アナライザから実験を解除
- 表示の印刷

標本アナライザは、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。標本アナライザを使用すると、コレクタの収集した実験レコードファイルを読み込んで実験データの検査および操作を行い、プログラム実行時の障害を特定してパフォーマンスを向上させることができます。

標本アナライザを使ったアプリケーションの調整例は、第 2 章を参照してください。

標本アナライザでのプログラムの実行および動作の説明は、第 6 章を参照してください。

標本アナライザの起動および実験の読み込み

注 – 実験を読み込むと、標本アナライザで以前に読み込んだデータがすべて破棄されます。ただし、記録された実験には影響しません。

標本アナライザを使うには、「標本アナライザ」ウィンドウを起動して実験レコードを読み込む必要があります。コマンド行から以下の手順を実行します。

- **以下のように入力します。ここで、*experiment_name* には、読み込む実験レコードファイルの名前を指定します。**

```
% analyzer experiment_name
```

実験名は、通常は `test.n.er` という形式です。

または、Sun WorkShop のメインウィンドウまたは 「標本コレクタ」ウィンドウから標本アナライザを起動し、実験を読み込みます。

1. Sun WorkShop **のメインウィンドウまたは 「標本コレクタ」ウィンドウの「解析」ボタンをクリックします。**

2. **「実験ファイルの読み込み」は、実験を指定しないで標本アナライザを起動した場合に自動的に表示されます。このダイアログで、読み込む実験レコードファイルを指定します。**

標本アナライザの実行中に、「実験ファイルの読み込み」ダイアログを使って実験を読み込むこともできます。

- **標本アナライザのメインメニューバーの「実験ファイル」メニューから「読み込み」を選択し、「実験ファイルの読み込み」ダイアログを表示します。**

標本アナライザのコマンド行オプション

標本アナライザをコマンド行から起動する場合は、表 4-1 に示す 2 つのオプションを指定することができます。

表 4-1　標本アナライザのコマンド行オプション

`-s` *session_name*	標本アナライザのインスタンスを複数実行している場合は、1 つのテキストエディタでまとめて処理せずに、インスタンスごとにテキストエディタを起動します。これにより、異なる 2 つの実験の注釈付きソースコードまたは逆アセンブリコードを比較するなどの作業が可能です。
`-V` *experiment_file*	標本アナライザのバージョン番号を標準出力に出力します。

標本アナライザの終了

標本アナライザを終了するには、以下の操作を行います。

- **標本アナライザのメインメニューバーの「実験ファイル」メニューから「終了」を選択します。**

標本アナライザウィンドウ

「アナライザ」ウィンドウは、標本アナライザの起動時に表示されるメインの表示です。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、実験データの表示される中央の表示区画があります。

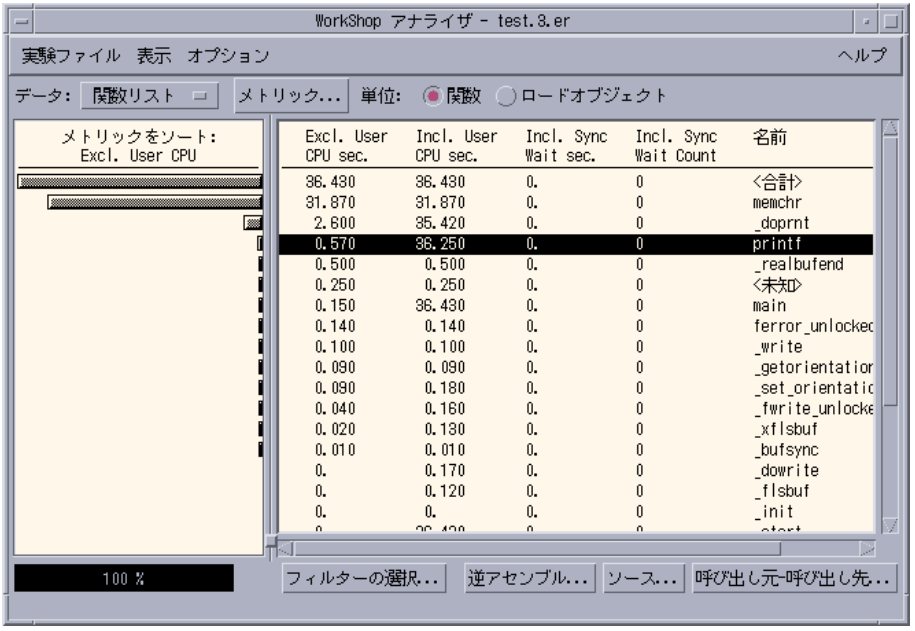

図 4-1　標本アナライザウィンドウ

関数およびロードオブジェクトの測定結果の検査

標本アナライザの起動時には、デフォルトで関数リストが表示され、関数およびロードオブジェクト固有の測定結果 (メトリック) が表示されます。関数リストは、2 つの区画で構成されています。

- 左の区画には、ソート済みの測定結果データのヒストグラムが表示されます。
- 右の区画には、関数またはロードオブジェクトの測定結果の表が表示されます。データの行に該当する関数またはロードオブジェクトの名前は、各行の右側に表示されます。

関数リストでは、表示された関数またはロードオブジェクトの測定結果ごとに、秒数または回数の絶対値と、プログラムの測定結果全体での割合が表示されます。

関数およびロードオブジェクトの測定結果の表示

デフォルトでは、関数リストには関数の測定結果が表示されます。

ロードオブジェクトの測定結果に切り替えるには、以下の操作を行います。

- **上部のツールバーの「単位」で「ロードオブジェクト」を選択します。**

関数の測定結果に戻すには、以下の操作を行います。

- **上部のツールバーの「単位」で「関数」を選択します。**

表示される測定結果について

関数リストでは、以下の種類の関数およびロードオブジェクトの測定結果 (メトリック) を表示することができます。

- 時間ベースのプロファイル
- スレッド同期待ちの監視
- ハードウェアカウンタのオーバーフロープロファイル

各測定結果の詳細は、34 ページの「標本コレクタが収集するデータ」を参照してください。

デフォルトでは、サポートデータが収集されていれば、関数リストに以下の測定結果が表示されます。

- 排他的ユーザー CPU 時間
- 包含的ユーザー CPU 時間
- 排他的スレッド同期待ち時間 (記録されている場合)
- 包含的スレッド同期待ち回数 (記録されている場合)
- 排他的ハードウェアカウンタのオーバーフロープロファイル回数 (記録されている場合)
- 包含的ハードウェアカウンタのオーバーフロープロファイル回数 (記録されている場合)

測定結果は、排他的 CPU 時間が記録されていれば、それを基準にしてソートされます。

「メトリックの選択」ダイアログでは、他のデータを選択して表示する、および異なるソート順を指定することができます。手順は、55 ページの「関数およびロードオブジェクトの測定値とソート順の選択」を参照してください。

時間ベースのプロファイル

時間ベースのプロファイルは、時計時間が基準になります。このプロファイルは、プログラムで各関数およびロードオブジェクトに消費された排他的時間と包含的時間を示します。このプロファイルを利用することで、プログラムの障害の発生場所を特定しやすくなります (排他的および包含的測定結果の詳細は、34 ページの「排他メトリック、包含メトリック、寄与メトリック」を参照してください)。

時間ベースのプロファイルデータは、プログラムの関数ごとに、以下の排他的時間のメトリックを示します。

- **ユーザー** CPU **時間** - アプリケーションが CPU 上で動作している時間
- **総** LWP **時間** - 全 LWP (軽量プロセス) の実行時間の合計
- **時計時間** - スレッド 1 で経過した LWP 時間
- **システム** CPU **時間** - オペレーティングシステムが費やした総 CPU 時間 (LWP がトラップ状態にある総 CPU 時間)
- **システム待ち時間** - CPU、ロックまたはカーネルページ待ちで経過した LWP 時間 (休眠や停止に費やされた時間)
- **テキストページフォルト時間** - テキストページ待ちで経過した LWP 時間
- **データページフォルト時間** - データページ待ちで経過した LWP 時間

これらの値は、秒数またはプログラムの合計測定結果での割合として表示します。時計時間を除き、測定結果はすべての LWP での合計になります。

スレッド同期待ちの監視

マルチスレッドのプログラムでは、スレッド同期待ちの監視によって、スレッドライブラリのスレッド同期ルーチンを呼び出すときの待ち時間が記録されます。実時間の遅延がユーザー定義しきい値を超える場合は、その呼び出しに対応するイベントが記録され、待ち時間が秒数で記録されます。

同期待ちの監視は、関数またはロードオブジェクトごとに、記録されたイベント数と、スレッド同期ルーチン呼び出し時の待ち時間でしきい値を超えたものの合計秒数のデータをサポートします。このデータから、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、あるいは同期ルーチンを呼び出すときの待ち時間が異常に長くなっているかどうかを確認することができます。

長い同期待ち時間は、スレッド間に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、特にスレッドごとにロックの必要のあるデータだけがロックされるように、ロックを再構築することで、競合を削減することができます。

ハードウェアカウンタのオーバーフロープロファイル

ハードウェアカウンタのオーバーフロープロファイルは、LWP の実行されている CPU のハードウェアカウンタでオーバーフローが発生したときに、各 LWP の呼び出しスタックを記録します。時刻、スレッド ID、LWP が記録されます。

通常は、ハードウェアカウンタのオーバーフロープロファイルは、命令キャッシュの消失、データキャッシュの消失、サイクル、発生または実行された命令のデータを記録します。

キャッシュ消失数が多い場合は、再構築によって局所性を向上して、再利用を増加させることで、プログラムのパフォーマンスが向上します。

サイクル数が多い場合は、通常は時間ベースのプロファイルも高くなります。ただし、サイクル実験によって、時間との相関が小さくなります。

関数およびロードオブジェクトの測定値とソート順の選択

プログラムのパフォーマンスが特定の障害によって低下していると推測される場合は、その障害を示す測定結果だけを関数リストで表示することができます。

関数リストで表示するデータの種類およびソート順を変更するには、以下の手順で行います。

1. **関数リストの上部ツールバーにある「メトリック」ボタンをクリックします。**

   「メトリックの選択」ダイアログが表示されます。

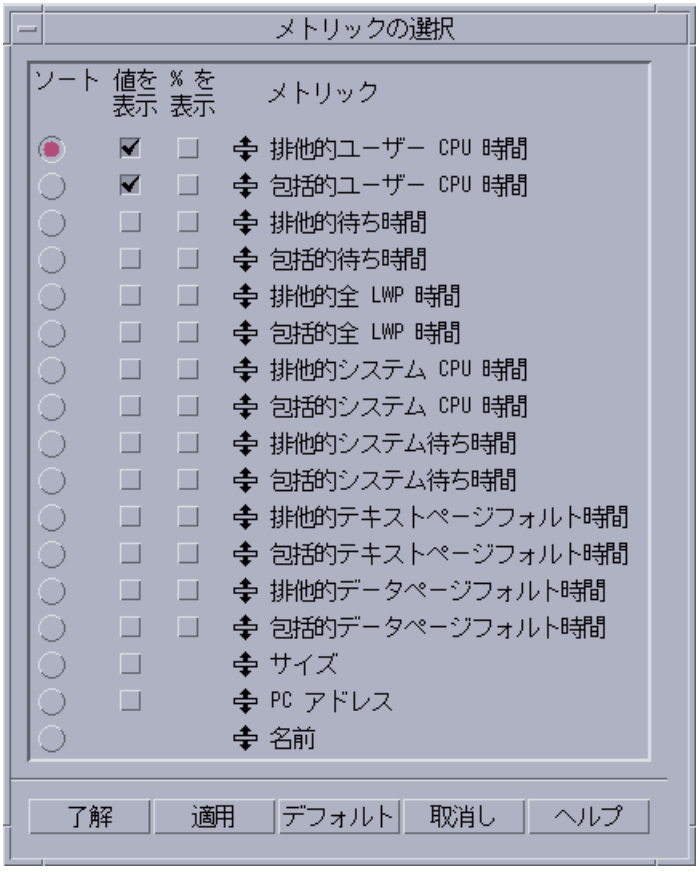

図 4-2　「メトリックの選択」ダイアログ

「メトリックの選択」ダイアログで表示されるデータの種類は、標本コレクタで収集したデータによって異なります。コレクタの実行時にすべての種類のデータを収集した場合は、以下の (排他的と包含的の) 測定結果がダイアログに表示されます。

- ユーザー CPU 時間
- 総 LWP 時間
- 時計時間 (スレッド 1 での LWP 時間)
- システム CPU 時間
- システム待ち時間
- テキストページフォルト時間
- データページフォルト時間
- 同期待ち回数 (記録されている場合)
- 同期待ち時間 (記録されている場合)

- ハードウェアカウンタのオーバーフロープロファイル回数 (記録されている場合)

これらのデータは、絶対値 (秒数または回数) およびプログラムの測定結果全体での割合で表示されます。

また、関数またはロードオブジェクトに関して以下のデータ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のアドレス

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。

2. **特定の種類の測定結果を表示するには、「メトリックの選択」ダイアログの「値」または「%」の列にあるチェックボックスを選択します。**
3. **ソート順を指定するには、「メトリックの選択」ダイアログの「ソート」列にあるボタンで選択します。**
4. **選択した測定結果を関数リストで表示するには、「了解」をクリックして「メトリックの選択」ダイアログを閉じるか、「適用」をクリックしてダイアログを表示したままで選択を有効にします。**

注 – 「メトリックの選択」ダイアログで測定結果のグループを変更するには、測定結果名の横にあるアイコンをクリックし、表示する場所までドラッグします。

関数またはロードオブジェクトの測定結果の要約表示

「概要メトリック」ウィンドウを使って、選択した関数またはオブジェクトの測定結果の合計およびその他の情報を、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。

関数またはロードオブジェクトの測定結果の要約を表示するには、以下の操作を行います。

1. **「単位」のラジオボタンで、「関数」と「ロードオブジェクト」のどちらを表示するかを選択します。**
2. **関数リストの右側の区画で、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」を選択して、「概要メトリック」ウィンドウを表示します。**

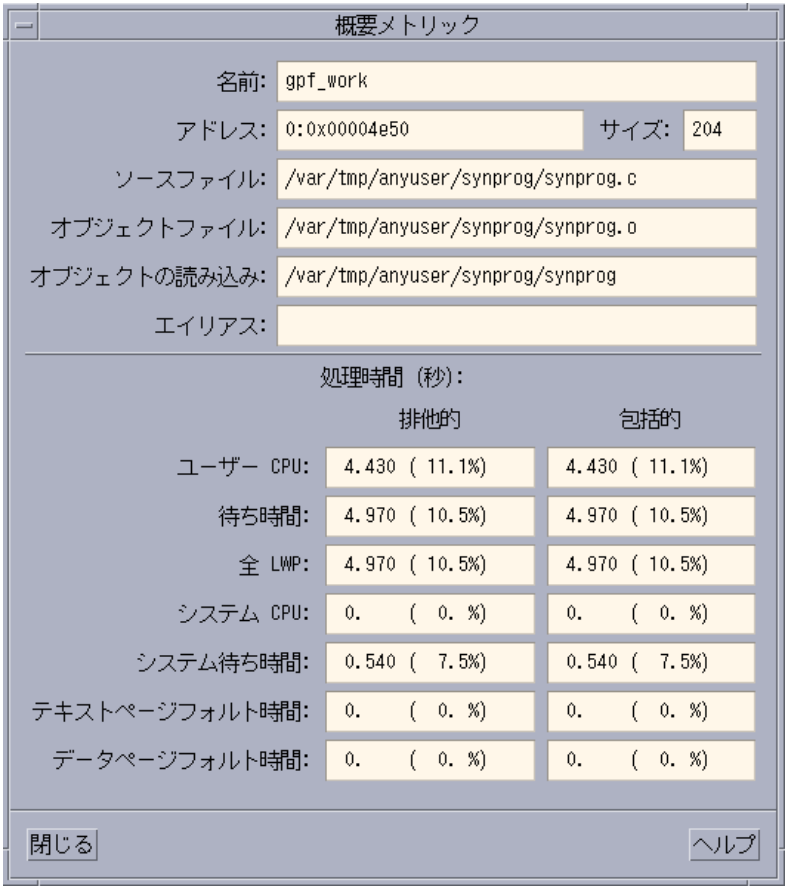

図 4-3　「概要メトリック」ウィンドウ

「概要メトリック」ウィンドウには、選択した関数またはオブジェクトに関して、以下のような排他的および包含的情報のうち 標本コレクタで収集されたものが表示されます。

- メモリーアドレス
- 関数サイズ (バイト単位)
- ユーザー CPU 時間
- 総 LWP 時間
- 時計時間 (スレッド 1 での LWP 時間)
- システム CPU 時間

- システム待ち時間
- テキストページフォルト時間
- データページフォルト時間
- 同期待ち時間 (記録されている場合)
- 同期待ち回数 (記録されている場合)
- ハードウェアカウンタのオーバーフロープロファイル回数 (記録されている場合)

注 – 「概要メトリック」ウィンドウのすべてのデータは、クリップボードにコピーし、任意のテキストエディタでペーストすることができます。

また、関数に対しては、関数のコードが含まれるソースファイル、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。

注 – 測定結果は、関数リストに表示していなくても、「概要メトリック」ウィンドウで表示することができます。「メトリックの選択」ダイアログを使って関数リストを変更しなくても、「概要メトリック」ウィンドウで表示可能なすべての関数データを表示することができます。

関数またはロードオブジェクトの検索

標本アナライザには、関数リストで関数またはロードオブジェクトの指定に使う検索ツールがあります。

特定の関数またはロードオブジェクトを検索するには、以下の手順を行います。

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

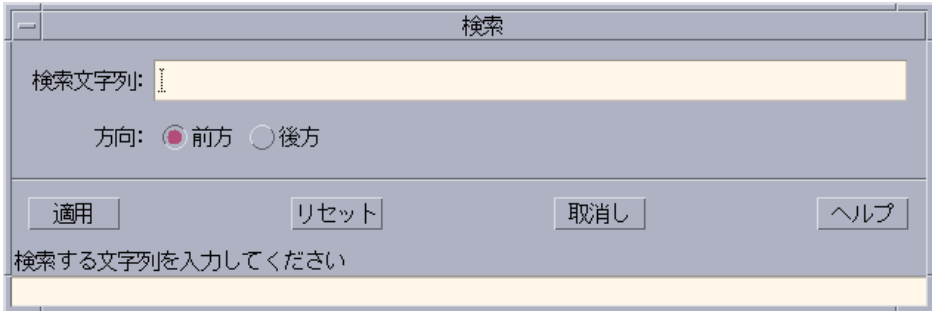

図 4-4　「検索」ダイアログ

2. **「検索文字列」フィールドに検索文字列を入力します。**

3. **「方向」ラジオボタンを使って、検索方向を指定することができます。デフォルトでは「前方」が選択されています。**

4. **「適用」をクリックします。**

   検索に成功すると、検索した関数のデータの行が関数リストで強調表示されます。

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。**

6. **最後に成功した検索の検索文字列を「検索文字列」フィールドに再度入力するには、「リセット」をクリックします。**

注 – 標本アナライザの検索機能は、UNIX の正規表現を使います。したがって、`c*` という検索文字列は、`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 の正規表現の詳細は、`regexp`(5) のマニュアルページを参照してください。

関数の呼び出し元と呼び出し先の測定結果の検査

標本アナライザの「呼び出し元-呼び出し先」ウィンドウで選択した関数の、呼び出し元と呼び出し先の測定結果を検査することができます。「呼び出し元-呼び出し先」ウィンドウを表示するには、次の操作を行います。

- **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 4-5 「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に、関数の呼び出し元の情報が上の区画に、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。

- 左の区画には、ソート済みの測定結果データのヒストグラムが表示されます。
- 右の区画には、関数の測定結果の表が表示されます。データの行に該当する関数の名前は、表の各行の右側に表示されます。

「呼び出し元-呼び出し先」ウィンドウでは、選択した関数、その関数を呼び出す関数、その関数が呼び出す関数に関する以下の情報のうち、標本コレクタで収集されたものが表示されます。

- ユーザー CPU 時間
- 総 LWP 時間
- 時計時間 (スレッド 1 での LWP 時間)
- システム CPU 時間
- システム待ち時間
- テキストページフォルト時間
- データページフォルト時間
- 同期待ち回数 (記録されている場合)
- 同期待ち時間 (記録されている場合)
- ハードウェアカウンタのオーバーフロープロファイル回数 (記録されている場合)

これらのデータは、絶対値 (秒数または回数) およびプログラムの測定結果全体での割合で表示されます。

デフォルトでは、「呼び出し元-呼び出し先」ウィンドウには以下の測定結果が表示されます。

- 属性 (寄与)、排他的、包含的の各ユーザー CPU 時間
- 属性および包含的の同期待ち回数
- 属性および包含的の同期待ち時間 (記録されている場合)
- 属性ハードウェアカウンタのオーバーフロー回数 (記録されている場合)

測定結果は、属性ユーザー CPU 時間を基準にしてソートされます。

呼び出し元区画または呼び出し先区画で関数をクリックして、プログラム構造を調べることができます。関数をクリックすると、その関数が中央に表示されます。排他的、包含的、属性時間を確認することで、実行時間が長い関数を特定することができます。

「呼び出し元-呼び出し先」ウィンドウでの測定結果およびソート順の選択

「呼び出し元-呼び出し先メトリックの選択」ダイアログで、「呼び出し元-呼び出し先」ウィンドウに表示するデータおよびソート順を指定することができます。

「呼び出し元-呼び出し先メトリックの選択」ダイアログを表示するには、次の操作を行います。

- **「呼び出し元-呼び出し先」ウィンドウで、「メトリック」ボタンをクリックします。**

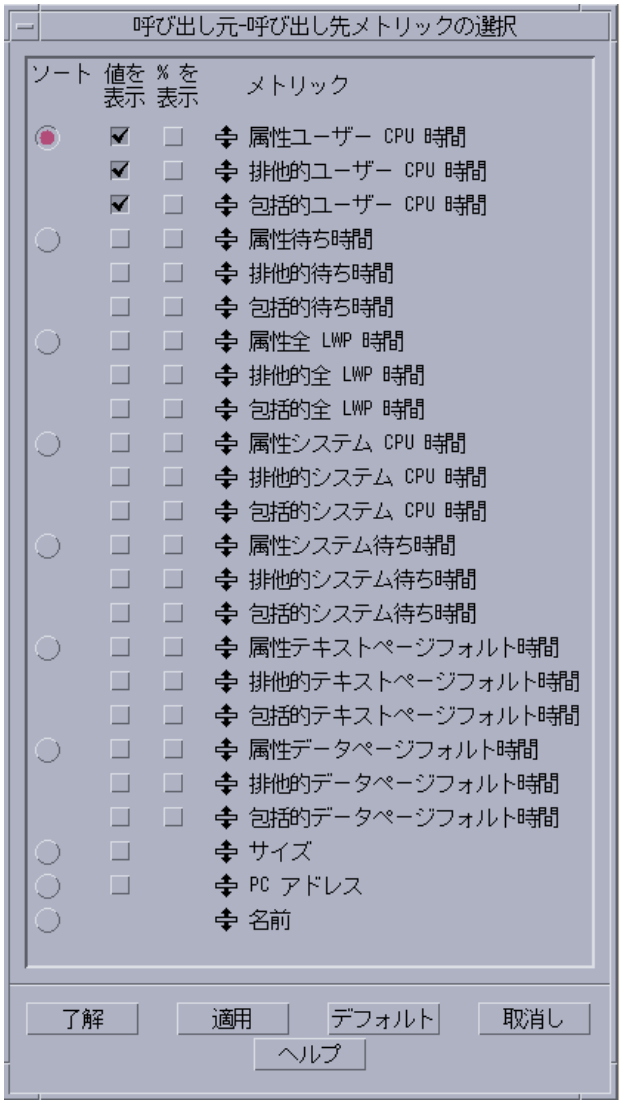

図 4-6　「呼び出し元-呼び出し先メトリックの選択」ダイアログ

「呼び出し元-呼び出し先メトリックの選択」ダイアログの操作は、「メトリックの選択」ダイアログと同様です。ただし、属性、排他的、包含的データの測定結果では、属性データを基準にしてのみソートすることができます (55 ページの「関数およびロードオブジェクトの測定値とソート順の選択」を参照してください)。

注 – 「呼び出し元-呼び出し先メトリックの選択」ダイアログで測定結果のグループを変更するには、測定結果名の横にあるアイコンをクリックし、表示する場所までドラッグします。

注釈付きソースコードおよび逆アセンブリコードの調査

プログラムの実行速度低下の原因である関数を特定したら、その問題となる個所のソースコードまたは逆アセンブリをパフォーマンス測定結果の注釈付きで表示し、障害の原因になっている実際の行または命令を特定することができます。

関数の注釈付きソースコードを表示するには、以下の操作を行います。

1. **関数リストの右の区画で、関数をクリックして選択します。**
2. **「アナライザ」ウィンドウの下部にあるツールバーの「ソース」をクリックします。**

   テキストエディタが起動し、選択した関数のソースコードが表示されます。ソースコードの各行のパフォーマンス測定結果が、コードの左側に表示されます。

表 4-2 に、注釈付きソースコードの行で表示可能な 4 種類の測定結果を示します。

表 4-2　注釈付きソースコードの測定結果

測定結果	意味
(空白)	この行に対応する PC がプログラムに存在しません。コメント行では常にこの結果になります。また、以下の場合のコード行でも同様の結果になります。 • コード行の命令がすべて最適化され取り去られた場合 • 別の場所とコードが一致していて、コンパイラが共通部分式を認識し、その行の全部の命令を別の行に結びつけた場合 • コンパイラがその行の命令に間違った行番号結びつけた場合
`0.`	この行に対応する 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。統計的に標本採取された呼び出しスタックまたはスレッド同期データのために追跡された呼び出しスタックに、この PC が存在しないことを示します。測定結果が `0.` の場合は、行が実行されなかったことを示すのではなく、統計上プロファイルに表れないこと、および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示します。
`0.000`	この行の PC の少なくとも 1 つがデータに表れているが、測定結果の合計値が丸められてゼロになったことを示します。
`1.234`	この行のすべての PC の測定結果の合計がゼロ以外になり、数値として表示されています。

関数の注釈付き逆アセンブリコードを生成するには、以下の操作を行います。

1. **関数リストの右の区画で、関数をクリックして選択します。**
2. **「アナライザ」ウィンドウの下部にある「逆アセンブル」をクリックします。**

テキストエディタが起動し、選択した関数のコードが表示されます。ソースコードの各行のパフォーマンス測定結果が、コードの左側に表示されます。

注 – 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、ソースコードに逆アセンブリコードが挿入されて表示されます。ソースコードが存在しない場合でも、逆アセンブリコードを調べることができます。

注釈での測定結果の種類は、ソースコードまたは逆アセンブリコードを呼び出したときに関数リストで表示されるものと同じです。測定結果を変更するには、「メトリックの選択」ダイアログで関数リストの測定結果を変更してから、注釈付きソースコードまたは逆アセンブリコードを表示します。

テキストエディタの選択

注釈付きソースコードおよび逆アセンブリコードは、テキストエディタで表示されます。そのため、コードを編集して問題を修正することができます。テキストエディタは、使用したいものを選択できます。

テキストエディタを選択するには、以下の操作を行います。

1. **関数リストで「オプション」メニューから「テキストエディタオプション」を選択して、「テキストエディタのオプション」ダイアログを表示します。**
2. **「使用するエディタ」リストボックスで、テキストエディタを選択します。**

   NEdit、Vi、GNU Emacs、XEmacs、gvim のいずれかを選択することができます。

注 – テキストエディタによっては、一部のロケールに対応していません。

フィルタ情報

プログラム中で障害の原因となっている可能性が高い場所の情報だけを表示できれば、より効率的に情報を処理することができます。標本アナライザでは、複数の方法で実験情報にフィルタを適用することができます。

- ロードオブジェクトを基準にする
- 標本、スレッド、LWP のいずれかまたはすべてを基準にする

ロードオブジェクトの選択

パフォーマンス解析では、多くの場合は、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の測定結果だけが必要であり、システムライブラリの測定結果は不要な場合です。標本アナライザでは、「関数リスト」および「概要」で測定結果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。

情報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」メニューの「条件付きロードオブジェクトの選択」を選択して、「次の条件を含んだロードオブジェクトを選択」ダイアログを表示します。**
2. **リストボックスで、表示しないファイルをクリックして選択を解除します。**

   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」をクリックして選択を有効にし、「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。**

標本、スレッド、LWP の選択

測定結果を表示する標本、スレッド、LWP だけを指定することで、表示する情報を限定することができます。「関数リスト」および「概要」では、指定した標本、スレッド、LWP の測定結果だけが表示されます。

注 – 標本を選択すると、「概要」の右の区画でその標本に影が表示されます。「概要」の使い方については、72 ページの「標本の概要の検査」を参照してください。

標本、スレッド、LWP は、個別、範囲、グループ単位で、任意の順序で選択することができます。

- **下部のツールバーの「フィルタの選択」ボタンをクリックします。**

  「フィルタを選択」ダイアログが表示されます。このダイアログには、以下のフィールドがあります。

  - 標本

- スレッド
- LWPs

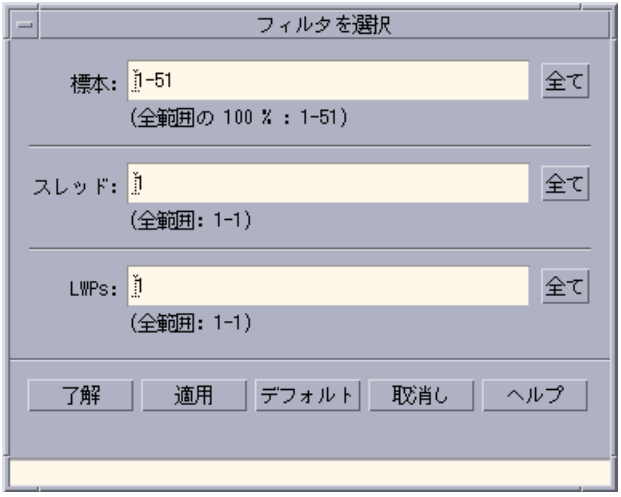

図 4-7　「フィルタを選択」ダイアログ

これらのフィールドに、データを表示する標本、スレッド、LWP を指定します。標本、スレッド、LWP は、任意の番号および組み合わせで指定することができます。

標本、スレッド、LWP を 1 つ選択するには、次の操作を行います。

- **該当するフィールドに、標本、スレッド、**LWP **のいずれかの** ID **番号を入力し、**Enter **を押します。**

標本、スレッド、LWP を範囲を指定して選択するには、次の操作を行います。

- **該当するフィールドに、範囲の最初と最後の** ID **番号をハイフンで区切って (**`5-12` **など) 入力し、**Enter **を押します。**

連続していない複数の標本、スレッド、LWP を選択するには、次の操作を行います。

- **該当するフィールドに、複数の** ID **をコンマで区切って (**`3,7,15,21` **など) 入力し、**Enter **を押します。**

実験レコードのすべての標本、スレッド、LWP を選択するには、次の操作を行います。

- **標本、スレッド、**LWPs **の「全て」ボタンをクリックします。**

マップファイルの作成および使用

標本アナライザでは、実験レコードのデータを使ってマップファイルを作成することができます。このマップファイルを静的リンカーで使用して、作成する実行可能ファイルのワーキングセットの縮小や命令キャッシュの動作の効率化を図ることができます。

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. **-xF オプションを指定してプログラムをコンパイルします。このオプションは、独立して再配置可能な関数を生成するように指定します。以下に例を示します。**

   C アプリケーションの場合は、以下のように入力します。

```
% cc -xF -c a.c b.c
% cc -o application_name a.o b.o
```

   C++ アプリケーションの場合は、以下のように入力します。

```
% CC -xF -c a.cc b.cc
% CC -o application_name a.o b.o
```

   Fortran アプリケーションの場合は、以下のように入力します。

```
% f95 -xF -c a.f b.f
% f95 -o application_name a.o b.o
```

   以下の警告メッセージが表示された場合は、非共有の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を調べ、これらのファイルが -xF オプションを指定してコンパイルされているかどうかを確認します。

```
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.
```

2. **アプリケーションをデバッグするため、**Sun WorkShop **に読み込み、標本コレクタを使ってパフォーマンスデータを収集します** (38 **ページの「**Sun WorkShop**でのパフォーマンスデータの収集」を参照)。アドレス空間データの収集が有効であることを確認します。**

3. **生成した実験を 標本アナライザに読み込みます** (50 **ページの「標本アナライザの起動および実験の読み込み」を参照)。**

4. **「実験ファイル」メニューの「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。**

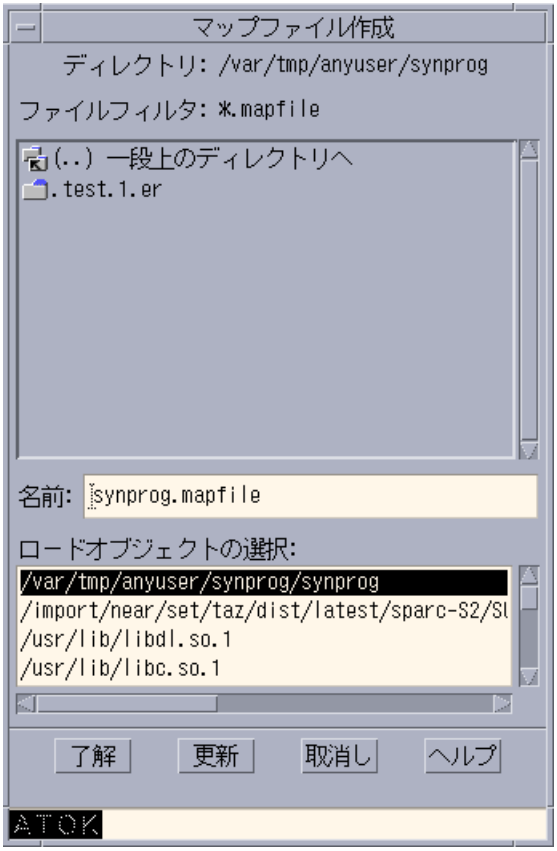

図 4-8　「マップファイル作成」ダイアログ

5. **必要に応じて、「マップファイル作成」ダイアログの「ディレクトリ」区画で、マップファイルの保存先ディレクトリを指定します。**

6. **「名前」フィールドでは、以下の操作が可能です。**

- ファイルフィルタを変更して既存のファイル名を表示し、そのファイルに上書きするように指定する
- デフォルトとは異なるマップファイルのパスおよびファイル名を入力する

7. **「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。**
8. **「了解」をクリックします。**

マップファイルを使ってプログラム内の順序を変更するには、以下のように入力します。

- **通常の手順で、マップファイルを使ってオブジェクトファイルをリンクします。以下に例を示します。**

C アプリケーションの場合は、以下のように入力して Enter を押します。

```
% cc -Wl -M mapfile_name a.o b.o
```

C++ アプリケーションの場合は、以下のように入力して Enter を押しします。

```
% CC -M mapfile_name a.o b.o
```

Fortran アプリケーションの場合は、以下のように入力して Enter を押しします。

```
% F90 -M mapfile_name a.o b.o
```

データオプションリストで他のデータを表示

標本アナライザウィンドウを表示して実験を読み込むと、デフォルトでは関数およびロードオブジェクトの情報を示す関数リストが表示されます。関数リストの詳細は、52 ページの「関数およびロードオブジェクトの測定結果の検査」を参照してください。

上部のツールバーの「データ」リストボックスで、表示区画の内容を変更し、他の種類のデータを表示することができます。

- **「概要」** - 高レベルの標本情報です。詳細は、72 ページの「標本の概要の検査」を参照してください。
- **「アドレス空間」** - プログラムの使うメモリーに関する情報です。詳細は、75 ページの「アドレス空間情報の検査」を参照してください。
- 「実行統計」 - プログラムの実行結果に関する全般的な情報です。詳細は、77 ページの「実行の統計情報の検査」を参照してください。

データオプションリストで「関数リスト」を選択すると、他の表示から関数リストに戻ることができます。

注 – 標本アナライザで情報を調べるには、先に標本コレクタで情報を収集して実験レコードに保存する必要があります。標本コレクタで収集して実験レコードに保存するデータの指定方法については、38 ページの「Sun WorkShopでのパフォーマンスデータの収集」を参照してください。

標本の概要の検査

概要を表示するには、以下の操作を行います。

- **「データ」リストボックスで「概要」を選択します。**

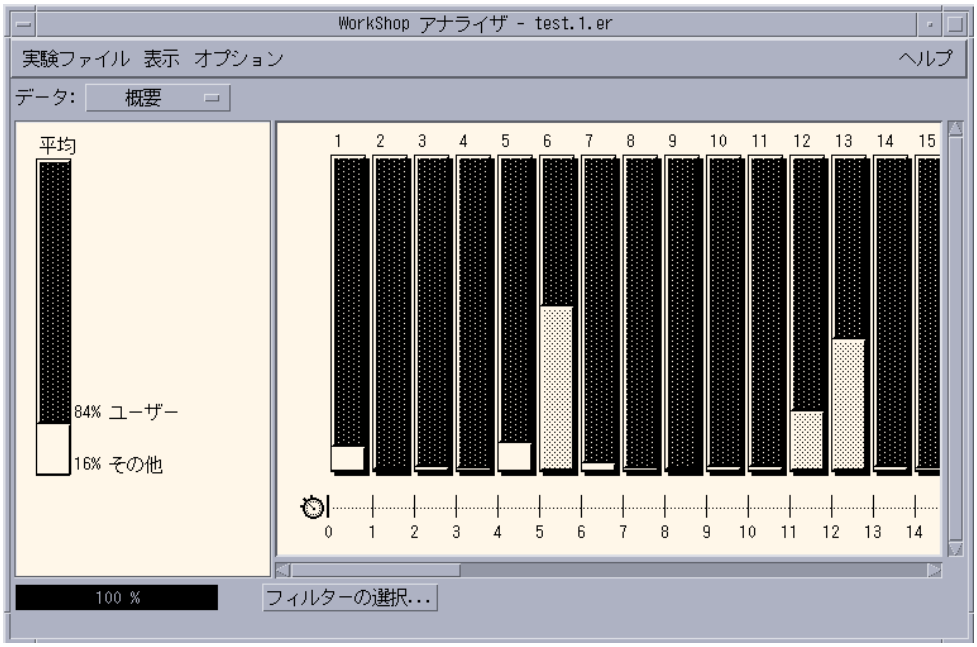

図 4-9 概要表示

概要では、プログラムの実行の一部または全体のプロセス時間に関する情報が表示されます。概要は、2 つの区画で構成されています。

- 左の区画には、選択した標本 (1 つまたは複数) のさまざまなプロセス状態で消費された時間の平均を示すグラフが表示されます。
- 右の区画には、選択した標本のさまざまなプロセス状態で消費された時間を示すグラフが連続して表示されます。各グラフは、1 回のサンプリング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、標本の識別番号が表示されます。

可変幅と固定幅での表示

デフォルトでは、概要は標本は固定幅で表示されます。つまり、標本のグラフは、サンプリング間隔が同じかどうかに関係なく、すべて同一幅で表示されます。標本は、サンプリング間隔に応じて可変幅で表示することもできます。

可変幅表示に変更するには、以下の操作を行います。

- **「オプション」メニューの「概要のカラムサイズを指定」の「比例」を選択します。**

固定幅表示に戻すには、以下の操作を行います。

- **「オプション」メニューの「概要のカラムサイズを指定」の「固定」を選択します。**

標本の詳細情報の表示

標本の詳細情報を表示するには、先に標本を選択する必要があります。選択する手順は、67 ページの「標本、スレッド、LWP の選択」を参照してください。

概要の左の区画には、選択したすべての標本に関して、さまざまなプロセス状態で消費された時間の平均と、各状態の示すサンプリング時間の割合が表示されます。たとえば、ある標本のセットの実行時間のうち、ユーザーコードの実行が 23%、システム待ち時間が 50%、値が小さいためにグラフでは個々に表示されない他の状態が 27% というように表示されます。

標本グラフでは値が小さすぎて表示されないプロセス状態の測定結果など、標本情報のより詳細な分析を表示するには、以下の操作を行います。

- **標本アナライザで、「表示」メニューの「標本の詳細を表示」を選択します。**

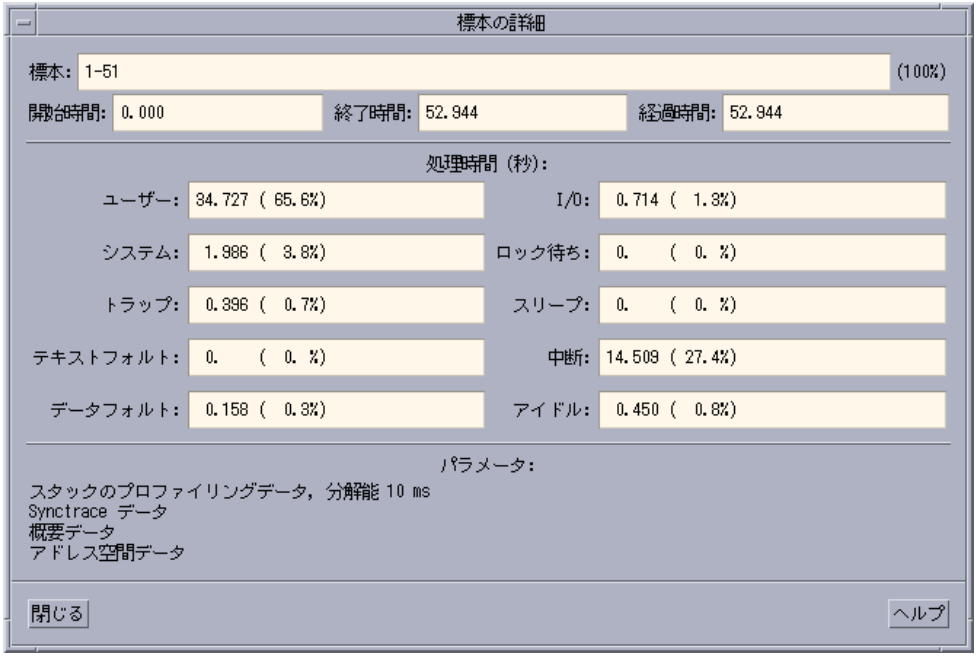

図 4-10 「標本の詳細」ウィンドウ

以下の測定結果を示す「標本の詳細」ウィンドウが表示されます。

- 標本の ID

- 選択した標本の割合
- サンプリングの開始時間、終了時間、長さ (秒単位)
- プロセス状態と各状態で消費された時間 (秒数と、選択した標本の測定結果の合計の割合) のリスト
  - ユーザー
  - システム
  - トラップ
  - テキストフォルト
  - データフォルト
  - I/O
  - ロック待ち
  - スリープ
  - 中断
  - アイドル
- 標本コレクタが実験レコードファイルに記録したデータの種類を示すパラメータのリスト

アドレス空間情報の検査

注 – 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ていた場合にだけ、標本アナライザで表示されます。選択していなかった場合は、アドレス空間は表示されません。

アドレス空間を表示するには、以下の操作を行います。

- **「データ」リストボックスで「アドレス空間」を選択します。**

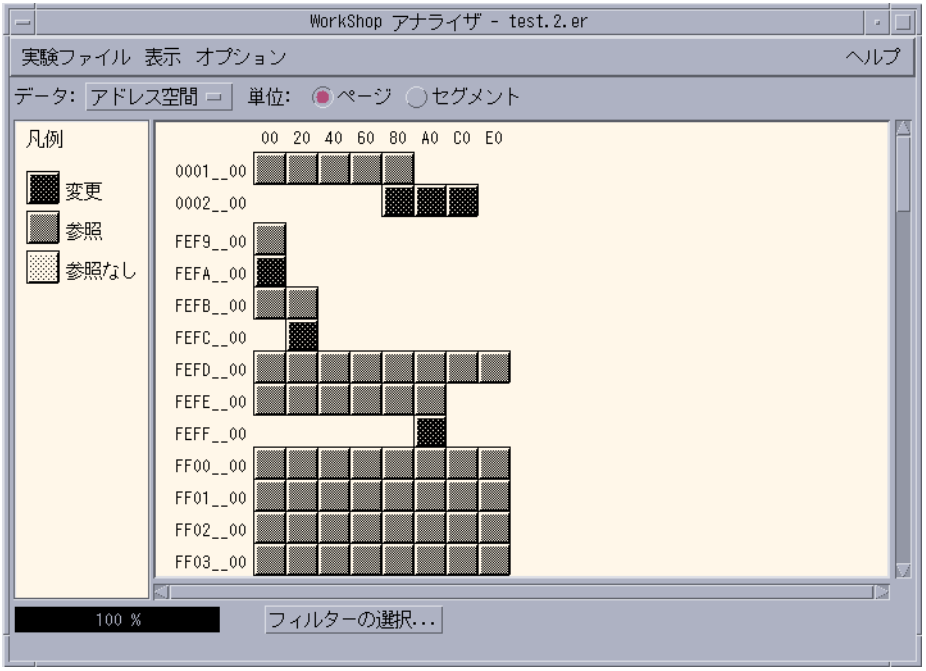

図 4-11 アドレス空間表示

アドレス空間は、2 つの区画で構成されています。

- 左の区画には、右の区画の表示の説明が表示されます。
- 右の区画には、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」ラジオボタンの「ページ」を選択しても、この表示になります)。各マス目は、アドレス空間のページを示します。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。

- ページを変更 (書き込み)
- ページを参照 (読み取り)
- 参照なし

アドレス空間のセグメントを表示するには、以下の操作を行います。

- **上部のツールバーの「単位」ラジオボタンで、「セグメント」をクリックします。**

  右の区画に、プログラムで使われ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。

ページおよびセグメントの詳細を表示する

ページまたはセグメントの詳細を表示するには、以下の操作を行います。

1. **「単位」ラジオボタンで、「ページ」または「セグメント」を選択します。**
2. **アドレス空間の右の区画で、ページまたはセグメントをクリックして選択します。**

   アドレス空間の右の区画で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に影が表示されます。
3. **標本アナライザで、「表示」メニューの「ページ属性を表示」または「セグメント属性を表示」を選択します。**

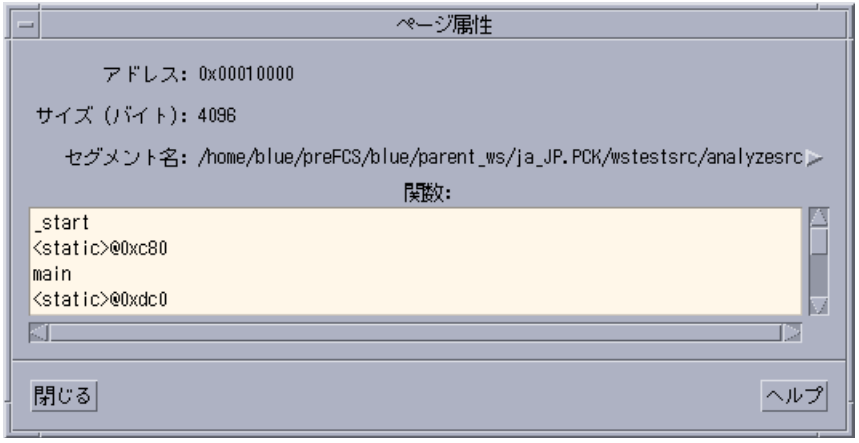

図 4-12　「ページ属性」ウィンドウ

以下の情報を示す「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)
- ページまたはセグメントに格納されている関数があれば、その関数のリスト

実行の統計情報の検査

実行の統計情報を調べるには、以下の操作を行います。

「データ」リストボックスで「実行統計」を選択します。

実行統計には、選択した標本のさまざまなシステム統計情報の合計が表示されます (標本の選択およびグループ化の方法については、67 ページの「標本、スレッド、LWP の選択」を参照)。

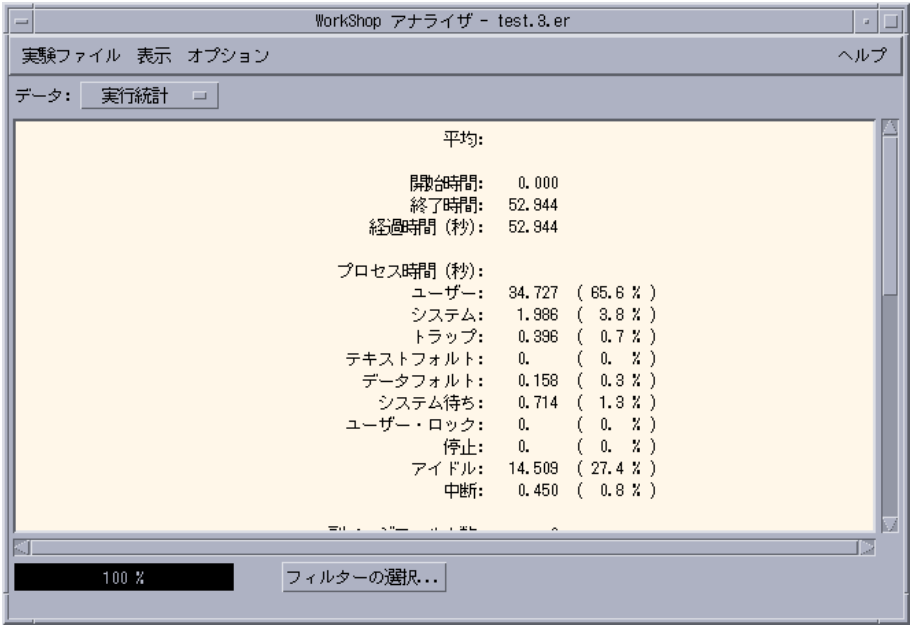

図 4-13 実行統計表示

注 – 「実行統計」ウィンドウのすべてのデータは、クリップボードにコピーし、任意のテキストエディタにペーストすることができます。

標本アナライザに実験を追加

標本アナライザでは、複数の実験レコードファイルを読み込むことができます。ただし、複数の実験レコードを読み込んだ場合は、以下の制限があります。

- すべての実験の時間ベースのプロファイル、スレッド同期待ちの監視、ハードウェアカウンタのオーバーフロープロファイルの結合データは、関数リストで結合して表示されます。すべての実験レコードのすべての標本データが表示されます。
- 標本、スレッド、LWP を基準にしたフィルタは無効になります。

- 関数リストだけを使用することができます。

標本アナライザに読み込み済みのレコードに実験レコードを追加するには、以下の操作を行います。

1. **標本アナライザのメニューバーの「実験ファイル」メニューから「追加」を選択して、「実験ファイルの追加」ダイアログを表示します。**
2. **追加する実験レコードファイルをリストボックスでダブルクリックするか、「名前」フィールドに実験レコードファイル名を入力します。**
3. **「了解」をクリックします。**

注 – 「実験ファイル」メニューの「追加」コマンドは、関数リストでのみ使用可能です。

標本アナライザから実験を解除

注 – 実験を解除すると、標本アナライザから削除されますが、実験レコードファイルには影響しません。標本アナライザで実験レコードファイルを直接削除することはできません。

標本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」メニューの「解除」を選択して、「実験ファイルの解除」ダイアログを表示します。**
2. **標本アナライザから解除する実験ファイルをリストボックスでクリックします。**
3. **ダイアログを表示したまま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。**

注 – 複数の実験ファイルが読み込まれている場合のみ、標本アナライザから実験を解除することができます。読み込まれている実験が 1 つだけの場合は、「解除」コマンドは使用できません。

表示の印刷

標本アナライザの表示をテキスト形式で印刷するには、以下の操作を行います。

1. **標本アナライザのメニューバーの「実験ファイル」メニューから「印刷」を選択して、「印刷」ダイアログを表示します。**
2. **「出力先」ラジオボタンで、「プリンタ」または「ファイル」を選択します。**
   - プリンタに出力する場合は、「プリンタ」フィールドのデフォルト名をそのまま使用するか、他のプリンタ名を入力します。
   - ファイルに出力する場合は、「ファイル」フィールド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、ディレクトリまたは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_print リファレンス

この章では、er_print ユーティリティの使い方について、以下の項目を説明します。

- er_print の構文
- er_print コマンド

er_print は、標本アナライザのサポートするさまざまな表示をテキスト形式で出力する Sun WorkShop のユーティリティです。情報は、ファイルまたはプリンタに出力先を変更しない限り、標準出力に出力されます。er_print の引数として、標本コレクタが生成した実験レコードファイルの名前を指定する必要があります。標本コレクタが実験レコードファイルにデータを保存していれば、呼び出し元と呼び出し先の関数のパフォーマンス測定結果 (メトリック)、ソースコードと逆アセンブリのリスト、サンプリング情報、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。

- 標本コレクタの収集するデータの詳細は、第 3 章を参照してください。
- 標本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 は、Sun WorkShop の C、C++、Fortran 77、Fortran 95 用コンパイラおよびアセンブラで使用することができます。

er_print の構文

er_print のコマンド行構文を以下に示します。

```
er_print [-script script | -command | -] exper_1 exper_2...exper_n
```

オプション

er_print では、以下のオプションを指定することができます。

-	端末で入力した er_print コマンドを読み込みます。
-script *script*	*script* で指定したファイルからコマンドを読み込みます。このファイルでは、1 行に 1 つずつ er_print コマンドを記述します (「er_print コマンド」を参照)。-script オプションが指定されていない場合は、er_print はコマンドを端末またはコマンド行から読み込みます。
-*command*	指定したコマンドを処理します。

er_print のコマンド行では、複数のオプションを指定することができます。オプションは、指定した順に処理されます。スクリプト、"-" 引数、コマンド名は、任意の順序で組み合わせて指定することができます。コマンドやスクリプトの引数を指定しない場合は、er_print は "-" 引数を指定した場合と同様に対話型モードで処理され、キーボードで入力したコマンドが読み込まれます。

er_print コマンド

er_print で指定可能なコマンドを以下に示します。コマンド名は、他のコマンドと区別できる程度に省略することができます。

関数リストコマンド

以下のコマンドは、関数の情報の表示を設定します。

functions

選択している測定結果 (メトリック) と関数を出力します。

デフォルトでは、排他的および包含的ユーザー CPU 時間、秒数およびプログラム全体に占める割合が出力されます。metrics コマンドを使って、表示する内容を変更することができます。

fsummary

関数ごとに、測定結果の要約を出力します。

関数の測定結果の要約については、57 ページの「関数またはロードオブジェクトの測定結果の要約表示」を参照してください。

`metrics` *metric_spec*

関数リストの測定結果を新しく指定します。

metric_spec には、以下のように測定結果のキーワードのリストをコロンで区切って指定します。

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

このコマンドは、`er_print` で以下を表示するように指定します。

- 包含的ユーザー CPU 時間 (秒単位)
- 包含的ユーザー CPU 時間 (割合)
- 排他的ユーザー CPU 時間 (秒単位)
- 排他的ユーザー CPU 時間 (割合)

`metrics` コマンドの実行が終了すると、新しく選択された測定結果を示す以下のようなメッセージが表示されます。

```
現在: i.user:i%user:e.user:e%user:names
```

選択している測定結果のリストを表示するには、`functions` コマンドを使います。

注 – 使用可能な `er_print` の測定結果のキーワードのリストは、89 ページの表 5-1 を参照してください。実験で使用可能なキーワードのリストを生成するには、`metric_list` または `cmetric_list` コマンドを使います。

`objects`

選択している測定結果のリストを使って、ロードオブジェクトを出力します。

デフォルトでは、排他的および包含的ユーザー CPU 時間が、秒数およびプログラム全体の測定結果に占める割合が出力されます。`metrics` コマンドを使って、表示する測定結果を変更することができます。

`osummary`

ロードオブジェクトリストのロードオブジェクトごとに、測定結果の要約を出力します。

ロードオブジェクトの測定結果の要約については、57 ページの「関数またはロードオブジェクトの測定結果の要約表示」を参照してください。

`sort` *metric_keyword*

指定した測定結果で関数リストをソートします。*metric_keyword* には、89 ページの表 5-1に示す測定結果のいずれかを以下のように指定します。

```
sort i.user
```

このコマンドは、包含的ユーザー CPU 時間で関数リストをソートします。

呼び出し元と呼び出し先の表示コマンド

以下のコマンドは、呼び出し側と呼び出し先の情報の表示を設定します。

`callers-callees`

関数ごとの「呼び出し元-呼び出し先」に、ソートして出力します。対象とされる (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    Name
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` が対象の関数です。この関数は、`_doprnt` および `_xflsbuf` によって呼び出され、`_rw_rdlock` および `_rw_unlock` を呼び出します。

`cmetrics` *metric_spec*

呼び出し元と呼び出し先の測定結果を新しく指定します。

metric_spec には、以下のように測定結果のキーワードのリストをコロンで区切って指定します。

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

このコマンドは、er_print で以下を表示するように指定します。

- 包含的ユーザー CPU 時間 (秒単位)
- 包含的ユーザー CPU 時間 (割合)
- 属性ユーザー CPU 時間 (秒単位)
- 属性ユーザー CPU 時間 (割合)

選択している測定結果のリストを表示するには、「callers-callees」コマンドを使います。

注 – 使用可能な er_print の測定結果のキーワードのリストは、89 ページの表 5-1 を参照してください。実験で使用可能なキーワードのリストを生成するには、metric_list または cmetric_list コマンドを使います。

---

csort *metric_keyword*

指定した測定結果で呼び出し元と呼び出し先のリストをソートします。*metric_keyword* には、89 ページの表 5-1 に示す測定結果のから以下のように指定します。

```
csort a.user
```

このコマンドは、属性ユーザー CPU 時間で呼び出し元と呼び出し先のリストをソートします。

ソースおよび逆アセンブリのリストコマンド

以下のコマンドは、注釈付きソースおよび逆アセンブリコードの表示を設定します。

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、指定したパスのディレクトリにファイルが存在する必要があります。

省略可能なパラメータ *N* (1 以上の整数) は、ファイルまたは関数名があいまいな場合にだけ指定します。あいまいな場合は、*N* 番目に該当するファイルまたは関数が使われます。N を指定しないであいまいな名前を指定すると、er_print はオブジェクトファイル名の候補を表示します。指定した名前が関数の場合は、関数名がオブジェクトファイル名に付けられ、*N* の値が示すそのオブジェクトファイルの番号も出力されます。

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、指定したパスのディレクトリにファイルが存在する必要があります。

省略可能なパラメータ *N* (1 以上の整数) は、ファイルまたは関数名があいまいな場合にだけ指定します。あいまいな場合は、*N* 番目に該当するファイルまたは関数が使われます。N を指定しないであいまいな名前を指定すると、er_print はオブジェクトファイル名の候補を表示します。指定した名前が関数の場合は、関数名がオブジェクトファイル名に付けられ、*N* の値が示すそのオブジェクトファイルの番号も出力されます。

標本、スレッド、LWP、ロードオブジェクトの選択コマンド

以下のコマンドは、表示する標本、スレッド、LWP を選択します。

lwp_list

解析対象として選択している LWP のリストを表示します。以下に例を示します。

```
lwp_list
現在: 1,3-9,17,20-38,40, 全体: 1-42
```

lwp_select *lwp_spec*

情報を表示する LWP を選択します。*lwp_spec* には、all (すべての LWP を選択します)、LWP の ID 番号のリスト、ID 番号の範囲 (*n*-*m*) を組み合わせて、コンマで区切って指定します (空白文字は指定できません)。以下に例を示します。

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

`object_list`

解析対象として選択しているロードオブジェクトのリストを表示します。以下に例を示します。

```
object_list
+ /home/user/a.out
+ /usr/lib/libthread.so.1
+ /usr/lib/libc.so.1
+ /usr/lib/libdl.so.1
```

`object_select` *object_spec*

情報を表示するロードオブジェクトを選択します。*object_spec* には、ロードオブジェクトのリストをコンマで区切って指定します (空白文字は指定できません)。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。

オブジェクト名には、完全パス名またはベース名を指定します。

`sample_list`

解析対象として選択している標本のリストを表示します。以下に例を示します。

```
sample_list
現在: 1,3-5,10,20-78, 全体: 1-78
```

`sample_select` *sample_spec*

情報を表示するサンプルを選択します。*sample_spec* には、`all` (すべてのサンプルを選択します)、サンプルの ID 番号のリスト、ID 番号の範囲 (*n*-*m*) を組み合わせて、コンマで区切って指定します (空白文字は指定できません)。以下に例を示します。

```
sample_select 1,3-5,10,20-78
```

`thread_list`

解析対象として選択しているスレッドのリストを表示します。以下に例を示します。

```
thread_list
現在: 1-41, 全体: 1-41
```

`thread_select` *thread_spec*

情報を表示するスレッドを選択します。*thread_spec* には、`all` (すべてのスレッドを選択します)、スレッドの ID 番号のリスト、ID 番号の範囲 (*n-m*) を組み合わせて、コンマで区切って指定します (空白文字は指定できません)。以下に例を示します。

```
thread_select all
```

測定結果のコマンド

以下のコマンドは、測定結果 (メトリック) の仕様キーワードのリストを表示します。

`metric_list`

測定結果のキーワードのリストを表示します。これらのキーワードは、他のコマンド (`metrics` や `sort` など) で指定して、関数リストのさまざまな種類の測定結果を参照することができます。

`size`、`address`、`name` 以外のキーワードは、`e` (排他的)、`i` (包含的)、`a` (属性付き) のいずれかの文字と、絶対値を示すピリオド (`.`) またはプログラムの測定結果全体での割合を示すパーセント記号 (`%`) のいずれかと、測定結果を説明する文字列で構成されます。

属性付き (寄与) 測定結果は、`cmetrics` コマンドのみで指定でき、`metrics` コマンドでは指定できません。また、`callers-callees` コマンドのみで表示でき、`functions` コマンドでは表示できません。

表 5-1 に、使用可能な er_print の測定結果のキーワードを示します。

表 5-1 測定結果の仕様キーワード

絶対値	割合	説明
e.user	e%user	排他的ユーザー CPU 時間
i.user	i%user	包含的ユーザー CPU 時間
a.user	a%user	属性ユーザー CPU 時間
e.wall	e%wall	排他的待ち時間
i.wall	i%wall	包含的待ち時間
a.wall	a%wall	属性付待ち時間
e.total	e%total	排他的合計 LWP 時間
i.total	i%total	包含的合計 LWP時間
a.total	a%total	属性合計 LWP 時間
e.system	e%system	排他的システム CPU 時間
i.system	i%system	包含的システム CPU 時間
a.system	a%system	属性システム CPU 時間
e.wait	e%wait	排他的システム待ち時間
i.wait	i%wait	包含的システム待ち時間
a.wait	a%wait	属性システム待ち時間
e.text	e%text	排他的テキストページフォルト時間
i.text	i%text	包含的テキストページフォルト時間
a.text	a%text	属性テキストページフォルト時間
e.data	e%data	排他的データページフォルト時間
i.data	i%data	包含的データページフォルト時間
a.data	a%data	属性データページフォルト時間
e.sync	e%sync	排他的スレッド同期待ち時間
i.sync	i%sync	包含的スレッド同期待ち時間
a.sync	a%sync	属性スレッド同期待ち時間
e.syncn	e%syncn	排他的スレッド同期待ち回数
i.syncn	i%syncn	包含的スレッド同期待ち回数
a.syncn	a%syncn	属性スレッド同期待ち回数

表 5-1 測定結果の仕様キーワード (続き)

絶対値	割合	説明
size		関数のサイズ (バイト単位)
address		メモリー上の関数のアドレス
name		関数名

`cmetric_list`

測定結果のキーワードのリストを表示します。これらのキーワードは、他のコマンド (`metrics` や `sort` など) で指定して、呼び出し元と呼び出し先のリストのさまざまな種類の測定結果を参照することができます。

`size`、`address`、`name` 以外のキーワードは、`e` (排他的)、`i` (包含的)、`a` (属性付き) のいずれかの文字と、絶対値を示すピリオド (`.`) またはプログラム全体に占める割合を示すパーセント記号 (`%`) のいずれかと、測定結果を説明する文字列で構成されます。

注 – 属性付き測定結果は、`cmetrics` コマンドのみで指定でき、`metrics` コマンドでは指定できません。また、`callers-callees` コマンドのみで表示でき、`functions` コマンドでは表示できません。

---

89 ページの表 5-1 に、使用可能な `er_print` の測定結果のキーワードを示します。

出力コマンド

以下のコマンドは、`er_print` の出力を設定します。

`limit` *n*

レポートの最初の *n* 個のエントリだけに出力するように指定します。*n* には、1 以上の符号無し整数を指定します。

`name { long | short }`

短い関数名と長い関数名のどちらを使うかを指定します (C++ だけで指定できます)。

`outfile {` *filename* | - }

開いている出力ファイルを閉じて、それ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ダッシュ (-) を指定した場合は、出力先は標準出力になります。

その他のコマンド

address_space

現在の実験のアドレス空間データを出力します。

header

現在の実験に関する詳細な情報を出力します。

help

ヘルプ情報を出力します。

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出力します。ファイル名の代わりにダッシュ (-) を指定した場合は、出力先は標準出力になります。

overview

選択している各標本の概要データを出力します。

quit

er_print を終了します。

script *script*

script で指定したスクリプトのコマンドを実行します。

statistics

現在の標本セットで収集した実行統計情報を出力します。

{ Version | version }

er_print のリリース番号を出力します。

第6章

上級項目: 標本アナライザとデータ

標本アナライザは、標本コレクタの収集したデータを読み込み、パフォーマンス測定結果に変換します。測定結果 (メトリック) は、目的とするプログラムのさまざまな要素に対して計算されます。収集されるイベントは、2 つに構成されます。

- 測定結果の計算に使うイベント固有データ
- 測定結果をプログラムの構造に関連付けるために使うアプリケーションの呼び出しスタック

この章では、以下の項目を説明します。

- イベント固有データとその内容
- 呼び出しスタックおよびプログラムの実行
- アドレスとプログラム構造のマッピング
- 注釈付きソースコードおよび逆アセンブリコード
- パフォーマンスコストについて

イベント固有データとその内容

記録されるイベント固有のデータには、正確な時刻、スレッド ID、LWP ID が含まれます。時間データを使って、実行の一部を選択することができます。スレッドおよび LWP の ID を使って、スレッドおよび LWP のサブセットを選択することができます。また、各イベントは固有の raw データを生成します。raw データについては、以下で説明します。

- 時間ベースのプロファイル
- 同期待ちの監視

■ ハードウェアカウンタのオーバーフロープロファイル

時間ベースのプロファイル

時間ベースのプロファイルは、各 LWP に対して送られたプロファイル信号ごとに集計したチック数 (時間カウント) のセットで構成されます。カーネルの管理するマイクロアカウント状態ごとに、個別にチックが存在します。マイクロアカウント状態の一部は、システム CPU 時間とシステム待ち時間に集計されます。残りの状態は、標本アナライザで個別に表示されます。

LWP が CPU のユーザーモードの場合は、通常生成されるチック配列には、ユーザー CPU 状態に対しては 1、他の状態に対してはゼロが含まれます。LWP が他の状態の場合は、チックは集計されますが、プロファイル信号はプロセスがユーザー CPU 状態に戻ったときに送信されます。

チックは整数で集計され、各チックは 1 つのプロファイル割り込み間隔を示します。LWP スケジューリングは、それよりも短時間の単位で行われます。したがって、プロファイルのパケットでの属性としては本質的に不安定です。通常は、すべての状態でのチックを合計して計算する LWP 時間の合計は、プロセスの `gethrtime()` が返す値と比較して、コンマ数パーセントまで同じです。CPU 時間は、プロセスの `gethrvtime()` が返す値と比較して、数パーセント程度異なる場合があります。負荷が高い場合は、差がさらに大きくなる場合があります。ただし、CPU 時間に差があっても系統的なひずみはなく、異なるルーチン、ソース行などに対してレポートされる相対時間は実質的にはひずみがありません。

`gethrtime()` および `gethrvtime()` については、これらの関数のマニュアルページを参照してください。

注 – 標本アナライザの表示する LWP 時間と `vmstat` の数値を比較する場合は、注意が必要です。標本アナライザの時間は、各 LWP のライフタイムの間のさまざまなマイクロ状態アカウント時間を合計した時間を示します。vmstat の表示する時間は、物理 CPU での時間を示します。たとえば、ターゲットプロセスの LWP 数が、LWP を実行するシステムの CPU 数よりも多い場合は、標本アナライザの示す待ち時間は `vmstat` の示す待ち時間よりも多くなります。CPU バウンドの LWP が 2 つで物理 CPU が 1 つという最も単純な例では、標本アナライザは 2 つの LWP の合計を示し、各 LWP の個別の待ち時間として約 50% を示します。`vmstat` は、アイドル時間がゼロであると示します。CPU は常にビジーですが、各 LWP の時間の半分は、他の LWP の実行中の待ち時間になります。

同期待ちの監視

同期待ちの監視イベントは、スレッドライブラリの関数呼び出しを追跡することで収集されます。イベントデータは、要求と許可 (追跡する呼び出しの最初と最後) の正確な時間データと、同期オブジェクト (要求される相互排他ロックなど) のアドレスで構成されます。要求と許可の時間差が、指定したしきい値を超えた場合にだけ、イベントが記録されます。同期トレースデータは、プロセスそのもので記録された時間と比較して、コンマ数パーセントの範囲で正確です。

ハードウェアカウンタのオーバーフロープロファイル

ハードウェアカウンタのオーバーフロープロファイルを使って、指定した LWP を実行している CPU のハードウェアカウンタおよびそのカウンタのオーバーフロー値 (増分数) を指定することができます。ハードウェアカウンタは、通常は、命令キャッシュミス、データキャッシュミス、クロックチック、実行された命令などを記録します。指定したカウンタがオーバーフロー値に到達すると、標本コレクタは LWP の呼び出しスタックを記録し、時間と、LWP およびその LWP 上で実行されているスレッドの ID を含めます。このデータは、標本アナライザで表示し、カウント測定結果のサポートに利用することができます。

ハードウェアカウンタはシステム固有であるため、選択可能なカウンタは使用するシステムによって異なります。多くのシステムは、ハードウェアカウンタのオーバーフロープロファイルをサポートしていません。その場合は、ハードウェアカウンタは使用することができません。

呼び出しスタックおよびプログラムの実行

呼び出しスタックは、プログラム内からの命令を示す一連のプログラムアドレス (PC) です。最初の PC (リーフ PC と呼びます) は、スタックの最後に格納される、次に実行する命令のアドレスです。次の PC は、リーフ PC を含む関数の呼び出しのアドレスです。さらに、その関数の呼び出しのアドレスが次の PC というように、スタックの先頭まで続きます。呼び出しスタックの記録プロセスを、スタックの展開と呼びます (101 ページの「スタックの展開」を参照)。

呼び出しスタックのリーフ PC は、その PC が含まれる関数にパフォーマンスデータの排他メトリックを関連付けます。スタックの他のすべての PC は、その PC が含まれる関数に包含メトリックを関連付けます。

ほとんどの場合は、記録された呼び出しスタックの PC はプログラムのソースコードでの記述通りに自然な形で関数に対応しており、標本アナライザの示す測定結果はそれらの関数に直接対応しています。ただし、プログラムの実際の実行が直観的な実行モデルとは異なり、標本アナライザの示す測定結果がわかりにくい場合があります。詳細は、102 ページの「アドレスとプログラム構造のマッピング」を参照してください。

シングルスレッドの実行および関数呼び出し

最も単純なプログラム実行は、シングルスレッドのプログラムがそ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、最初に実行するアドレス、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用に使われるメモリー領域) を含めるコンテキストが確立されます。最初のアドレスは、すべての実行可能ファイルで組み込まれている `_start()` という関数に常に含まれています。

プログラムが実行されると、命令が順に実行されます。命令が呼び出し、ジャンプ、分岐に到達すると、分岐のターゲットが示すアドレスに制御が移動し、そこから実行が続行されます。

呼び出しを示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格納され、呼び出し先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、呼び出し先関数の最初の数個の命令のいずれかで、新しいフレームがスタックにプッシュされ、復帰アドレスがそのフレームに格納されます。復帰アドレス用に使われるレジスタは、呼び出し先関数が他の関数を呼び出すときに使うことができます。関数が復帰するときは、スタックからフレームをポップし、関数の呼び出し元のアドレスに制御が戻ります。

共有オブジェクト間の関数呼び出し

ある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、プログラム内での単純な関数呼び出しよりも複雑になります。各共有オブジェクトには、プログラムリンクテーブル (PLT) が 1 つずつ含まれています。PLT には、共有オブジェクトの外部にあり、その共有オブジェクトを参照するすべての関数のエントリが含まれています。最初は、PLT 内の各外部関数の実際のアドレスは、動的リンカーである `ld.so` 内のアドレスです。このような関数が最初に呼び出されると、制御が動

的リンカーに移動します。動的リンカーは、その呼び出しから実際の外部関数を特定して、それ以降の呼び出しではその外部関数が使われるように PLT のアドレスを修正します。

シグナル

シグナルがプロセスに送信されると、シグナル送信時のリーフ PC をシステムルーチン `sigacthandler()` の呼び出しの復帰アドレスと見なすように、さまざまなレジスタおよびスタック操作が実行されます。`sigacthandler()` は、関数が他の関数を呼び出す場合と同様に、ユーザー指定のシグナルハンドラを呼び出します。標本アナライザは、この呼び出しのスタックフレームを通常と同様に処理します。ただし、スタックフレームは、任意の命令が呼び出しを生成できるように見せることができます。

高速トラップ

一部の命令は、カーネルに割り込み、軽量シグナルのユーザーモードに再度引き渡されます。これを高速トラップと呼びます。標本アナライザは高速トラップを認識しますが、SPARC-v9 で不正に割り当てられた整数メモリ参照は認識されません。その場合は、トラップ命令がハンドラを呼び出したものと見なされ、不正に割り当てられた整数トラップのフレームが表示されます。

カーネルトラップ

一部の命令は、カーネルに割り込み、カーネル内でエミュレートされます。たとえば、UltraSPARC-III プラットフォームの `fitos` 命令は、整数を単精度の浮動小数点数に変換します。標本アナライザでは特別な処理は実行されませんが、カーネルの処理が終了するまでトラップ命令は実行できないため、トラップ命令以降の命令の表示に時間がかかります。

テール呼び出し最適化

特定のルーチンが最後に実行する処理が他のルーチンの呼び出しである場合は、特別な最適化を実行することができます。呼び出し元は、実際に呼び出しを実行した後にスタックからフレームをポップして復帰する代わりに、スタックをポップしてからその呼び出し先を呼び出します。この最適化は、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が目的です。

たとえば、プログラムソースでは以下のような処理が指定されているとします。

```
A -> B -> C -> D
```

B および C でテール呼び出し最適化を実行すると、呼び出しスタックはプログラムが以下のような処理を実行するように見えます。

```
A -> B
A -> C
A -> D
```

つまり、呼び出しツリーが平坦化されます。-g オプションを指定してコードをコンパイルすると、テール呼び出し最適化は O4 以上でだけ実行されます。-g オプションを指定しないでコンパイルすると、テール呼び出し最適化は O2 以上でだけ実行されます。

明示的なマルチスレッド化

単純なプログラムは、単一の LWP (軽量プロセス) 上で単一スレッドで実行されます。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ます。このルーチンは、スレッドを実行する別の LWP を作成します。オペレーティングシステムは、CPU への LWP の割り当てを制御して LWP を実行します。スレッドライブラリは、スレッドを LWP で実行するスケジュールを制御します。新しく作成されたスレッドは、_thread_start() というルーチンで実行が開始されます。このルーチンは、スレッドを作成する呼び出しで引き渡された関数を呼び出します。スレッド化は、各スレッドが特定の LWP に結合される結合スレッドか、各スレッドが異なる LWP、異なる時間にスケジュールされることがある非結合スレッドのいずれかを使って行います。

並列実行およびコンパイラ生成の本体関数

コードに Sun、Cray、OpenMP の並列化指示が含まれている場合は、並列実行用にコンパイルすることができます (OpenMP は、Fortran 95 で使用可能です。並列化戦略の基礎および OpenMP の指示については、『Fortran プログラミングガイド』の並列化および OpenMP の章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、コンパイラ生成コードは複数のスレッドによって実行され、マイクロタスクライブラリによって調整されます。

コンパイラが並列構造を検出すると、並列構造の本体を別の本体関数に配置し、並列構造をマイクロタスクライブラリのルーチンの呼び出しに置き換えることで、並列実行用コードを生成します。マイクロタスクライブラリのルーチンは、スレッドを振り分けて本体関数を実行します。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引き渡されます。

- 並列構造が以下のいずれかで区切られている場合を説明します。
  - Sun の `c$par doall` 指示
  - Cray の `c$mic doall` 指示
  - OpenMP の `c$omp PARALLEL`、`c$omp PARALLEL DO`、`c$omp PARALLEL SECTIONS` 指示

  この場合は、並列構造がマイクロタスクライブラリのルーチン `__mt_MasterFunction_()` の呼び出しに置き換えられます。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_()` の呼び出しに置き換えられます。

- `c$omp PARALLEL` 構造に、`c$omp DO` または `c$omp SECTIONS` というワークシェアリング指示が含まれている場合は、各ワークシェアリング構造がマイクロタスクライブラリのルーチン `__mt_Worksharing_()` の呼び出しに置き換えられます。

コンパイラは、以下の形式の本体関数に名前を割り当てます。

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

- *string1* は、並列構造の種類 (`parallel`、`sections`、`doll`、`DOALL`) を示します。
- *namelength* は、*functionname* の文字数を示します。
- *functionname* は、構造の抽出元の関数の名前で、通常は最後が _ になります。
- *linenumber* は、元のソースでの構造の行番号です。
- *string2* は、ソースファイル名に対応しています。

データを解析しやすくするため、標本アナライザは、コンパイラ生成名に加えて、本体関数により読みやすい名前を割り当てます。

実行時は、最初はメインスレッドだけが実行されます。最初に `__mt_MasterFunction_()` が呼び出されると、`__mt_MasterFunction_()` は複数のワークスレッドの作成を開始します。作成されるワークスレッドの個数は、環境変数 `PARALLEL` または `OMP_NUM_THREADS` で指定するか、OpenMP の実行時ルーチンである `omp_set_num_threads()` を呼び出して指定します。それ以降は、`__mt_MasterFunction_()` がマスタースレッドとワークスレッドの間の作業配分を管理します。

メインスレッドでは、`__mt_MasterFunction_()` は一連の振り分け関数を呼び出し、最終的には本体関数を呼び出します (これは、並列化用にコンパイルされたコードを CPU が 1 つのマシンで実行する場合や、CPU が複数のマシンでスレッドを 1 つだけ使う場合にも該当します)。

ワークスレッドは、Solaris のスレッドライブラリを使って作成します。ワークスレッドの呼び出しスタックは、`_thread_start()` というスレッドライブラリのルーチンを最初に呼び出します。`_thread_start()` は、`__mt_SlaveFunction_()` を呼び出します。この関数は、スレッドのライフタイムの間、スレッドによって継続して実行されます。`__mt_SlaveFunction_()` は、`__mt_WaitForWork_()` を呼び出し、作業が配分されるまでスレッドを待機させます。作業が配分されると、スレッドが `__mt_SlaveFunction_()` に戻り、本体関数の呼び出しを開始します。作業が終了すると、ワークスレッドは `__mt_SlaveFunction_()` に戻り、`__mt_WaitForWork_()` を再度呼び出します。スレッドの制御フローは、標本アナライザの「呼び出し元-呼び出し先」ウィンドウで確認することができます。このウィンドウの使い方については、60 ページの「関数の呼び出し元と呼び出し先の測定結果の検査」を参照してください。

注 – 前述の呼び出し順序では、標本アナライザはコンパイラ生成関数の抽出元関数の呼び出しを示します。この呼び出しは、元の関数がコンパイラ生成関数を呼び出したように挿入され、包含的データは元の関数に対して表示されます。

ワークスレッドは、`__mt_WaitForWork()` では通常は CPU 時間を使って、作業が配分されたとき、つまりメインスレッドが新しい並列構造に到達したときの応答時間を短縮します (これをビジー待機と呼びます)。ただし、環境変数でスリープ待機を指定することもできます。この場合は、標本アナライザでは CPU 時間ではなく LWP 時

間として表示されます。ワークスレッドが作業の配分を待つのは、一般的に以下の 2 つの場合です。プログラムを設計し直して、この待ち時間を削減することをお勧めします。

- メインスレッドがシリアル領域で実行されていて、ワークスレッドが実行する作業がない場合
- 作業負荷が平均化されていないため、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場合

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います。Makefile の `FLAG` 変数を `UNBOUND` に設定してからプログラムを構築するか、環境変数 `MT_BIND_LWP` を `FALSE` に設定することで、デフォルト設定を無効にすることができます。

インデックスが `long long` のループは、インデックスが `integer` または `long` のループとは異なるマイクロタスクライブラリのルーチンを呼び出します。

注 – 多重処理振り分けプロセス全体は、システムに依存しません。また、将来のリリースでは変更される場合があります。

スタックの展開

標本コレクタがイベントを記録するときは、イベント発生時のプロセスの呼び出しスタックを記録します。記録される呼び出しスタックは、次に実行される命令のアドレス (リーフ PC)、復帰レジスタの内容、スタックの各フレームの復帰アドレスの内容で構成され、最後は `_start()` (メインスレッド用) および `_thread_start()` (ワークスレッド用) での呼び出し命令のアドレスになります。

標本コレクタは、常に復帰レジスタを記録します。標本アナライザは、発見的手法を使って、復帰アドレスがスタックにプッシュされているかどうかを特定します。プッシュされている場合は、復帰レジスタは無視されます。プッシュされていない場合は、復帰レジスタが呼び出し元 PC として使われます。フレームポインタと呼び特定のレジスタを使って、スタックの最初のフレームが特定されます。各フレームには、呼び出し元のフレームを特定するために使う前のフレームポインタが含まれています。Intel マシンの場合は、最適化されたコードでは、前のフレームポインタが各スタックフレームに含まれていないため、発見的手法を使ってスタックが展開されます。

アドレスとプログラム構造のマッピング

標本アナライザは、呼び出しスタックを処理して PC 値を計算した後に、PC をプログラム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。アドレス空間には、実行可能ファイルの命令を示すテキストや、通常は実行されないデータなど、プロセスが占有する多数の領域があります。呼び出しスタックに記録される PC は、通常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ます。

プロセスの最初のテキストセクションは、実行可能ファイルそのものから派生します。他のテキストセクシンは、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、または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。呼び出しスタックの PC は、呼び出しスタックの記録時に読み込まれている実行可能ファイルおよび共有オブジェクトに基づいて解析されます。実行可能ファイルおよび共有オブジェクトは非常に似ていて、集合的にロードオブジェクトと呼ばれます。

共有オブジェクトは、プログラム実行中に読み込みおよび読み込み解除することができるため、実行中は時間が異なれば PC が指す関数も異なる場合があります。また、共有オブジェクトが読み込み解除された後に異なるアドレスで再度読み込まれた場合は、異なる PC が同一の関数を指す場合もあります。

ロードオブジェクトおよび関数

実行可能ファイルや共有オブジェクトにかかわらず、各ロードオブジェクトには、生成された命令を示すテキストセクション、データ用データセクション、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必要です。ELF シンボルテーブルには、そのオブジェクトの大域的な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

ボルテーブルを拡張するもので、大域的でない関数に関する情報、関数の派生元のオブジェクトモジュールに関する情報、アドレスをソース行に関連付ける行番号を提供します。

関数という用語は、ソースコードで記述された高レベルの処理を示す命令セットを意味します。関数には、Fortran のサブルーチン、C++ のメソッドなども含まれます。関数は、ソースコードで明確に記述され、通常は、アドレスセットを示すシンボルテーブルで関数の名前が示されます。プログラムカウンタがアドレスセット内に含まれる場合は、プログラムのその関数が実行されています。

原則として、ロードオブジェクトのテキストセグメント内のアドレスは、関数にマッピングすることができます。まったく同一のマッピングが、呼び出しスタックのリーフ PC および他のすべての PC にも使われます。以下で説明する一部の関数を除き、関数はプログラムのソースモデルに直接対応しています。

- 103 ページの「エイリアス関数」
- 104 ページの「一意でない関数名」
- 104 ページの「ストリップ済み共有ライブラリの静的関数」
- 105 ページの「Fortran の代替エントリポイント」
- 105 ページの「インライン関数」
- 106 ページの「コンパイラ生成の本体関数」
- 107 ページの「アウトライン関数」
- 107 ページの「<未知> 関数」
- 108 ページの「<合計> 関数」

エイリアス関数

通常、関数は大域的に定義されます。つまり、関数名はプログラムのすべての場所で既知になります。大域関数の名前は、実行可能なファイル内で一意にする必要があります。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は、実行時リンカーはそれらの関数のいずれか 1 つだけに対する参照を有効にし、他の関数は実行されないため、参照されない関数は関数リストには表示されません。「概要メトリック」ウィンドウでは、選択した関数を含む共有オブジェクトおよびオブジェクトモジュールを確認することができます。

さまざまな状況で、関数が異なる名前で認識される場合があります。代表的な例として、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を使う場合があります。強い名前は、通常は対応する弱い名前と同一です。ただし、最後に _ が付きます。スレッドライブラリの多くの関数にも、pthread および Solaris スレッド用の別

名と、強い名前、弱い名前、内部の代替シンボルがあります。いずれの場合も、標本アナライザの関数リストではいずれかの名前だけが表示されます。表示されるのは、与えられたアドレスにおいて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う名前に対応します。「概要メトリック」ウィンドウでは、選択した関数のすべてのエイリアス (別名) が表示されます。

一意でない関数名

エイリアス関数は、コードの同一部分に対して複数の名前を使います。一方で、コードの複数の部分に同一名を使う場合があります。

- モジュール性を実現するため、関数が静的として定義されることがあります。つまり、関数名がプログラムの一部 (通常はコンパイルしたオブジェクトモジュール) でだけ認識されます。このような場合は、プログラムの全く異なる部分に対して同一名が 標本アナライザで表示されることがあります。「概要メトリック」ウィンドウでは、このような関数を区別するため、各関数のオブジェクトモジュール名が表示されます。また、どの関数を選択しても、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、ライブラリ関数の呼び出しに対して、名前がそのライブラリ内の関数の弱い名前であるラッパー関数または割り込み関数を代わりに使う場合があります。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ます。この場合は、名前の両方のインスタンスが 標本アナライザの関数リストで表示されます。このような関数は、共有オブジェクトおよびオブジェクトモジュールが異なるため、それらを基準にして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し、ラッパー関数と実際の関数の両方を 標本アナライザで表示することができます。

ストリップ済み共有ライブラリの静的関数

静的関数は、ライブラリ内部での関数名がユーザーの使う関数名と衝突しないように、ライブラリ関数内でよく使われます。ライブラリをストリップすると、静的関数の名前は削除されます。この場合は、標本アナライザは `<static>@0x12345` という形式の名前を生成します。ここで、`@` 記号の後には、その関数のライブラリ内でのオフセットを示す文字列が付きます。標本アナライザは、隣接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は測定結果がいっしょに表示される場合があります。

Fortran の代替エントリポイント

Fortran では、コードの一部に対して複数のエントリポイントを使い、呼び出し元が関数の途中に割り込むことができます。このようなコードをコンパイルすると、コンパイル後の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、関数のコードの本体で構成されます。各導入部は、関数が最後に復帰するためのスタックを作成してから、コード本体に分岐または接続します。

Fortran の代替エントリポイントのあるサブルーチンの処理は、コンパイラによって異なります。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示すテキストの領域に常に対応しますが、ルーチンの本体のコードは、2 つのエントリポイント名のどちらかに関連付けられます。

- SPARC プラットフォームでは、WS 6 コンパイラは代替エントリポイントの導入部を最初に生成し、次にメインのエントリポイントおよび本体のコードを生成します。本体のコードのすべての測定結果は、メインのエントリポイントの名前に関連付けられます。
- x86 プラットフォームでは、コンパイラはメインのエントリポイントの導入部を最初に生成し、次に代替エントリポイントおよび本体のコードを生成します。本体のコードの測定結果は、代替エントリポイントの名前に関連付けられます。多くの場合は、導入部の時間はわずかで、サブルーチン本体に関連付けられたエントリポイントとは別のエントリポイントに対応する関数は表示されません。

インライン関数

インライン関数は、ソースでは関数として定義され、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場所に挿入される命令になるコードです。インライン化には 2 種類あり、いずれも 標本アナライザに影響します。

- C++ のインライン関数定義
- 高レベルの最適化 (O4 および O5) で実行される明示的または自動インライン化

いずれのインライン化も、パフォーマンス向上のために行われます。

C++ のインライン化を指定するには、メソッドの本体をそのメソッドのクラス定義に含めるか、メソッドをインライン関数として明示的に示します。このインライン化を行うのは、関数の呼び出しの方がインライン関数よりも処理時間がかかるためです。したがって、呼び出しを実行する代わりに、呼び出しを実行する場所に単に関数のコードを挿入する方が効率的です。アクセス関数では、必要な命令が 1 つだけの場合

が多いため、通常はアクセス関数はインライン関数として定義されます。通常は、-g オプションを指定してコンパイルすると、インライン関数として定義した関数であっても、通常の関数としてコンパイルされます。ただし、C++ で -g0 というオプションを指定してコンパイルした場合は、他の最適化を指定しない場合でも、インライン関数として定義されたすべての関数がインライン関数としてコンパイルされます。

高レベルの最適化では、-g オプションを指定した場合でも、明示的および自動インライン化が実行されます。こ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が、多くの場合は、レジスタを使う命令をより多く提供し、命令のスケジューリングの最適化を行うことが目的です。

いずれのインライン化も、関数リストでは同様に表示されます。ソースコードには記述されているがインライン化された関数は、関数リストでは表示されず、インライン関数の呼び出し場所で通常であれば包含的な測定結果と見なされる結果 (呼び出し先関数で消費された時間を示します) は、実際は排他的な測定結果 (呼び出し場所に挿入されたインライン関数の命令を示します) として表示されます。

注 – 多くの場合は、インライン化によってデータの解析が難しくなります。したがって、パフォーマンス解析を行うときはインライン化を無効にすることをお勧めします。

場合によっては、関数がインライン化されていても、いわゆる行の範囲外の関数が残る場合があります。ある呼び出し場所では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、関数はインライン化されていないように関数リストで表示されますが、その測定結果は行の範囲外の呼び出しだけを反映しています。

コンパイラ生成の本体関数

コンパイラが関数内のループまたは並列化指示のある領域を並列化すると、プログラムのソースモデルでは明示的に現れない新しい本体関数を作成します。このような本体関数の詳細は、98 ページの「並列実行およびコンパイラ生成の本体関数」を参照してください。

標本アナライザは、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て、抽出元の関数に基づくラベルを付けます。これらの関数の排他的および包含的な測定結果は、本体関数で消費された時間を示します。また、構造の抽出元関数は、各本体関数の包含的な測定結果を示します。

並列ループを含む関数がインライン化された場合は、そのコンパイラ生成の本体関数は、元の関数ではなく、インライン化された関数を反映します。

アウトライン関数

フィードバック最適化の際に、アウトライン関数が作成されることがあります。アウトライン関数は、通常は実行されないコードを示します。特に、フィードバック生成に使われる「試験実行」の際に実行されないコードを示します。ページングおよび命令キャッシュの動作を向上するため、アウトライン関数はアドレス空間の別の場所に移動され、以下の形式の名前の関数に変換されます。

```
_$1$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対応しています。
- *namelength* は、*functionname* の文字数を示します。
- *functionname* は、構造の抽出元の関数の名前です。
- *linenumber* は、元のソースでの構造の行番号です。
- *string2* は、コンパイラの内部名に対応しています。

アウトライン関数は、通常の関数として、適切な包含的および排他的な測定結果とともに表示されます。また、アウトライン関数の測定結果は、元の関数での包含的測定結果として追加されます。

コンパイラ生成の本体関数と同様に、標本アナライザはアウトライン関数の元の関数からの呼び出しを表示します。

<未知> 関数

場合によっては、PC が既知の関数にマッピングされないことがあります。このような場合は、PC は <未知> という特別な関数にマッピングされます。

以下の場合は、PC が <未知> にマッピングされます。

- PC が動的リンカーである `ld.so` に属する場合。

- PC がロードオブジェクト内の PLT に対応している場合。この状況は、ある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発生します。実際の呼び出しは、最初に PLT の 3 つの命令のシーケンスに転送され、次に実際の宛先に転送されます。

- 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する場合。通常はデータは実行できないため、データのアドレスは呼び出しスタックには表示されません。ただし、コードが自己修飾の場合は、プログラムがこれらの書き込み可能命令をプログラムのデータ空間に書き込んでから実行することがあります。SPARC v7 の `libc.so` では、データセクションに複数の関数 (`.mul` や `.div` など) があります (ライブラリが SPARC v8 または v9 マシンで実行されていることを検出した場合に、マシンの命令を使うように動的に書き込み直すことができるように、コードはデータセクションに含まれています)。

- PC が既知のロードオブジェクト内にない場合。多くの場合は、展開の失敗により、PC として記録された値が PC ではなく別の語であることが原因です (PC が復帰レジスタで、既知のロードオブジェクト内にないと見なされる場合は、 <未知> 関数にマッピングされるのではなく無視されます)。

<合計> 関数

<合計> 関数は、プログラム全体を示すための構造です。すべてのパフォーマンス測定結果は、呼び出しスタックの関数にマッピングされる以外に、<合計> という特別な関数にもマッピングされます。この関数は、関数リストの最初に表示され、そのデータを使って他の関数のデータを概観することができます。

「呼び出し元-呼び出し先」ウィンドウ

ここでは、「呼び出し元-呼び出し先」ウィンドウと、このウィンドウでのプログラム実行の表示について説明します。

<合計> 関数

特別な関数である <合計> は、プログラム実行のメインスレッドでの `_start()` の名目上の呼び出し元、および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。

Fortran の代替エントリポイント

Fortran の代替エントリポイントのあるサブルーチンでの時間を示す呼び出しスタックは、通常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。いずれの場合も、標本アナライザは、収集されたデータを使ってメインのエントリポイントの呼び出しと代替エントリポイントの呼び出しを区別することができません。

同様に、サブルーチンからのすべての呼び出しは、サブルーチンの本体に関連付けられた名前から作成されたものとして表示されます。

インライン関数

インライン関数は、インライン化後のルーチンの呼び出し先としては表示されません。いずれかの場所でインライン化されたが、別の場所の通常の関数として表示される関数のデータを解析するときは注意してください。標本アナライザでは、通常の関数の測定結果だけが表示されます。この結果は、その関数のすべてのインスタンス (インライン化されたものと通常のもの) の合計の測定結果の一部だけを示している場合があります。

コンパイラ生成の本体関数

コンパイラ生成の本体関数は、98 ページの「並列実行およびコンパイラ生成の本体関数」で説明するように、マイクロタスクライブラリのルーチンによって直接呼び出されます。ただし、標本アナライザで表示される動作を実行のソースモデルに近づけるため、標本アナライザはループルーチンの抽出元関数からの疑似呼び出しを抽出元の行で表示します。したがって、標本アナライザでは、本体のルーチンの抽出元関数が呼び出し元として表示され、その関数に対する包含的時間が測定されます。

アウトライン関数

アウトライン関数は、実際には呼び出し先ではなくジャンプ先になります。同様に、復帰で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モデルに近づけるため、標本アナライザはメインルーチンからの疑似呼び出しをそのアウトライン部分に関連付けます。

テール呼び出し最適化

テール呼び出し最適化が行われた中間呼び出しは、「呼び出し元-呼び出し先」ウィンドウでは明示的に表示されません。

シグナル

標本アナライザは、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。シグナルの送信時のユーザーコードがシステムルーチン `sigacthandler()` を呼び出し、`sigacthandler()` がユーザーのシグナルハンドラを呼び出すように表示されます。`sigacthandler()` およびユーザーのシグナルハンドラの両方、およびそれらが呼び出す他の関数の包含的な測定結果は、割り込まれたルーチンの包含的な測定結果として表示されます。

ストリップ済み静的関数

ストリップ済み静的関数は、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。ただし、静的関数の PC が、静的関数の save 命令の後に表示されるリーフ PCである場合を除きます。シンボル情報がない場合は、標本アナライザは save のアドレスを認識せず、復帰レジスタを呼び出し元として使うかどうかを決定できません。そのため、常に復帰アドレスを無視します。複数の関数が `<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ルーチンと区別されないことがあります。

<未知> 関数

<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 を示し、通常と同様に処理されます。

再帰呼び出し

再帰呼び出しとは、関数がそれ自身を呼び出すことです。「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。

注釈付きソースコードおよび逆アセンブリコード

標本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の機能は、関数内の処理が原因で性能が低下する場合に便利です。

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資源消費を示します。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 をソース行にマッピングすることで作成されます。標本アナライザは、注釈付きソースファイルを作成するため、最初に特定のオブジェクトモジュール (`.o` ファイル) で生成されたすべての関数を特定し、各関数の PC すべてのデータを調べます。注釈付きソースを作成するには、標本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込み、PC とソース行のマッピングを特定できる必要があります。また、表示用に、ソースファイルを読み込み、注釈付きソースのコピーを作成できる必要があります。

コンパイル処理では、要求される最適化のレベルに応じて多くの段階があり、プログラム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、さまざまな発見的手法によってソース行の命令を追跡しますが、発見的手法は完全ではありません。

表 6-1 に、注釈付きソースコードの行で表示される 4 種類の測定結果を示します。

表 6-1　注釈付きソースコードの測定結果

測定結果	意味
(空白)	この行に対応する PC がプログラムに存在しません。コメント行では常にこの結果になります。また、以下の場合のコード行でも同様の結果になります。 • コード行の命令がすべて最適化され取り去られた場合 • 別の場所とコードが一致していて、コンパイラが共通部分式を認識し、その行の全部の命令を別の行に結びつけた場合 • コンパイラがその行の命令に間違った行番号結びつけた場合
`0.`	この行に対応する 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。統計的に標本採取された呼び出しスタックまたはスレッド同期データのために追跡された呼び出しスタックに、この PC が存在しないことを示します。測定結果が 0. の場合は、行が実行されなかったことを示すのではなく、統計上プロファイルに表れないこと、および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示します。
`0.000`	この行の PC の少なくとも 1 つがデータに表れているが、測定結果の合計値が丸められてゼロになったことを示します。
`1.234`	この行のすべての PC の測定結果の合計がゼロ以外になり、数値として表示されています。

コンパイラのコメント

コンパイラのさまざまな部分で、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対応しています。

一部のコメントは、`f95` コンパイラによって挿入されます。これらのコメントは、配列セクションをサブルーチンに引き渡す際に必要なコピーインおよびコピーアウトが原因のパフォーマンス低下の可能性を示します。コードを並列化の解析用にコンパイルした場合は、ループの並列化状態を示すコメントも挿入されます。

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。

不明行 - <sum of all instructions without line numbers>

PC に対応するソース行を特定できない場合は、その PC の測定結果は、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関連付けられます。このソース行の測定結果が高い場合は、オブジェクトモジュールのコードの一部に行がマッピングされていないことを示します。注釈付き逆アセンブリを使って、マッピングのない命令の内容を特定することができます。

共通部分式の除去

最も一般的な最適化では、複数の場所に出現する同一の式を検出し、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を向上します。たとえば、コードの同一ブロックの `if` と `else` の分岐の両方で同一の処理が記述されている場合は、コンパイラはその処理を `if` の前に移動します。その場合は、コンパイラはそれ以前に出現する同一式のいずれかを基準にして、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` 構造の一方の分岐に対応していて、実際にはもう一方の分岐が常に実行される場合は、注釈付きソースでは、実行されない分岐内の行の測定結果が表示されます。

注釈付き逆アセンブリ

注釈付き逆アセンブリは、各命令に対応するパフォーマンス測定結果が挿入された、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を提供します。命令の出現頻度が高いほど、その関数に消費される時間は多くなります。注釈付き逆アセンブリは、行番号マッピングおよびソースファイルが使用可能かどうか、および注釈付き逆アセンブリが要求され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、複数の方法のいずれかで表示されます。

- オブジェクトモジュールが既知ではない場合は、標本アナライザは指定した関数だけの命令を逆アセンブルし、逆アセンブリ内のソース行は表示しません。
- 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。
- ソースファイルが使用可能で、行番号データが記録されている場合は、標本アナライザはソースに逆アセンブリを挿入します。
- コンパイラがオブジェクトコードにコメントを挿入した場合は、コメントも注釈付きソースおよび逆アセンブリに挿入されます。

コードが最適化されていない場合は、行番号は単純で、ソースおよび逆アセンブルされた命令はそのまま表示されます。最適化されている場合は、後の命令が前の行よりも前に表示されることがあります。標本アナライザの挿入アルゴリズムでは、命令が行 *N* に記述されている場合は、行 *N* までのすべてのソース行をその命令の前に挿入します。ソースの行 *N* に対応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。

逆アセンブリコードの各命令には、以下の注釈が付きます。

- コンパイラがレポートするソースでの行番号
- 命令の相対アドレス
- 命令を示す 16 進数
- 命令を示すアセンブラのテキスト

可能な場合は、呼び出しアドレスがシンボルに変換されます。命令の行には測定結果が表示されますが、挿入されたソースまたはコメントには表示されません。表示される測定結果の値については、112 ページの表 6-1 に示すソースコードの注釈を参照してください。

パフォーマンスコストについて

関数レベル、ソース行レベル、逆アセンブリ命令レベルで、測定結果の値を調べることができます。各レベルの測定結果の値が高い場合は、別の方法でコードを改良して効率を向上することができます。

関数レベルのパフォーマンス

関数の測定結果は、実行回数が多い、あるいは 1 回の実行にかかる時間が長い場合に値が高くなります。

- 関数の実行回数が多い場合は、呼び出し回数を削減するか、関数をインライン化することで、パフォーマンスが向上する可能性があります。
- 1 回の実行にかかる時間が長い場合は、関数のアルゴリズムをより効率的にすることで、パフォーマンスが向上する可能性があります。

通常は、関数の注釈付きソースを調べることで、パフォーマンスの向上方法を特定するのが最も簡単です。

ソース行レベルのパフォーマンス

注釈付きソースで測定結果の値が高い行は、関数で実行時間の大半が消費されている部分を示しています。アルゴリズムを改良または見直すか、関数の最適化レベルを高くすることで、パフォーマンスが向上する可能性があります。アルゴリズムが効率的で最適化にも問題がない場合は、注釈付き逆アセンブリを調べることで、パフォーマンスの向上方法を特定できることがあります。

命令レベルのパフォーマンス

通常は、命令レベルでの効率的なコードの生成はコンパイラが行います。場合によっては、特定のリーフ PC の示す命令の実行前に遅延があり、出現頻度が高くなることがあります。また、命令がカーネルに割り込むときなどのように、前の命令の実行に時間がかかり、割り込みが不可能なため、特定のリーフ PC が出現することがあります。

命令の実行遅延には複数の原因があり、いずれもパフォーマンス向上につながる可能性があります。命令の実行遅延は、数値命令がレジスタを必要とするが、そのレジスタの内容が前の命令によってセットされていて、前の命令がまだ終了していないために発生することがあります。このような遅延の例として、データキャッシュミスのある読み込み命令や、実行に 1 サイクル以上かかる浮動小数点数の除算などの浮動小数点数演算命令などがあります。

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にも、命令の出現頻度が過度に高くなることがあります。ある命令が常に前の命令と同一のクロックで実行され、次に実行される命令として表示されない場合には、出現頻度が過度に低くなることがあります。

第7章

ループ解析ツール

Fortran および Cコンパイラは、並列化しても安全が損なわれることなく、かつ並列化することが効果的であると判断されるループを対象として、自動的に並列化を行います。パフォーマンス解析ツールであるループツールは、これらのコンパイラによって作成されるループタイミングファイルを読み取ります。ループツールにはグラフィカルユーザインタフェース (GUI) が用意されています。また、ループツールのコマンド行バージョンとしてループレポートも利用できます。

この章は、以下の項目から構成されています。

- 基本概念
- 環境の設定
- ループタイミングファイルの生成
- ループツールの起動
- ループツールの使用
- ループレポートの起動
- コンパイラヒント
- コンパイラの最適化とループへの影響

基本概念

ループツールおよびループレポートには以下の機能があります。

- 直列/ 並列を問わず、すべてのループのタイミングを測定します。
- ループタイミングの表を作成します。
- コンパイル中にコンパイラからヒントを収集します。

ループツールはループルーチンのグラフを表示し、どのループが並列化されたかを示します。ループのグラフ表示画面から任意のループのソースコードへと直接移動することにより、ループツールの中でソースコードを直接編集できます。

ループレポートは、グラフィカルな表示方法ではなく、ASCII ファイルとして、ループ実行時のレポートを作成します。

ループツールおよびループレポートを使用するための基本的な 4 つの手順を示します。

1. 環境変数を設定する。
2. ループ解析用のタイミングファイルを作成するためのオプションを使って、プログラムをコンパイルする。
3. プログラムを実行して、タイミングファイルを作成する。
4. そのタイミングファイル上でループツールまたはループレポートを起動する。

注 – この項の例は Fortran (f77 と f95) コンパイラを使用していますが、表示されているオプション (`-xparallel`、`-Zlp` など) は、C コンパイラでも利用できます。

環境の設定

`-Zlp` オプションをつけて実行可能ファイルをコンパイルする前に、使用するマシンに搭載されたプロセッサの数を環境変数 `PARALLEL` に設定します。

以下のコマンドでは、`psrinfo` というシステムユーティリティを使用しています。逆引用符の内部に注目してください。

```
% setenv PARALLEL `/usr/sbin/psrinfo | wc -l`
```

このコマンドはシェル起動ファイル (`.cshrc` または `.profile`) に入れることができます。

ループタイミングファイルの生成

ループタイミングファイルを生成するには、自動的にコードを並列化し、最適化するコンパイラオプション (`-xparallel` と `-xO4`) を使って、プログラムをコンパイルします。 また、ループツールやループレポート用にコンパイルを行うには、 `-Zlp` オプションを追加します。これらのオプションを使ってプログラムをコンパイルすると、Sun WorkShop はループツールおよびループレポートが処理するためのタイミングファイルを生成します。

これら 3 つのコンパイルオプションの使用例を以下に示します。

```
% f77 -xO4 -xparallel -Zlp source_file
```

注 – いずれの例も FORTRAN 77、Fortran 95、および C プログラムに適用されます。

ループの確認および並列化に使用できる便利なオプションがほかにも多数あります。表 7-1に、これらのオプションを示します。

表 7-1 コンパイルオプション

オプション	効果
`-o` *program*	実行可能ファイル名を *program* に変更します。
`-xexplicitpar`	`DOALL` プラグマの付いたループを並列化します。
`-xloopinfo`	`stderr` にヒントを出力します。

その他のコンパイラオプション

ループツールおよびループレポートでは、各種のコンパイラオプションを組み合わせることができます。

自動並列化用にコンパイルする場合は、通常は -xparallel および -x04 というコンパイラオプションを指定します。ループツールとループレポート用のコンパイルを行うには、 -Zlpを追加します。

```
% f77 -x04 -xparallel -Zlp source_file
```

-xO3 または -xO4 と -xparallel は併用できます。-xO3、-xO4 のいずれも指定せず、-xparallel のみ指定する場合、コンパイラは -xO3 を使用します。表 7-2 には、特定のオプションについての最適化レベルオプションの追加方法をまとめてあります。

表 7-2　最適化レベルオプションと暗黙指定

入力	指定されるオプション
-xparallel	-xparallel -xO3
-xparallel -Zlp	-xparallel -xO3 -Zlp
-xexplicitpar	-xexplicitpar -xO3
-xexplicitpar -Zlp	-xexplicitpar -xO3 -Zlp
-Zlp	-xdepend -xO3 -Zlp

これ以外に、-xexplicitpar と -xloopinfo というコンパイルオプションがあります。

Fortranコンパイラオプション -xexplicitpar は、 プラグマ DOALL とともに使用されます。ソースコード中で、あるループの前に DOALL を挿入すると、並列化のための明示的なマークが付けられます。-xexplicitpar を指定してコンパイルすると、そのループが並列化されます。

以下のコードは、ループの並列化を明示的にマークする方法を示したものです。

```
    subroutine adj(a,b,c,x,n)
      real*8 a(n), b(n), c(-n:0), x
        integer n
c$par DOALL
    do 19 i = 1, n*n
      do 29 k = i, n*n
        a(i) = a(i) + x*b(k)*c(i-k)
29    continue
19  continue
    return
    end
```

-Zlp を単独で使用すると、-xdepend と -xO3 が追加されます。-xdepend オプションは、コンパイラに対してループを識別するときに必要なデータ依存関係の解析を実行するよう指示します。-xparallel オプションには -xdepend も含まれますが、-xdepend は -xparallel の暗黙指定(トリガー)ではありません。

-xloopinfo オプションは、プログラムのコンパイル時に、ループに関するヒントを stderr (ファイル記述子 2 の UNIX 標準エラーファイル)に出力します。このヒントには、ルーチン名、ループの開始地点を示す行番号、ループの並列化の有無、並列化されていない場合はその理由などが示されます。

以下の例では、ソースファイル gamteb.F のループに関するヒントをファイル *gamtab*.loopinfo にリダイレクトしています。

```
% f77 -xO3 -parallel -xloopinfo -Zlp gamteb.F 2> gamteb.loopinfo
```

-Zlp と -xloopinfo の最大の違いは、ループに関するコンパイラヒントを出力することに加えて、-Zlpによって実行時にタイミングに関する統計情報を記録するようになるという点です。このため、ループツールとループレポートは、-Zlp を指定してコンパイルされたプログラムだけを対象として解析を行います。

プログラムの実行

-Zlpを指定してコンパイルした後、実行可能ファイルを実行してください。この結果、ループタイミングファイル *program*.looptimes が作成されます。ループツールとループレポートは、計測機構付きの実行可能ファイルとループタイミングファイルの 2 つのファイルを処理します。

ループツールの起動

読み込むプログラム (実行可能ファイル) の名前を指定することにより、ループツールを起動できます。

```
% looptool program &
```

ファイル指定なしにループツールを起動した場合、「ファイルを開く」ダイアログボックスが開き、検証するファイルの選択を要求されます。

```
% looptool &
```

ループツールは、プログラムに関連するタイミングファイルを読み込みます。タイミングファイルには、ループに関する情報が記載されています。通常、タイミングファイルには *program*.looptimes という形式の名前が付けられ、プログラムと同じディレクトリに入れられます。

デフォルトでは、ループツールは実行可能ファイルのあるディレクトリを検索して、タイミングファイルを探します。したがって、タイミングファイルが所定のディレクトリにあれば (正常な状態) 、探す場所を指定する必要はありません。

```
% looptool program &
```

コマンド行でタイミングファイルの名前を指定すると、ループツールおよびループレポートはそのファイルを使用します。

```
% looptool program program.looptimes &
```

コマンド行オプション -p を指定すると、ループツールおよびループレポートは -p で示されたディレクトリを検索して、タイミングファイルを探します。

```
% looptool -p timing_file_directory program &
```

環境変数 LVPATH が設定されている場合、ツールは LVPATH で指定されたディレクトリを検索して、タイミングファイルを探します。

```
% setenv LVPATH timing_file_directory
% looptool program &
```

ループツールの使用

メインウィンドウに、ソースファイルがコンパイラに渡された順序に従って、プログラムにある各ループの各実行時間が棒グラフとして表示されます。

図 7-1には、ループツールウィンドウの構成要素を示します。

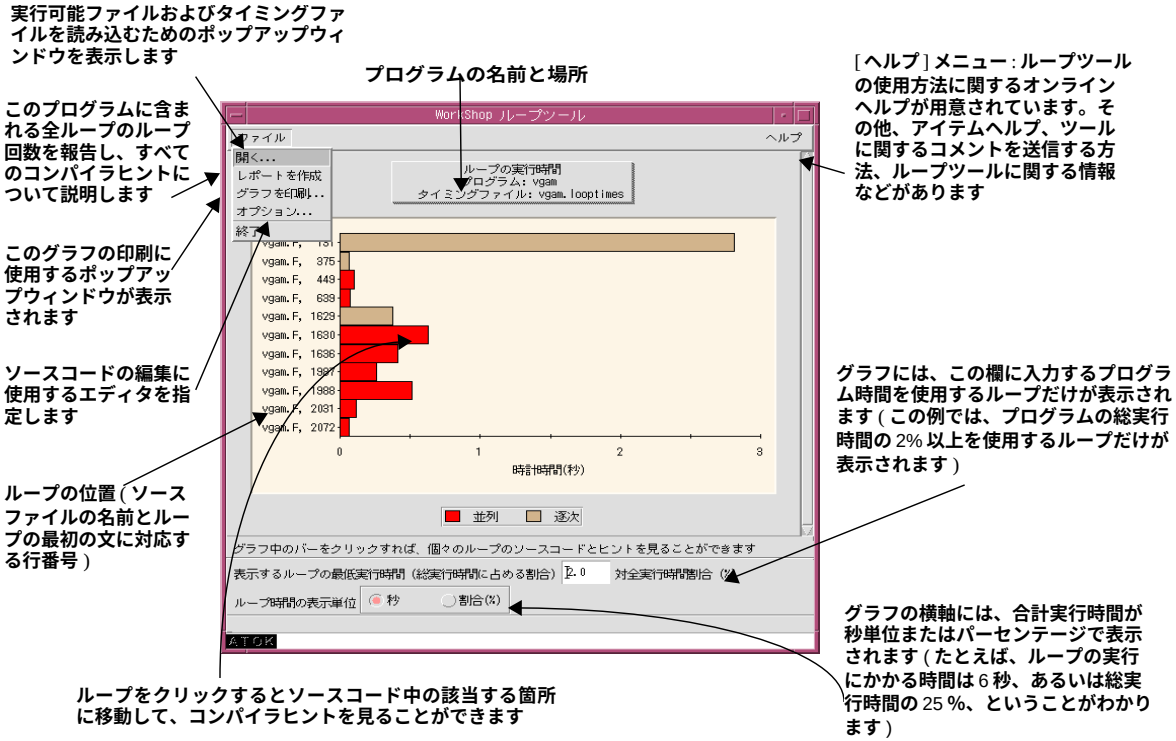

図 7-1 「ループツール」ウィンドウ

ファイルを開く

実行可能ファイルとタイミングファイルを開くには、メインウィンドウの「ファイル」メニューの「開く」コマンドを選択します。

以下の 2 つの方法で、開くファイルを指定できます。

1. 開くファイルの名前を入力する。

2. ファイルチューザを起動する。

実行可能ファイルのパスを入力した後は、タイミングファイルを指定する必要はありません。ただし、タイミングファイルが別のディレクトリにある場合やデフォルト以外の名前を使用している場合は除きます。

ファイルを開く方法の詳細については、Sun WorkShop オンラインヘルプの「プログラム中のループの分析」を参照してください。

全ループに関するレポートの作成

プログラム中にある全ループに関する情報を含むウィンドウ (図 7-2 参照) を開くには以下を行います。

メインウィンドウの「ファイル」メニューから「レポートを作成」を選択します。(図 7-2 参照)

ここで作成されるレポートは、ループレポートで作成されるものと同じです。

レポートウィンドウの「ヘルプ」 ボタンは、コンパイラに関するヒントを含む Sun WorkShop オンラインヘルプの関連項目にリンクされています。

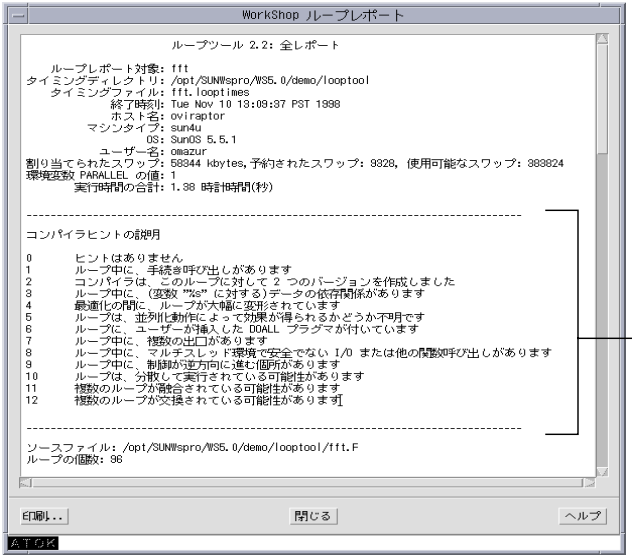

図 7-2　ループレポート

ループツールグラフの印刷

ループツールのグラフを印刷するには、メインウィンドウの「ファイル」メニューから 「グラフを印刷」を選択し、選択しているプリンタ名を入力します。グラフをファイルに保存するには、プリンタ名の代わりにファイル名を入力します。

印刷についての詳細は、Sun WorkShop オンラインヘルプを参照してください。

エディタの選択

エディタを選択するには、メインウィンドウの「ファイル」メニューから「オプション」コマンドを選択し、「テキストエディタのオプション」ダイアログボックスでソースコードの編集に使用するエディタを選択します。`vi`、`mule` または `xemacs` のエディタが使用可能です。

注 – vi と xemacs は、ループツールと同じディレクトリ(通常は /opt/SUNWspro/bin) にインストールされます (エディタがまだシステムにインストールされていない場合に限ります) 。mule の場合は、ユーザー自身がインストールする必要があります。いずれの場合も、使用するエディタは、ループツールが検出できるように検索パス上のディレクトリに存在しなければなりません。たとえば、mule をシステム上の /usr/local に置く場合、PATH 環境変数にはこのパスが指定されていなければなりません。

エディタの選択の詳細は、Sun WorkShop のオンラインヘルプの「テキストエディタの選択」を参照してください。

ソースコードの編集とヒント

メインウィンドウ上 (123 ページの図 7-1) でループを選択すると、以下の 2 つの動作が行われます。

- ソースコードを編集するためのウィンドウが表示され (127 ページの図 7-3 参照) 、vi、xemacs、mule の各 エディタが使用可能となります。

  vi の詳細については、vi(1) のマニュアルページを参照してください。また xemacs および mule については、オンラインヘルプが用意されています (「ヘルプ」 ボタンをクリックしてください) 。

  Sun WorkShop の vi エディタには、「バージョン」という特殊なメニューがあります。このメニューでは、SCCS (ソースコード管理システム) ユーティリティを使用して、ファイルを共有化できます。詳細については、ループツールのオンラインヘルプおよび sccs(1) のマニュアルページを参照してください。

- 選択したループに関する 1 つあるいは複数のヒントを含む「ヒント」ウィンドウが表示されます。このウィンドウの「ヘルプ」ボタンを押すと、WorkShop オンラインヘルプのコンパイラヒントに関連する内容が表示されます。ヒントの詳細については、133 ページの「コンパイラヒント」を参照してください。

図 7-3 には、ループが選択された状態の xemacs エディタウィンドウと、コンパイラヒントの説明が表示されたヒントウィンドウを示します。

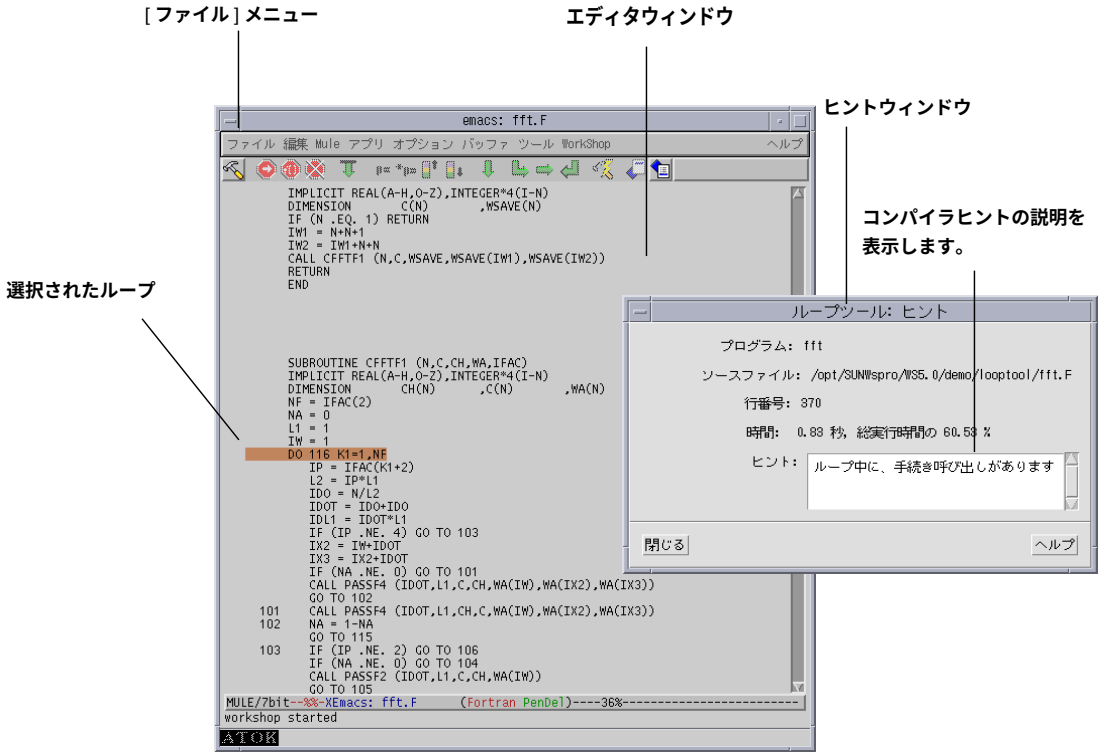

図 7-3　テキストエディタとヒントウィンドウ

注意 – ソースコードを編集すると、ループツールで表示される行番号がソースの行番号と異なる場合があります。編集済みのソースをいったん保存し、これを再コンパイルしてから最新の実行可能ファイルとともにループツールを起動すると、新しいループ情報が表示され、行番号が一致するようになります。

ループレポートの起動

ループレポートの起動時に、プログラム名を入力します。`loopreport` の後ろに検証したいプログラム名 (実行可能ファイル) を入力してください。

```
% loopreport program
```

プログラム名を指定せずにループレポートを起動することも可能です。ただし、プログラム名を指定せずにループレポートを起動する場合、ループレポートは現在の作業ディレクトリを検索して、a.out という名前のファイルを探します。

```
% loopreport > a.out.loopreport
```

出力をファイルにリダイレクトしたり、パイプによってほかのプログラムへと受け渡すことも可能です。

```
% loopreport program > program.loopreport
% loopreport program | more
```

タイミングファイル

ループレポートはプログラムに関連したタイミングファイルを一緒に読み込みます。タイミングファイルは、-zlp オプションを使用する場合に作成され、ループに関する情報を含みます。通常、タイミングファイルには *program*.looptimes という形式の名前が付けられ、プログラムと同じディレクトリに収められます。

しかし、 タイミングファイルの位置を指定する方法も 4 種類ほど用意されています。ループレポートは、以下に示すルールに従ってタイミングファイルを選択します。

- タイミングファイルがコマンド行で指定された場合、ループレポートはそのファイルを使用します。

```
% loopreport program newtimes > program.loopreport
```

- コマンド行オプション -p が使用された場合、ループレポートは -p によって指定されるディレクトリからタイミングファイルを探します。

```
% loopreport program -p /home/timingfiles > program.loopreport
```

- 環境変数 LVPATH が設定されている場合、ループレポートは LVPATH で指定されるディレクトリからタイミングファイルを探します。

```
% setenv LVPATH /home/timingfiles
% loopreport program > program.loopreport
```

- ループレポートはループ統計情報のテーブルを標準出力である stdout に書き込みます。この出力はファイルにリダイレクトしたり、パイプによってほかのコマンドに受け渡すことも可能です。

```
% loopreport program > program.loopreport
% loopreport program | more
```

図 7-4 に、ループレポートの例を示します。

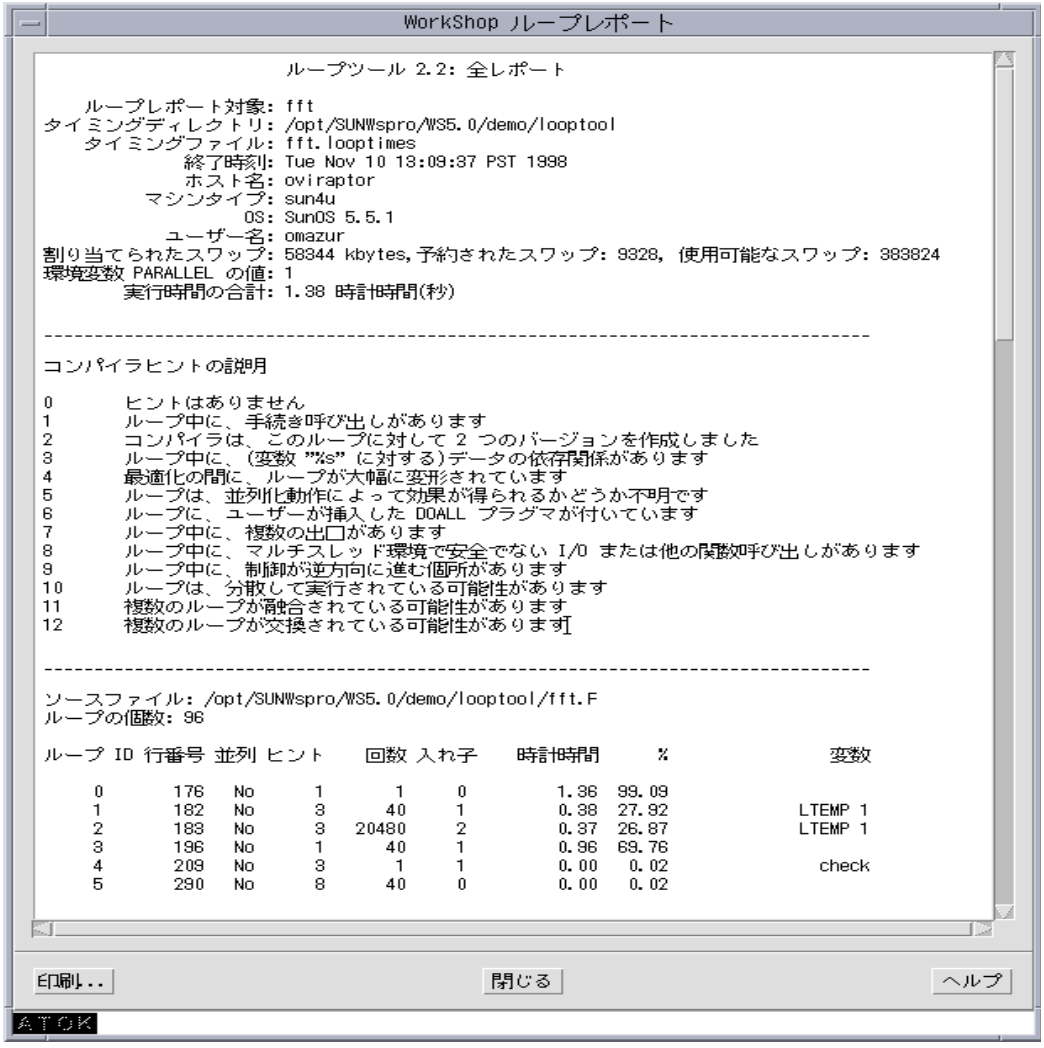

図 7-4　ループレポートの例

ループレポートのフィールド

以下の説明は、ループツールの「レポートの作成」出力およびループレポートの出力にもそのままあてはまります。

ループレポートには、以下の情報が含まれています。

- ループ ID

コンパイル時にコンパイラによって割り当てられる任意の番号。これはループとの対話には便利な内部的なループ ID で、プログラムとの実質的な関連はありません。

- 行番号

  ソースファイルにおけるループの先頭ステートメントの行番号。

- 並列

  Yes ならば、ループが並列化の対象としてマークされていることを示し、No ならば、そうでないことを示します。

- ヒント

  「コンパイラヒントの説明」リストのヒントテキストに対応した番号。

- 回数

  上位からそのループへと入り込む回数。これは、ループが実行される総数であるループ反復回数とは異なります。たとえば、以下のコードは Fortran で書かれた 2 つのループです。

```
do 10 i=1,17
    do 10 j=1,50
    ...some code...
    10 continue
```

  最初のループには 1 度だけ入り、ループは 17 回反復されます。2 番目のループには 17 回入り込み、ループは 17*50 = 850 回反復されます。

- 入れ子

  ループの入れ子のレベル。ループがトップレベルのループならば、入れ子のレベルは 0 です。ループがほかのループの子ループならば、レベルは 1 となります。

  たとえば、以下に示す C のコードでは、`i` ループはレベル 0、`j` ループはレベル 1、`k` ループはレベル 2 となります。

```
for (i=0; i<17; i++)
    for (j=0; j<42; j++)
        for (k=0; k<1000; k++)
            do something;
```

- 時計時間

  プログラム全体に対して、このループを実行するために費やされる経過時計時間 (待ち時間) の総計。外側のループの経過時計時間には、内側のループの経過時計時間が含まれます。以下に例を示します。

```
for (i=1; i<10; i++)
    for (j=1; j<10; j++)
        do something;
```

  たとえば、上記の例で、外側のループ (iループ) に割り当てられる時間が 10 秒だとしたら、内側のループ (j ループ) に割り当てられる時間は 9.9 秒となります。

- % (割合)

  ループの実行に費やされる時計時間として計測されたプログラム実行時間全体におけるパーセンテージ。時計時間においては、外側のループにはそこに含まれるループに費やされる時間も含まれます。

- 変数

  ループにおいてデータ依存を生じる変数の名前。このフィールドは、コンパイラヒントに、ループがデータ依存の弊害を受けていると示される場合のみ表示されます。あるループ内の繰り返しの中で計算される値がほかのループでも使用されるなどの理由で、ループの並列化が安全に行われない場合などに、データ依存の問題は発生します。データ依存の例を以下に示します。

```
do i = 1, N
    a(i) = b(i) + c(i)
    b(i) = 2 * a(i + 1)
end do
```

上記のループ例が並列処理で実行される場合、a(2) の値を基に b(1) の値を再計算する反復 1 は、a(2) の再計算を終えたばかりの反復 2 の後に実行される可能性が出てきます。つまり、ループが並列処理されなかった場合とは異なり、b(1) の値は、オリジナルの値ではなく、新しい a(2) の値によって決まってしまいます。

コンパイラヒント

ループツールとループレポートは、特定のループに対して適用された最適化についてのヒント、そして、なぜ並列化が行われなかったかについてのヒントを提示します。最適化の間にコンパイラが生成するヒントは、わかりにくい場合もあります。こうしたヒントは全体の文脈の中で理解されるべきものであり、特定のループに対して生成されるコードについての絶対的な事実ではありません。ただし、コンパイラがループの並列化を含めたより一層の最適化を実行できるように、コードを変換する方法についての重要な指針となることも少なくありません。

並列化の上で参考になる説明とヒントについては、『 Fortran ユーザーズガイド』を参照してください。

0. ヒントはありません。

このループに適用されるヒントはありません。これは、その他のヒントがまったく適用されないという意味ではなく、単にコンパイラから導かれるヒントがないということです。

1. ループ中に手続き呼び出しがあります。

ループ中にマルチスレッドで動作すると安全でない手続き呼び出しが含まれているので、並列化できません。このようなループを並列化すると、ループの複数のコピーによって関数呼び出しが同時にインスタンス化され、この関数に固有の変数や戻り値に悪影響を与えたり、関数の目的を無効にする可能性があります。このループ中の手続き呼び出しがマルチスレッドで使用しても安全なことがわかっている場合は、ループ本体の前に `DOALL`プラグマを追加することにより、このループを無条件に並列化するよう指示できます。たとえば、`foo` がマルチスレッドで使用しても安全な関数呼び出しである場合、`c$par DOALL` を挿入することによって、強制的な並列化を指定できます。

```
c$par DOALL
     do 19 i = 1, n*n
        do 29 k = i, n*n
            a(i) = a(i) + x*b(k)*c(i-k)
            call foo()
 29     continue
 19  continue
```

コンパイラは -parallel または -explicitpar を指定してコンパイルした場合に限り、DOALL プラグマを解釈します。-autopar を指定してコンパイルすると、DOALLプラグマは無視されます。

2. コンパイラはこのループに対して 2 つのバージョンを生成しました。

コンパイル時には、ループの中に並列化する意味のある要素が含まれているかどうか、コンパイラは判断できません。コンパイラは、逐次形式と並列形式の 2 つのバージョンのループを生成します。実行時の検査によって、どちらのバージョンを使用するかを決定します。この実行時検査は、ループの繰り返し値を検査して、ループの処理量を判断します。

3. 変数 *list* によってこのループ内でデータ依存性が発生する

ループ中の変数が、前回の繰り返しに使用された変数の値による影響を受けます。たとえば、次のとおりです。

```
do 99 i=1,n
    do 99 j = 1,m
    a[i, j+1] = a[i,j] + a[i,j-1]
99 continue
```

これは、あくまで説明の便宜上、意図的に考えた例にすぎません。このような単純なループの場合は、単にオプティマイザによって、内側と外側のループを入れ替えて、内側のループを並列化できますが、この例は、「ループ伝播データ依存性」と呼ばれるデータの依存関係の概念をわかりやすく示したものでもあります。

コンパイラは、ループ伝播データ依存性の原因となっている変数の名前を特定できる場合があります。プログラムを変更して、この種の依存関係を取り除く (または最小限に抑える) ことにより、コンパイラはさらに積極的な最適化を実行できるようになります。

4. 最適化の間に、ループが大幅に変形されています。

コンパイラがこのループを最適化した結果、生成されたコードとソースコードの対応が取れなくなる場合があります。このため、行番号の整合性がなくなっている可能性があります。ループを大幅に変更する可能性のある最適化の例として、ループの分散化、融合、交換 (ヒント 10、ヒント 11、および ヒント 12 を参照) などがあります。

5. ループは、並列化動作によって効果が得られるかどうか不明です。

コンパイラは、ループが並列化動作によって効果が得られるかどうかを、コンパイル時には判断できません。このヒントが表示されたループは、場合によって、「並列」のラベルが付けられることがあります。これは、コンパイラが 2 つのバージョンのループを生成 (ヒント 2を参照) し、並列バージョンと逐次バージョンのいずれを使用するかは実行時に決定するという意味です。

コンパイラヒントは、ループの並列化の有無を示すフラグも含め、すべてコンパイル時に生成されるので、「並列」とラベル付けされたループが実際に並列的に実行されるかどうかについて確証はありません。

6. ループにユーザーが挿入した DOALL プラグマが付いています。

このループは、コンパイラに対する DOALL プラグマの指示によって並列化されています。このヒントは、明示的に並列化を指定したループを簡単に識別する場合に利用できます。

DOALL プラグマは、-parallel または -explicitpar を指定してコンパイルする場合に限り、コンパイラにより解釈されます。-autopar を指定してコンパイルすると、コンパイラは DOALL プラグマをすべて無視します。

7. ループ中に複数の出口があります。

ループの中に、通常のループの出口以外に、GOTO やループを終了するその他の分岐が使用されています。したがって、コンパイラではループの実行時の動作が予測できないため、ループの並列化は危険と判断されます。

8. ループ中にマルチスレッドで安全でない I/O または他の関数呼び出しがあります。

これは、ヒント 1 に似ています。ただし、ヒント 8 がマルチスレッドで動作すると安全でない「入出力」に適用され、ヒント 1 はマルチスレッドで動作すると安全でない「関数の呼び出し」に適用される点が異なります。

9. ループ中に制御が逆方向に進む箇所があります。

ループの中に、GOTO や、逆方向に流れたりループ本体から抜けたりするような制御フローが使用されています。すなわち、コンパイラはループの内部にある文が、すでに実行済みのコードに戻るフローになっていると解釈します。複数の出口が存在するループの場合と同様に、このループを並列化することは安全ではありません。

逆方向の制御フローを取り除く (または最小限に抑える) ことにより、コンパイラはより積極的な最適化を実現できるようになります。

10. ループは分散して実行されている可能性があります。

ループの内容が、何度か繰り返すうちに分散している可能性があります。すなわち、ループを並列化できるようにコンパイラによってループ本体が書き換えられている可能性があります。ただし、この書き換えはオプティマイザの内部表現で使用する言語によって行われるので、もとのソースコードと書き換えたコードを対応させるのはきわめて困難になります。このため、分散されたループについてのヒントでは、実際のソースコードにある行番号には一致しない行番号が示される可能性があります。

11. 複数のループが融合されている可能性があります。

並列化の効果を高めるよう大きなループにするため連続した 2 つのループが 1 つにまとめられています。また、この場合も、ソースの行番号が誤っている可能性があります。

12. 複数のループが交換されている可能性があります。

内部ループと外部ループのインデックスが入れ替えられています。これは、データの依存関係を内部ループから可能な限り遠い位置に置き、この入れ子ループを並列化するための処置として行うものです。深いレベルで入れ子にしたループの場合は、3 つ以上のループで交換が行われている可能性があります。

コンパイラの最適化とループへの影響

コンパイラヒントの説明からもわかるとおり、最適化コードとソースコードを対応させるのは難しい場合があります。コンパイラが出力する情報が、可能な限りソースコードに近い形式で表示されるほうが望ましいのは明らかです。残念ながら、コンパイラオプティマイザは、プログラムを内部表現によって「読み取る」ため、もとのソースコードに対応させようとしても、それほど効果はありません。

次に、混乱の原因となりやすい最適化の手法を紹介します。

インライン化

インライン化とは、 最適化レベル `-O4` に限り、1 つのファイルに含まれる関数だけに適用される最適化の 1 つです。すなわち、あるファイルに 17 個の Fortran 関数が含まれている場合、16 個の関数を残りの 1 つの関数に展開でき、`-O4` レベルでコンパイルされていれば、その 16 個の関数のソースコードを 1 つの関数の本体にコピーします。これ以上の最適化を適用すると、ソースコードのどの行番号がどの最適化の対象となっているかが特定できなくなります。

コンパイラヒントの説明が不明確と思われる場合は、`-O3 -parallel -Zlp` を付けてコンパイルすると、コンパイラが関数をインライン化する前の段階で、ループに関するコンパイラからの指摘をより詳しく確認できます。

特に実体のないループ (すなわちコンパイラが、実在することを前提とするループで、実際のソースコード中には存在しないもの) は、インライン化の対象となっている可能性があります。

ループの変形：展開、詰め込み、分割、および入れ替え

コンパイラは、ループ本体を大きく書き換えるようなループの最適化を多数実行します。 具体的には、ループの最適化、展開、詰め込み、分割、入れ替えなどの操作があります。

ループツールとループレポートは、可能な限り、意味のあるヒントを提供しようとしますが、最適化済みコードとソースコードを対応させる上での根本的な問題があるため、ヒントの内容が誤解を招くこともあります。

逐次ループの内部に入れ子にした並列ループ

逐次ループの中に並列ループが入れ子にされている場合、各ループを繰り返すごとに時計時間を使用するという条件があるため、ループツールおよびループレポートが報告する実行時間の情報が誤解を招く場合があります。内部ループが並列化されていると、ループの一部を並列的に繰り返す場合であっても、反復ごとの時計時間が確保されます。

これに対して、外部ループには、入れ子にした並列ループ全体の実行時間しか割り当てられません。すなわち、内部ループを並列的にインスタンス化した場合の最も長い時間に相当します。このように計測方法が二重になる結果、内部ループに比べ、外部ループの消費時間が短くなるという、「外部ループの変則性」という問題を生じることになります。

付録 A

従来のプロファイルツール

この章では、プログラムの動作時間を測定したり、解析対象となるパフォーマンスデータを得るための標準的なユーティリティについて解説します。`prof` および `gprof` は、SPARC プラットフォームおよび Intel プラットフォームの Solaris 2.6、7 および 8 に付属のプロファイルツールです。また、`tcov` はコードカバレージツールです。

注意 – 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) を追跡する場合は、従来のプロファイルツールを使ってください。プログラムでの時間消費を詳細に解析する場合は、標本コレクタおよび標本アナライザを使ってより正確な情報を取得することができます。これらのアプリケーションの使用については、第 3 章および第 4 章を参照してください。

従来のプロファイルツールの一部は、C 以外のプログラミング言語で記述されたモジュールでは動作しません。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。

この章は、以下の項目から構成されています。

- 基本概念
- prof によるプログラム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
- プロファイルに対応した共用ライブラリの生成

基本概念

prof、gprof、および tcov は Sun WorkShop の開発環境を拡張し、パフォーマンスデータの収集および利用を可能にします。

- prof は単純なファイル形式のプログラムプロファイルを生成します。
- gprof は呼び出しグラフプロファイルを作成します。
- tcov は、どの行がどのくらいの頻度で使用されたかを示す文レベルの情報をソースファイルのコピーに生成します。

表 A-1 では、これらの標準パフォーマンスプロファイルツールで生成される情報を説明しています。

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	各関数に制御が渡る正確な回数とともに、プログラムによって使用される CPU 時間の統計的プロファイルを生成します。このツールは Solaris オペレーティングシステム環境に付属しています。
gprof	各関数に制御が渡る正確な回数、およびプログラムの呼び出しグラフの各弧 (呼び出す側と呼び出される側の組み合わせ) 上で制御が受け渡しされる回数とともに、プログラムによって使用される CPU 時間の統計的プロファイルを生成します。このツールは Solaris オペレーティングシステム環境に付属しています。
tcov	プログラムの各文が実行される正確な回数を生成します。tcov には、オリジナルの tcov と、拡張された tcov の 2 つのバージョンがあります。これら 2 つのバージョンは、出力で使用される生データを生成する実行時サポートが異なります。

prof によるプログラムプロファイルの生成

prof はプログラムによって使用される CPU 時間の統計プロファイルを生成し、プログラム中の各関数へ制御が渡される回数をカウントします。さらに詳細なデータは、呼び出しグラフプロファイルおよびコードカバレージツールによって提供されます。

prof を利用するための手順は以下の 3 段階となっています。

1. prof 用のプログラムをコンパイルします。
2. プログラムを実行して、プロファイルデータを作成します。
3. prof を使ってデータを要約したレポートを生成します。

prof によるプロファイリング用のプログラムをコンパイルするには、コンパイラに対して -p オプションを使用してください。たとえば、index.assist.c という名前の C のソースファイルをプロファイリング用にコンパイルするには、以下のコンパイラコマンドを使用します。

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

コンパイラは index.assist という名前のプログラムを作成します。

次に、この index.assist プログラムを実行してください。プログラムを実行するたびに、プロファイルデータが mon.out と呼ばれるファイルに送られます。プログラムを実行するたびに、新しい mon.out が生成され、古いバージョンは上書きされます。

最後に prof コマンドを使ってレポートを生成します。

```
% index.assist
% ls mon.out
mon.out
% prof index.assist
```

出力例

prof の出力は、以下の例のようになります。

%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

.
. (以下の出力はそれほど重要ではありません)

prof 出力の例

この表示例は、プログラムで一番多く時間を消費しているのが compare_strings() ルーチンであることを示しています。次は、 _strlen() です。このプログラムを改良するには、compare_strings() 関数を見直します。

プロファイルサンプルの実行結果は、以下に示す列のタイトルで表示されています。

- %Time — プログラムのこのルーチンによって消費される CPU 時間の合計におけるパーセンテージ。
- Seconds — この関数によって占められる CPU 時間の合計。
- Cumsecs — この関数およびその上位にリストされている関数によって占められる CPU 時間の総合計 (秒) 。
- #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るたびに消費する平均ミリ秒数。
- Name — ルーチンの名前

このプロファイルデータからはいったいどんな結果が導き出されるのでしょうか。compare_strings 関数は、プログラムの総時間の約 20% を消費しています。そこで、この index.assist を改良するには、compare_strings() が使用しているアルゴリズムを改良するか、compare_strings() の呼び出し回数を減らすという方法が考えられます。

この単純な呼び出しグラフからは compare_strings() がかなり強い再帰性を持つかはわかりませんが、次の項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によって、こうした情報も得ることができます。ここで紹介しているケースの場合は、アルゴリズムの改良によって呼び出し回数も減少します。

注 – Solaris 2.6、7 および 8 では、複数の CPU を使用するプログラムに対して CPU 時間のプロファイルは正確ですが、呼び出し回数はロックされていないため、関数の呼び出し回数の正確さには影響があるかもしれません。

---

gprof による呼び出しグラフプロファイルの生成

prof の単純なプロファイルは、パフォーマンス改善のために価値のあるデータを提供しますが、呼び出しグラフプロファイルを利用すれば、さらに詳細な解析を実行できます。呼び出しグラフプロファイルは、どのモジュールがほかのモジュールから呼び出されているか、どのモジュールがほかのモジュールを呼び出しているかを識別する一覧表を表示します。ときには、呼び出しをすべて削除することで、パフォーマンスが改善される場合もあります。

注 – gprof は関数内の時間を、各弧が往復する回数に比例して、呼び出し側に割り当てます。しかし、すべての呼び出しで同じ動作をするわけではないため、こうした動作は誤った仮定に陥る可能性もあります。例は、12 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 タイムの統計的プロファイルを生成し、関数に制御が渡される回数を数えます。gprof はさらに、プログラムの呼び出しグラフにおいて各弧上で制御が受け渡される回数も数えます。弧とは、呼び出す側と呼び出される側の組み合わせを指します。

注 – Solaris 2.6、7 および 8 では、複数の CPU を使用するプログラムに対してCPU 時間のプロファイルは正確ですが、呼び出し回数はロックされていなため、関数の呼び出し回数の正確さには影響があるかもしれません。

---

gprof を利用するための手順は以下の 3 段階となっています。

1. gprof 用のプログラムをコンパイルします。
2. プログラムを実行して、プロファイルデータファイルを作成します。
3. gprof を使ってデータを要約したレポートを生成します。

呼び出しグラフプロファイリング用のプログラムをコンパイルするには、C コンパイラに対しては -xpg オプションを、Fortran コンパイラに対しては -pg オプションを使用してください。以下に例を示します。

```
% cc -xpg -o index.assist index.assist.c
```

この結果 index.assist プログラムが実行可能になります。gprof 用にコンパイルされたプログラムを実行するたびに、呼び出しグラフプロファイルデータが gmon.out という名前のファイルに送られます。そのプログラムを実行するたびに、新しい gmon.out が作成されます。

プロファイルの結果レポートを生成するには、gprof コマンドを使用します。gprof の出力はかなり大量になる可能性があります。そのため、結果をファイルにリダイレクトすると、レポートはかなり読みやすくなるでしょう。gprof の出力を g.output というファイルにリダイレクトするには、以下の一連のコマンドを使用してください。

```
% index.assist
% ls gmon.out
gmon.out
% gprof index.assist > g.output
```

gprof の出力は以下の 2 つの主要項目から構成されます。

- プロファイリングの実行結果のフラグメントを示す全呼び出しグラフプロファイル。
- prof コマンドのプロファイル結果に類似した、単純なプロファイル。

gprof の出力には、要約の各部分がどのような意味を持っているかの説明も含まれます。gprof はさらに標本収集の細分性を示します。

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

ここでは、4 byte(s) とは 1 命令の解像度を意味しています。つまり、各標本が全実行時間の 0.07 パーセントであり、CPU 時間が 10 ミリ秒であることを意味します。

以下の例は、呼び出しグラフのプロファイルの一部です。

index	%time	self	descendents	called / total parents called+self called / total children	name	index
---						
		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 `index.assist` プログラムの入力ファイルのデータには 761 行が含まれていると想定すると、以下の結論が導き出されます。

- `fgets()` は 762 回呼び出されています。`fgets()` の最後の呼び出しは、end-of-file を返します。

- insert_index_entry() 関数は、main() から 760 回呼び出されています。
- insert_index_entry() 関数が main() から 760 回呼び出されていることに加えて、insert_index_entry() 関数は自分自身を 10,392 回呼び出しています。つまり、insert_index_entry() の再帰性はかなり強いということです。
- compare_entry() (insert_index_entry() から呼び出されている) は 11,152 回呼び出され、これは 760+10,392 回と等しくなります。
  insert_index_entry() が呼び出されるごとに、compare_entry() の呼び出しが 1 回存在するということで、これは論理的に矛盾がありません。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ことを疑ってみるべきでしょう。
- insert_page_entry() は、トータルで 820 回呼び出されています。プログラムがインデックスノードを構築している間に main() から 760 回呼び出され、insert_index_entry() から 59 回呼び出されています。この頻度は、59 個の重複するインデックスエントリが存在し、その結果、それらのページ番号エントリはインデックスノードによってチェーンにリンクされていることを示します。 重複するインデックスはその後解放されます。そのため、free() は 59 回呼び出されています。

tcov による文レベルの解析

tcov は、プログラムの実行方法に関する行単位の情報を提供します。tcov はソースファイルのコピーを作成し、そこにどの行がどれくらいの頻度で使用されているかを示す注釈を挿入します。また、基本的なブロックについての情報も集約します。tcov は時間ベースのデータは生成しません。

tcov を利用するための手順は以下の 3 段階となっています。

1. tcov の実験データ作成用に、プログラムをコンパイルします。
2. 実験を実行します。
3. tcov を使って、プログラムの各文の実行カウントの集計を生成します。

tcov 用のコンパイル

コードカバレージ用にプログラムをコンパイルするには、C コンパイラに対して -xa オプションを使用します。index.assist という名前のプログラムを例として使用する場合、以下のコマンドによって、tcov 用のコンパイルを行ってください。

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

C++ または f77 コンパイラに対しては、-a コンパイラオプションを使用します。

C コンパイラは index.assist.c に存在する基本ブロックについてのデータベースエントリを含む index.assist.d ファイルを生成します。プログラム index.assist が実行され、終了した時点で、コンパイラは index.assist.d ファイルを更新します。

注 – tcov は C および C++ プログラムとは動作しますが、#line または #file 指示が含まれるファイルはサポートしません。tcov は、#include ヘッダーファイルのコードのテストカバレージ解析も行うことができません。-xa (C)、-a (その他のコンパイラ)、+d (C++) オプションを指定してコンパイルされたアプリケーションは通常よりも実行速度が遅くなります。+d オプションは C++ のインライン関数の拡張を禁止するため、各実行時の .d ファイルの更新にかなりの時間がかかります。

---

index.assist.d ファイルは、環境変数 TCOVDIR が指示する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、index.assist.d はカレントディレクトリに作成されます。

index.assist.c のコンパイルが済んだら、index.assist を実行してください。

```
% index.assist
% ls *.d
index.assist.d
```

それでは、tcov を実行して、プログラムの各文の実行カウントの集計結果が含まれるファイルを生成してください。tcov は index.assist.d ファイルを使って、コードの注釈リストを含む index.assist.tcov ファイルを生成します。この出力には、各ソース文が実行された回数が示されます。ファイルの末尾には、短い要約が付いています。

```
% tcov index.assist.c
% ls *.tcov
index.assist.tcov
```

次に、index.assist のモジュールの 1 つから、C コードの一部分 を示します。問題となっているモジュールは、呼び出し頻度の高い insert_index_entry 関数です。

```
        struct index_entry *
11152-> insert_index_entry(node, entry)
        structindex_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);
            }
        }
```

C コードの左の数値は、各文が実行された回数を表しています。insert_index_entry 関数は11,152回呼び出されます。

tcov は、index.assist.tcov のファイルの末尾に以下のような集計情報を追加します。

```
            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
```

コードカバレージ解析用にコンパイルされたプログラムは、(異なる入力を基にしての) 繰り返し実行が可能です。つまり、プログラムに対して `tcov` を繰り返し使用し、各実行時の動作を比較できるということです。

tcov によるプロファイルに対応した共用ライブラリの生成

tcov によるプロファイルに対応した共用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たライブラリの代わりに使用することが可能です。共有可能なライブラリを生成する場合は、-xa オプション (C 用) または -a オプション (その他のコンパイラ) を使用してください。以下に例を示します。

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

このコマンドは、共用ライブラリに tcov プロファイリングサブルーチンを組み込み、ライブラリのクライアントが再リンクしなくても済むようにします。ライブラリを使用するプログラムもまたプロファイリング用にリンクされる場合、そのプログラムにリンクされた tcov プロファイリングサブルーチンが、プログラムと共用ライブラリの両方で使用されます。

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の更新に、単純なファイルロックメカニズムを使用します。tcov はこの目的のために単独ファイル /tmp/tcov.lock を使用します。そのため、-xa (C コンパイラ) または -a (その他のコンパイラ) を使ってコンパイルされた実行可能ファイルは、1 つのシステムで一度に 1 つしか実行すべきではありません。-xa または -a オプションを使ってコンパイルされたプログラムの実行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。

プログラムが tcovによるプロファイリング用にリンクされると、-xa または -a オプションを使ってコンパイルされたファイルは自動的にプロファイルツールのサブルーチンを呼び出します。プログラムの終了時点で、これらのサブルーチンは、ファイル xyz.f の実行時に収集された情報と、ファイル xyz.d に保存された既存のプロファイリング情報とを組み合わせます。プロファイル化されたバイナリファイルを同時に実行している人々にこの情報を変更されないようするため、更新期間中は xyz.d 用にロックファイル xyz.d.lock が作成されます。xyz.d またはそのロックファイルのオープンまたは読み取り時にエラーが存在する場合、あるいは、実行時の情報と保存されている情報との間に不整合が存在する場合、xyz.d に保存されているデータは更新されません。

xyz.d の編集または再コンパイルは、xyz.d のカウンタの数を変える可能性があります。古いプロファイル化されたバイナリの実行の際に、こうした変化が検出されます。

プロファイル化されたバイナリを実行する人が多すぎると、ロックが得られない可能性もあります。その場合は、数秒の遅れの後に、次のようなエラーメッセージが表示されます。

```
tcov_exit: Failed to create lock file
'/tmp_mnt/net/rbbb/export/home/src/newpattern/foo.d.lock'
for coverage data file
'/tmp_mnt/net/rbbb/export/home/src/newpattern/foo.d'
after 5 tries. Is somebody else running this binary?
```

そして、保存されている情報の更新は行われません。このロックはネットワークを通じて安全に機能します。ロックはファイル単位で行われるため、ほかのファイルは正しく更新されます。

プロファイリングサブルーチンは、アクセス不可能となっている自動マウントファイルシステムを処理しようと試みます。それでも、異なるマシン上で、カバレージデータファイルを含むファイルシステムに異なる名前がマウントされていたり、プロファイル化されたバイナリを実行するユーザーがカバレージデータファイルまたはそれを含む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。すべてのディレクトリには、統一化された名前を付け、バイナリを実行する可能性のあるユーザー誰もが書き込み可能な状態にしておいてください。

tcov の実行時ルーチンで発生するエラー

以下のエラーメッセージが、tcov ランタイムルーチンによって報告されることがあります。

- このバイナリを実行するユーザーには、カバレージデータファイルに対する読み取り権および書き込み権がありません。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も、こうした問題が起こります。

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

- このバイナリを実行するユーザーには、カバレージデータファイルを含むディレクトリに書き込み権がありません。カバレージデータファイルを含むディレクトリがバイナリを実行するマシンにマウントされていない場合も、こうした問題が起こります。

```
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。カバレージデータファイルが更新されている間に、マシンがクラッシュしたときも、ロックファイルが放置され、こうした問題は起こります。万一、クラッシュが起こった場合には、2 つのファイルのうちサイズの大きい方をクラッシュ後のカバレージデータファイルとして使用してください。ロックファイルは手動で削除してください。

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

- 利用可能なメモリがなく、標準 I/O パッケージが動作できません。この時点では、カバレージデータファイルの更新はできません。

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

- ロックファイル名は、カバレージデータファイル名にさらに 6 文字追加されます。そのため、派生ロックファイル名が規則違反とな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 プロファイル用の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。古いバイナリは一定サイズのカバレージデータファイルを予期しますが、編集によってそのサイズはしばしば変更されます。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、カバレージファイルは空白、あるいは壊れた状態で表示されます。

```
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、プログラムの実行方法についての行単位の情報を提供します。拡張 tcov は、どの行がどれくらいの頻度で使用されているかを示す注釈が付けられたソースファイルのコピーを作成します。また、基本ブロックに関する要約情報も提供します。拡張 tcov は、C および C++ のソースファイルに対して動作します。

拡張 tcov の利点

拡張 tcov では、オリジナル tcov のいくつかの欠点が改善されています。

- C++ に対するサポートがより強化されています。
- 拡張 tcov は、#include ヘッダーファイルに含まれるコードをサポートし、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る問題をを修正しています。
- 拡張 tcov のランタイムは、オリジナルの tcov のランタイムよりも効率的です。
- 拡張 tcov は、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向けにサポートされています。

拡張 tcov 用のコンパイル

拡張 tcov を使用するには、オリジナルの tcov と同様の以下の操作を行います。

1. 拡張 tcov の実験データ用にプログラムをコンパイルします。
2. 実験を実行します。
3. tcov を使って、結果を解析します。

オリジナルの tcov では、-xa オプションを使ってコンパイルを行いました。拡張 tcov によってコードカバレージ用のプログラムをコンパイルするには、すべてのコンパイラに対して、-xprofile=tcov オプションを使用してください。index.assist という名前のプログラムを例として使用する場合、拡張 tcov を使ったコンパイルは以下のようなコマンドになります。

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

拡張 tcov は、tcov とは異なり、.d ファイルを作成しません。プログラムが実行されるまで、カバレージデータファイルは作成されません。カバレージ解析用にコンパイルされた各モジュールごとに 1 つのファイルではなくて、プログラム実行後に 1 つのカバレージデータファイルが作成されます。

index.assist.c のコンパイルの後、index.assist を実行して、プロファイルデータを生成してください。

```
% index.assist
% ls -dF *.profile
index.assist.profile/
% ls *.profile
tcovd
```

デフォルトでは、tcovd ファイルが保存されるディレクトリの名前は、実行可能ファイルの名前から派生して付けられます。また、そのディレクトリは、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 (オリジナルの tcov は、モジュールがコンパイルされたディレクトリに .d ファイルを作成します)。

tcovd ファイルが保存されるディレクトリはまた、「プロファイルのバケツ」とも呼ばれています。プロファイルバケツは、環境変数 SUN_PROFDATA を使用することで、変更できます。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、この操作は有効です (たとえば、違う名前のシンボリックリンクから実行可能ファイルが実行された場合など)。

プロファイルバケツが生成されるディレクトリを変更することも可能です。実行ディレクトリとは別のディレクトリを指定するには、環境変数 SUN_PROFDATA_DIR を使ってパスを指定してください。この変数には、相対パス名、絶対パス名のどちらでも指定できます。相対パス名は、プログラムの実行完了時のカレントの作業ディレクトリに対する相対的な位置となります。

TCOVDIR は、下方互換性を維持するため SUN_PROFDATA_DIR と同じ意味を持ち、環境変数としてサポートされています。SUN_PROFDATA_DIR が設定されている場合、TCOVDIR が無視されます。TCOVDIR と SUN_PROFDATA_DIR の両者が設定されていると、プロファイルのバケツが生成されるときに、警告が表示されます。SUN_PROFDATA_DIR は、TCOVDIR よりも優先されます。これらの変数は、-xprofile=tcov オプションによってコンパイルされたプログラムの実行時に使用されます。さらに、tcov コマンドによっても使用されます。

注 – このスキーマはプロファイルフィードバックメカニズムによっても使用されます。

カバレージデータが作成されたら、生データをソースファイルに関連付けるレポートを生成できます。

```
% tcov -x index.profile index.assist.c
% ls *.tcov
index.assist.c.tcov
```

このレポートの出力は、前の例 (オリジナルの tcov のもの) と同じです。

プロファイルに対応した共用ライブラリの生成

拡張 tcov とともに使用するための共用ライブラリの生成は、以下の類似したコンパイルオプションを使って実行できます。

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

ファイルのロック

拡張 tcov は、単純なファイルロックメカニズムを用いてブロックカバレージデータファイルを更新します。拡張 tcov は、tcovd ファイルと同じディレクトリに生成された単独ファイルを使用します。そのファイル名は tcovd.temp.lock です。カバレージ解析用にコンパイルされたプログラムの実行が手動で終了された場合、ロックファイルは手動で削除しなければなりません。

ロックの競合が存在する場合、このロック方式は指数的遅延を行います。そして、5回試みた後、tcov ランタイムがロックを獲得できないと、tcov はあきらめ、その実行用のデータは失われます。この場合、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 用ディレクトリおよび環境変数

tcov 用にプログラムをコンパイルし、そのプログラムを実行すると、実行プログラムはプロファイルバケツを生成します。既にプロファイルバケツが存在している場合、プログラムはそのプロファイルバケツを使用します。プロファイルバケツが存在していない場合は、プロファイルバケツを生成します。

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指定します。プロファイル出力の名前と場所はデフォルトで決まっていますが、環境変数で指定することも可能です。

注 – tcov は、プロファイルフィードバックを集めるために使用したコンパイラオプション -xprofile=collect と -xprofile=use によって使用される同じデフォルト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の詳細については、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の名前は、実行可能ファイル名 + .profile となり、実行可能ファイルが実行されるディレクトリに生成されます。そのため、カレントディレクトリが /home/joe で、/usr/bin/xyz というプログラムを実行すると、デフォルト動作としては、/home/joe 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。

デフォルト設定を変更するための環境変数を以下に示します。

- SUN_PROFDATA

  実行時のプロファイルバケツの名前の指定に使用できます。SUN_PROFDATA_DIR も設定されている場合は、SUN_PROFDATA の値は SUN_PROFDATA_DIR の値に追加されます。

- SUN_PROFDATA_DIR

プロファイルバケツを含むディレクトリの名前の指定に使用できます。これは実行時および tcov コマンドにおいて使用されます。

- TCOVDIR

  TCOVDIR は、SUN_PROFDATA_DIR と同じ意味を持ち、下方互換性を維持するための環境変数としてサポートされています。SUN_PROFDATA_DIR が設定されている場合、TCOVDIR は無視されます。TCOVDIR と SUN_PROFDATA_DIR の両者が設定されていると、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、 -xprofile=tcov を使用してコンパイルされたプログラムの実行時にに使用され、さらに tcov コマンドによって使用されます。

デフォルト設定の変更

デフォルトの設定を変更するには、環境変数を使用してプロファイルバケツを変更します。

1. **環境変数 SUN_PROFDATA を使って、プロファイルバケツの名前を変更します。**
2. **環境変数 SUN_PROFDATA_DIR を使って、プロファイルバケツが置かれているディレクトリを変更します。**

この 2 つの環境変数はプロファイルバケツのデフォルトの保存場所と名前を変更します。保存場所と名前は別々に変更することができます。たとえば、SUN_PROFDATA_DIR の設定だけを選択すると、プロファイルバケツは SUN_PROFDATA_DIR に設定されたディレクトリに作成されますが、デフォルトの名前 (実行可能ファイル名 +.profile) が引き続きプロファイルバケツの名前として使用されます。

絶対パス名と相対パス名

プロファイルフィードバック用のコンパイル行上では、SUN_PROFDATA_DIR によって、絶対パス名 (「/」から始まる) と相対パス名の 2 種類のディレクトリの形式を指定できます。絶対パス名を使用する場合、プロファイルバケツはそのディレクトリに収められます。一方、相対パス名を使用する場合、プロファイルバケツは実行可能ファイルが実行されるカレントの作業ディレクトリに対して相対的な位置関係になります。

たとえば、カレントディレクトリが /home/joe で、SUN_PROFDATA_DIR が .. に設定された状態で /usr/bin/xyz というプログラムを実行すると、プロファイルバケツは /home/joe/../xyz.profile となります。環境変数に指定された値が相対的であるため、プロファイルバケツは /home/joe と相対的な位置関係となります。またこの例では、実行可能ファイル名を使用した、デフォルトのプロファイルバケツ名が使用されます。

TCOVDIR と SUN_PROFDATA_DIR

旧バージョンの tcov (-xa または -a フラグによるコンパイルによって使用可能となります) は、TCOVDIR という環境変数を使用していました。TCOVDIR は、ソースファイルと同じディレクトリではなく、tcov カウンタファイルが収められるディレクトリを指定しました。この環境変数との互換性を維持するため、新しい環境変数 SUN_PROFDATA_DIR は TCOVDIR 環境変数と同様に動作します。両方の変数が設定されている場合、警告が出力され、SUN_PROFDATA_DIR が、TCOVDIR よりも優先されます。

-xprofile=tcov

デフォルトでは、プロファイルバケツは、カレントディレクトリにある <argv[0]>.profile になります。

SUN_PROFDATA を設定している場合、プロファイルバケツは、どこに保存されていようと、 $SUN_PROFDATA になります。

SUN_PROFDATA_DIR を設定している場合、プロファイルバケツは、指定されたディレクトリに保存されます。

SUN_PROFDATA と SUN_PROFDATA_DIR はそれぞれ独立しています。いずれの環境変数も設定されていると、プロファイルバケツ名は、SUN_PROFDATA_DIR を使ってプロファイルバケツを見つけ出すことによって生成され、SUN_PROFDATA はそのディレクトリのプロファイルバケツの命名に使用されます。

UNIX のプロセスでは、プログラム実行中、カレントの作業ディレクトリが変更されることがあります。そのため、プロファイルバケツの生成に使用されるカレントの作業ディレクトリは、プログラムの実行終了時におけるカレントの作業ディレクトリとなります。まれですがプログラムの実行中にそのカレント作業ディレクトリを変更するような場合には、環境変数を利用して、プロファイルバケツが生成される場所を制御してください。

tcov プログラムについて

-xprofile=bucket オプションは、tcov プロファイル用に使用されるプロファイルバケツの名前を指定します。SUN_PROFDATA_DIR またはTCOVDIR が設定されている場合、その設定値がこの引数に付加されます。

索引

L

M

P

S

T

X

Z

あ

き

こ

ち

て

と

な

は

ひ

ふ

へ

ほ

ま

み

め

よ

ら

り

る

れ

ろ