Oracle® Developer Studio 12.5: パフォーマンスアナライザ

Part No: E71957
2016 年 6 月

ORACLE

Part No: E71957

Copyright © 2015, 2016, Oracle and/or its affiliates. All rights reserved.

このソフトウェアおよび関連ドキュメントの使用と開示は、ライセンス契約の制約条件に従うものとし、知的財産に関する法律により保護されています。ライセンス契約で明示的に許諾されている場合もしくは法律によって認められている場合を除き、形式、手段に関係なく、いかなる部分も使用、複写、複製、翻訳、放送、修正、ライセンス供与、送信、配布、発表、実行、公開または表示することはできません。このソフトウェアのリバース・エンジニアリング、逆アセンブル、逆コンパイルは互換性のために法律によって規定されている場合を除き、禁止されています。

ここに記載された情報は予告なしに変更される場合があります。また、誤りが無いことの保証はいたしかねます。誤りを見つけた場合は、オラクルまでご連絡ください。

このソフトウェアまたは関連ドキュメントを、米国政府機関もしくは米国政府機関に代わってこのソフトウェアまたは関連ドキュメントをライセンスされた者に提供する場合は、次の通知が適用されます。

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

このソフトウェアまたはハードウェアは様々な情報管理アプリケーションでの一般的な使用のために開発されたものです。このソフトウェアまたはハードウェアは、危険が伴うアプリケーション(人的傷害を発生させる可能性があるアプリケーションを含む)への用途を目的として開発されていません。このソフトウェアまたはハードウェアを危険が伴うアプリケーションで使用する際、安全に使用するために、適切な安全装置、バックアップ、冗長性(redundancy)、その他の対策を講じることは使用者の責任となります。このソフトウェアまたはハードウェアを危険が伴うアプリケーションで使用したことに起因して損害が発生しても、Oracle Corporationおよびその関連会社は一切の責任を負いかねます。

OracleおよびJavaはオラクル およびその関連会社の登録商標です。その他の社名、商品名等は各社の商標または登録商標である場合があります。

Intel、Intel Xeonは、Intel Corporationの商標または登録商標です。すべてのSPARCの商標はライセンスをもとに使用し、SPARC International, Inc.の商標または登録商標です。AMD、Opteron、AMDロゴ、AMD Opteronロゴは、Advanced Micro Devices, Inc.の商標または登録商標です。UNIXは、The Open Groupの登録商標です。

このソフトウェアまたはハードウェア、そしてドキュメントは、第三者のコンテンツ、製品、サービスへのアクセス、あるいはそれらに関する情報を提供することがあります。適用されるお客様とOracle Corporationとの間の契約に別段の定めがある場合を除いて、Oracle Corporationおよびその関連会社は、第三者のコンテンツ、製品、サービスに関して一切の責任を負わず、いかなる保証もいたしません。適用されるお客様とOracle Corporationとの間の契約に定めがある場合を除いて、Oracle Corporationおよびその関連会社は、第三者のコンテンツ、製品、サービスへのアクセスまたは使用によって損失、費用、あるいは損害が発生しても一切の責任を負いかねます。

ドキュメントのアクセシビリティについて

オラクルのアクセシビリティについての詳細情報は、Oracle Accessibility ProgramのWeb サイト(http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc)を参照してください。

Oracle Supportへのアクセス

サポートをご契約のお客様には、My Oracle Supportを通して電子支援サービスを提供しています。詳細情報は(http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info)か、聴覚に障害のあるお客様は (http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs)を参照してください。

目次

このドキュメントの使用法 15

1 パフォーマンスアナライザの概要 17
パフォーマンス解析ツール 17
コレクタツール 18
パフォーマンスアナライザツール 18
`er_print` ユーティリティー 20
パフォーマンスアナライザのウィンドウ 20

2 パフォーマンスデータ 21
コレクタが収集するデータ 21
クロックプロファイリングデータ 22
ハードウェアカウンタプロファイリングデータ 26
同期待ちトレースデータ 29
ヒープトレース (メモリー割り当て) データ 30
I/O トレースデータ 31
標本データ 31
MPI トレースデータ 32
プログラム構造へのメトリックの対応付け 36
関数レベルのメトリックス: 排他的、包括的、属性 36
属性メトリックスの解釈: 例 37
関数レベルのメトリックに再帰が及ぼす影響 39

3 パフォーマンスデータの収集 41
プログラムのコンパイルとリンク 41
ソースコードを分析するためのコンパイル 42
静的リンク 43
共有オブジェクトの処理 43
コンパイル時の最適化 43
Java プログラムのコンパイル 44

データ収集と解析のためのプログラムの準備 44
動的割り当てメモリーの利用 44
システムライブラリの使用 46
データ収集とシグナル 48
setuid と setgid の使用 50
libcollector ライブラリを使用したデータ収集のプログラム制御 50
C/C++、Fortran、および Java API 関数 52
動的な関数とモジュール 53
データ収集に関する制限事項 54
クロックプロファイリングに関する制限事項 54
トレースデータの収集に関する制限事項 55
ハードウェアカウンタプロファイリングに関する制限事項 56
ハードウェアカウンタプロファイリングによる実行時のひずみと実行時間の拡大 56
派生プロセスのデータ収集における制限事項 57
OpenMP プロファイルに関する制限事項 57
Java プロファイルに関する制限事項 57
Java プログラミング言語で書かれたアプリケーションの実行時のひずみと実行時間の拡大 58
収集データの格納場所 58
実験名 59
実験の移動 60
必要なストレージ容量の概算 60
データの収集 62
collect コマンドによるデータの収集 62
データ収集オプション 63
実験制御オプション 73
出力オプション 78
その他のオプション 80
collect ユーティリティーによる動作中のプロセスからのデータの収集 81
▼ collect ユーティリティーを使用して動作中のプロセスからデータを収集する 82
dbx collector サブコマンドによるデータの収集 82
▼ dbx からコレクタを実行する 82
データ収集のサブコマンド 83
実験制御のサブコマンド 86
出力サブコマンド 87
情報のサブコマンド 89
Oracle Solaris プラットフォーム上での dbx による動作中のプロセスからのデータの収集 89

▼ dbx の制御下にない実行中のプロセスからデータを収集する方法 89
動作中のプロセスからのトレースデータの収集 90
スクリプトからのデータの収集 91
collect と ppgsz を組み合わせた使用法 92
MPI プログラムからのデータの収集 92
MPI 用の collect コマンドの実行 93
MPI 実験の格納 93

4 パフォーマンスアナライザツール 97
パフォーマンスアナライザについて 97
パフォーマンスアナライザの起動 98
analyzer コマンドオプション 99
パフォーマンスアナライザのユーザーインタフェース 101
メニューバー 101
ツールバー 102
ナビゲーションパネル 102
「選択の詳細」ウィンドウ 102
「呼び出し元/呼び出し回数」パネル 103
パフォーマンスアナライザのビュー 103
「ようこそ」ページ 104
「概要」画面 105
「関数」ビュー 107
「タイムライン」ビュー 107
「ソース」ビュー 109
「呼び出しツリー」ビュー 111
「呼び出し元-呼び出し先」ビュー 111
インデックスオブジェクトビュー 113
「メモリーオブジェクト」ビュー 115
「I/O」ビュー 116
「ヒープ」ビュー 117
「データサイズ」ビュー 117
「経過時間」ビュー 117
「OpenMP 並列領域」ビュー 118
「OpenMP タスク」ビュー 118
「行」ビュー 119
「PC」ビュー 119
「逆アセンブリ」ビュー 119
「ソース/逆アセンブリ」ビュー 120
「競合」ビュー 121

「デッドロック」ビュー 121
「デュアルソース」ビュー 121
「統計」ビュー 121
「実験」ビュー 121
「命令頻度」ビュー 122
「MPI タイムライン」ビュー 122
「MPI チャート」ビュー 123
ライブラリとクラスの可視性の設定 124
データのフィルタリング 124
フィルタの使用 125
拡張カスタムフィルタの使用 126
フィルタリングのためのラベルの使用 127
パフォーマンスアナライザからのアプリケーションのプロファイリング 127
実行中のプロセスのプロファイリング 128
実験の比較 129
比較形式の設定 129
パフォーマンスアナライザのリモートでの使用 130
デスクトップクライアント上でのパフォーマンスアナライザの使用 130
パフォーマンスアナライザでのリモートホストへの接続 131
構成設定 131
「ビュー」設定 132
メトリック設定 133
タイムライン設定 134
ソース/逆アセンブリ設定 135
呼び出しツリー設定 136
書式設定 136
検索パス設定 137
パスマップ設定 138
パフォーマンスアナライザの構成ファイル 138

5 `er_print` コマンド行パフォーマンス解析ツール 141
`er_print` について 142
`er_print` の構文 142
メトリックリスト 143
関数リストを制御するコマンド 146
`functions` 147
`metrics` *metric-spec* 147
`sort` *metric_spec* 148
`fsummary` 149

fsingle *function-name* [*N*] 149
呼び出し元 - 呼び出し先リストを管理するコマンド 149
callers-callees 149
csingle *function-name* [*N*] 150
cprepend *function-name [N | ADDR]* 150
cappend *function-name [N | ADDR]* 150
crmfirst 151
crmlast 151
呼び出しツリーリストを制御するコマンド 151
calltree 151
トレースデータに共通のコマンド 151
datasize 151
duration 151
リークリストと割り当てリストを管理するコマンド 152
leaks 152
allocs 152
heap 152
heapstat 152
I/O アクティビティーレポートを制御するコマンド 152
ioactivity 152
iodetail 153
iocallstack 153
iostat 153
ソースリストと逆アセンブリリストを管理するコマンド 153
source|src { *filename* | *function-name* } [*N*] 153
disasm|dis { *filename* | *function-name* } [*N*] 154
scc *com-spec* 154
sthresh *value* 155
dcc *com-spec* 155
dthresh *value* 156
cc *com-spec* 156
PC と行を制御するコマンド 156
pcs 156
psummary 156
lines 157
lsummary 157
ソースファイルの検索を制御するコマンド 157
setpath *path-list* 157

addpath *path-list* 158
pathmap *old-prefix new-prefix* 158
データ領域リストを制御するコマンド 158
data_objects 158
data_single *name* [*N*] 159
data_layout 159
インデックスオブジェクトリストを制御するコマンド 159
indxobj *indxobj-type* 160
indxobj_list 160
indxobj_define *indxobj-type index-exp* 160
メモリーオブジェクトリストを制御するコマンド 160
memobj *mobj-type* 161
mobj_list 161
mobj_define *mobj-type index-exp* 161
memobj_drop *mobj_type* 162
machinemodel *model_name* 162
OpenMP インデックスオブジェクトのコマンド 162
OMP_preg 162
OMP_task 162
スレッドアナライザをサポートするコマンド 163
races 163
rdetail *race-id* 163
deadlocks 163
ddetail *deadlock-id* 163
実験、標本、スレッド、および LWP を一覧表示するコマンド 163
experiment_list 164
sample_list 164
lwp_list 164
thread_list 164
cpu_list 164
実験データのフィルタリングを制御するコマンド 165
フィルタ式の指定 165
フィルタ式のキーワードの一覧表示 165
フィルタリング用の標本、スレッド、LWP、および CPU の選択 165
ロードオブジェクトの展開と短縮を制御するコマンド 167
object_list 167
object_show *object1,object2,...* 168
object_hide *object1,object2,...* 168

object_api *object1,object2,...* 168
objects_default 168
object_select *object1,object2,...* 169
メトリックを一覧するコマンド 169
metric_list 169
cmetric_list 169
data_metric_list 169
indx_metric_list 170
出力を制御するコマンド 170
outfile {*filename*|-|--} 170
appendfile *filename* 170
limit *n* 170
name { long | short } [:{ *shared-object-name* | *no-shared-object-name* }] 171
viewmode { user| expert | machine } 171
compare { on | off | delta | ratio } 172
printmode *string* 172
その他の情報を出力するコマンド 172
header *exp-id* 172
ifreq 173
objects 173
overview *exp_id* 173
sample_detail [*exp_id*] 173
statistics *exp_id* 174
実験のためのコマンド 174
add_exp *exp_name* 174
drop_exp *exp_name* 174
open_exp *exp_name* 174
.er.rc ファイル内のデフォルトの設定 174
dmetrics *metric-spec* 175
dsort *metric-spec* 176
en_desc { on | off | =*regexp*} 176
その他のコマンド 176
procstats 176
script *filename* 177
version 177
quit 177
exit 177

help 177
... . 177
式の文法 177
フィルタ式の例 179
er_print コマンドの例 181

6 パフォーマンスアナライザとそのデータについて 185
データ収集の動作 185
実験の形式 186
実験の記録 188
パフォーマンスメトリックの解釈 189
クロックプロファイリング 189
ハードウェアカウンタオーバーフローのプロファイリング 192
データ領域プロファイリングとメモリー領域プロファイリング 193
同期待ちトレース 194
ヒープトレース 194
I/O トレース 194
MPI トレース 195
呼び出しスタックとプログラムの実行 195
シングルスレッド実行と関数呼び出し 195
明示的なマルチスレッド化 198
Java テクノロジーベースのソフトウェア実行の概要 199
Java プロファイリングの表示モード 200
OpenMP実行の概要 201
不完全なスタック展開 207
プログラム構造へのアドレスのマッピング 208
プロセスイメージ 208
ロードオブジェクトと関数 208
別名を持つ関数 209
一意でない関数名 209
ストリップ済み共有ライブラリの静的関数 210
Fortran での代替エントリポイント 210
クローン関数 211
インライン関数 211
コンパイラ生成の本体関数 212
アウトライン関数 212
動的にコンパイルされる関数 213
<Unknown> 関数 213
OpenMP の特殊な関数 214

<JVM-System> 関数 214
<no Java callstack recorded> 関数 215
<Truncated-stack> 関数 215
<Total> 関数 215
ハードウェアカウンタオーバーフロープロファイルに関連する関数 216
インデックスオブジェクトへのパフォーマンスデータのマッピング 216
メモリーオブジェクトへのパフォーマンスデータのマッピング 216
プログラムデータオブジェクトへのデータアドレスのマッピング 217
データオブジェクト記述子 217

7 注釈付きソースと逆アセンブリデータについて 221
ツールがソースコードを見つけるしくみ 221
注釈付きソースコード 222
パフォーマンスアナライザの「ソース」ビューのレイアウト 223
注釈付き逆アセンブリコード 229
注釈付き逆アセンブリの解釈 230
「ソース」タブ、「逆アセンブリ」タブ、「PC」タブの特別な行 233
アウトライン関数 234
コンパイラ生成の本体関数 234
動的にコンパイルされる関数 235
Java ネイティブ関数 237
クローン関数 237
静的関数 238
包括的メトリック 239
ストア命令とロード命令の注釈 239
分岐先 239
実験なしのソース/逆アセンブリの表示 239
-func 240

8 実験の操作 243
実験の操作 243
er_cp ユーティリティーを使った実験のコピー 243
er_mv ユーティリティーを使った実験の移動 244
er_rm ユーティリティーを使った実験の削除 244
実験へのラベル付け 244
er_label コマンドの構文 245
er_label の例 247
スクリプトでの er_label の使用 247

その他のユーティリティー 248
er_archive ユーティリティー 249
er_export ユーティリティー 251
er_html ユーティリティー 251

9 カーネルプロファイリング 253
カーネル実験 253
カーネルプロファイリング用のシステムの設定 253
er_kernel ユーティリティーの実行 254
▼ er_kernel を使用してカーネルをプロファイルする方法 255
▼ er_kernel を使用してロード状態でプロファイルする方法 255
カーネルのハードウェアカウンタオーバーフローのプロファイリング 256
カーネルおよびユーザープロセスのプロファイリング 257
カーネルとロードを同時にプロファイリングするための代替方法 258
カーネルプロファイルの分析 259

索引 261

このドキュメントの使用法

- **概要** – Oracle Developer Studio ソフトウェアのパフォーマンス解析ツールについて説明します。コレクタおよびパフォーマンスアナライザという 2 つのツールを併用することによって、広範囲のパフォーマンスデータを収集し、そのデータを関数、ソース行、および命令レベルでプログラム構造に関連付けます。収集されるパフォーマンスデータには、統計的な時間プロファイリング、ハードウェアカウンタプロファイリング、さまざまな呼び出しのトレースなどがあります。
- **対象読者** - アプリケーション開発者、開発者、アーキテクト、サポートエンジニア
- **必要な知識** - プログラミングの経験、プログラム/ソフトウェア開発テスト、ソフトウェア製品の構築およびコンパイルの能力

製品ドキュメントライブラリ

この製品および関連製品のドキュメントとリソースは `http://www.oracle.com/pls/topic/lookup?ctx=E71939` で入手可能です。

システム要件および既知の問題は、『Oracle Developer Studio 12.5: Release Notes』に記載されています。

フィードバック

このドキュメントに関するフィードバックを `http://www.oracle.com/goto/docfeedback` からお聞かせください。

♦ ♦ ♦ 第 1 章

1 パフォーマンスアナライザの概要

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

この章では、次の内容について説明します。

- 17 ページの「パフォーマンス解析ツール」
- 20 ページの「パフォーマンスアナライザのウィンドウ」

パフォーマンス解析ツール

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

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

これらのツールは、次のような疑問の解決に役立ちます。

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

コレクタツール

コレクタツールは次のようにパフォーマンスデータを収集します。

- クロックトリガーまたはハードウェアパフォーマンスカウンタのオーバーフローに基づく、プロファイリングと呼ばれる統計的な方法を使用する
- スレッド同期呼び出し、メモリー割り当ておよび割り当て解除呼び出し、I/O 呼び出し、Message Passing Interface (MPI) 呼び出しをトレースする
- システムおよびプロセスに関するサマリーデータとして

Oracle Solaris プラットフォームでは、クロックプロファイリングデータにはマイクロステートアカウンティングデータが含まれます。記録されたすべてのプロファイリングイベントおよびトレースイベントには、呼び出しスタックのほか、スレッドおよび CPU ID が含まれます。

コレクタは C、C++、および Fortran に関するプログラムのあらゆる種類のデータを収集できるとともに、Java™ プログラミング言語で書かれたアプリケーションに関するプロファイルデータを収集できます。また、動的に生成される関数と派生プロセスに関するデータも収集できます。収集対象のデータについては第2章「パフォーマンスデータ」を、コレクタの詳細については第3章「パフォーマンスデータの収集」を参照してください。コレクタはパフォーマンスアナライザ、`collect` コマンド、または `dbx collector` コマンドでアプリケーションをプロファイルするときに実行されます。

パフォーマンスアナライザツール

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

- クロックプロファイリングメトリック
- ハードウェアカウンタプロファイリングメトリック
- 同期待ちトレースメトリック
- I/O トレースメトリック
- ヒープトレースメトリック
- MPI トレースメトリック
- OpenMP メトリック
- 標本ポイント

パフォーマンスアナライザの「タイムライン」ビューには、raw データが時間の関数としてグラフィカル形式で表示されます。「タイムライン」ビューには、時間の関数として記録されたイベントおよび標本ポイントのチャートが表示されます。データは、水平バーに表示されます。

パフォーマンスアナライザでは、ターゲットプログラムのデータ領域の構造およびメモリーサブシステムの構造的コンポーネントに関するパフォーマンスのメトリックも表示できます。このデータは、ハードウェアカウンタメトリックの拡張です。

サポートされるアーキテクチャー上で記録された実験は、同じアーキテクチャーまたはその他の任意のサポートされるアーキテクチャー上で実行されているパフォーマンスアナライザで表示できます。たとえば、アプリケーションを Oracle Solaris SPARC サーバーで実行しているときにプロファイルし、Linux マシンで実行しているパフォーマンスアナライザで結果の実験を表示できます。

パフォーマンスアナライザのクライアントバージョンはリモートパフォーマンスアナライザと呼ばれ、Java が使用可能な Oracle Solaris、Linux、Windows、または Mac OS システムにインストールできます。このリモートパフォーマンスアナライザを実行して、完全な Oracle Developer Studio 製品がインストールされているサーバーに接続し、リモートから実験を表示できます。詳細は、130 ページの「パフォーマンスアナライザのリモートでの使用」を参照してください。

パフォーマンスアナライザは、Oracle Developer Studio 解析スイートのその他のツールで使用されます。

- スレッドアナライザは、スレッド解析実験の検査にこれを使用します。個別のコマンド `tha` は、パフォーマンスアナライザを専用のビューで起動し、特にこのような種類のデータの検査のために生成できる実験でのデータ競合およびデッドロックを表示します。

  『Oracle Developer Studio 12.5: スレッドアナライザユーザーズガイド』では、スレッドアナライザを使用する方法について説明しています。
- `uncover` コードカバレージユーティリティーでは、パフォーマンスアナライザを使用して、「関数」、「ソース」、「逆アセンブリ」、および「命令頻度」データビューにカバレージデータを表示します。詳細は、『Oracle Developer Studio 12.5: Discover および Uncover ユーザーズガイド』を参照してください。

ツールの使用についての詳細は、第4章「パフォーマンスアナライザツール」およびパフォーマンスアナライザの「ヘルプ」メニューを参照してください。

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

第6章「パフォーマンスアナライザとそのデータについて」では、パフォーマンスアナライザとそのデータに関連する、データ収集の動作、パフォーマンスメトリックの解釈、呼び出しスタックとプログラムの実行などのトピックについて説明しています。

第7章「注釈付きソースと逆アセンブリデータについて」では、注釈付きソースと逆アセンブリについて説明し、パフォーマンスアナライザが表示する各種インデックス行とコンパイラ解説について説明しています。この章では、コンパイラ解説を含むがパフォーマンスデータは含まない注釈付きソースコードのリストおよび逆アセンブリコードのリストの表示に使用できる `er_src` コマンド行ユーティリティーについても説明します。

第8章「実験の操作」では、実験のコピー、移動、削除、実験へのラベルの追加、および実験のアーカイブとエクスポートの方法について説明します。

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

注記 - パフォーマンスアナライザのデモコードは、Oracle Developer Studio 12.5 の Web ページ (`http://www.oracle.com/technetwork/server-storage/solarisstudio`) から、サンプルアプリケーション zip ファイルとしてダウンロードできます。

パフォーマンスアナライザでのサンプルコードの使用方法については、『Oracle Developer Studio 12.5: パフォーマンスアナライザチュートリアル』を参照してください。

er_print ユーティリティー

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

パフォーマンスアナライザのウィンドウ

このセクションでは、パフォーマンスアナライザのウィンドウレイアウトの簡単な概要を示します。ここで説明する機能および特徴の詳細は、第4章「パフォーマンスアナライザツール」および「ヘルプ」メニューを参照してください。

パフォーマンスアナライザを起動したときは、「ようこそ」ページを使用すると、さまざま方法でアプリケーションのプロファイリングを開始したり、最近の実験を表示したり、実験を比較したり、ドキュメントにナビゲートしたりすることが簡単にできます。

実験を開くと、記録されたデータの主要部分と使用可能な一連のメトリックが、「概要」に表示されます。調べるメトリックを選択できます。

パフォーマンスアナライザは、左側のナビゲーションバーからアクセスするデータビューを中心にして整理されています。各ビューには、プロファイルされたアプリケーションのパフォーマンスデータが異なる観点で表示されます。データビューは相互に接続され、あるビューの関数を選択すると、ほかのデータビューも更新され、その選択した関数に焦点が当てられます。

ほとんどのデータビューでは、コンテキストメニューからフィルタを選択するか、「フィルタ」ボタンをクリックすると、パフォーマンスアナライザの強力なフィルタリングテクノロジを使用してパフォーマンスの問題を詳しく調べることができます。

各ビューの詳細は、103 ページの「パフォーマンスアナライザのビュー」を参照してください。

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

♦ ♦ ♦ 第 2 章

2 パフォーマンスデータ

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

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

この章では、次の内容について説明します。

- 21 ページの「コレクタが収集するデータ」
- 36 ページの「プログラム構造へのメトリックの対応付け」

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

コレクタが収集するデータ

コレクタは、次のいくつかの方法を使用して、さまざまな種類のデータを収集します。

- **プロファイリングデータ**の収集は、一定の間隔でプロファイルイベントを記録することによって行います。間隔は、システム時間を使用して取得した時間間隔、または特定のタイプのハードウェアイベントの数です。指定の間隔に達すると、シグナルがシステムに送られ、次の機会にデータが記録されます。
- **トレースデータ**は、さまざまなシステム関数やライブラリ関数の呼び出しをインターセプトし、その呼び出しに関するデータを記録できるように、それらの関数にラッパー関数で割り込むことによって収集されます。
- **標本データ**の収集は、さまざまなシステムルーチンを呼び出して情報を取得することによって行います。
- **関数と命令のカウントデータ**は、実行可能ファイルおよび共有オブジェクトを組み込むことにより、実行可能ファイルに対して収集されます。また、動的に開かれているか、計測され、その実行可能ファイルに静的にリンクしているすべての共有オブジェクトに対して収集されます。各関数または基本ブロックが実行された回数が記録されます。
- **スレッド解析データ**は、スレッドアナライザをサポートするために収集されます。

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

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

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

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

こうした共通の情報のほかに、各イベントに固有のデータパケットには、データの種類に固有の情報が含まれます。

このあとの各項では、これらのデータの種類とその使用方法について説明します。

- 22 ページの「クロックプロファイリングデータ」
- 26 ページの「ハードウェアカウンタプロファイリングデータ」
- 29 ページの「同期待ちトレースデータ」
- 30 ページの「ヒープトレース (メモリー割り当て) データ」
- 31 ページの「I/O トレースデータ」
- 32 ページの「MPI トレースデータ」
- 31 ページの「標本データ」

クロックプロファイリングデータ

クロックプロファイリング時に収集されるデータは、オペレーティングシステムが提供する情報によって異なります。

Oracle Solaris でのクロックプロファイリング

Oracle Solaris でのクロックプロファイリングでは、各スレッドの状態が定期的な時間間隔で格納されます。この時間間隔は、プロファイリング間隔と呼ばれます。収集されるデータは、プロファイリング間隔の分解能を使用して、それぞれの状態で経過した時間に変換されます。

デフォルトのプロファイル間隔は、約 10 ミリ秒 (10 ms) です。約 1 ms の高分解能のプロファイリング間隔と約 100 ms の低分解能のプロファイリング間隔を指定できます。オペレーティングシステムで許可されている場合は、カスタム間隔を指定することもできます。ほかの引数を指定せずに `collect -h` コマンドを実行して、システム上で許容可能な範囲と分解能を出力します。

次の表に、実験にクロックプロファイリングデータが含まれる場合に、パフォーマンスアナライザと `er_print` が表示できるパフォーマンスメトリックを示します。すべてのスレッドからのメトリックが同時に追加されることに注意してください。

表 1 Oracle Solaris でのクロックプロファイリングからのタイミングメトリック

メトリック	定義
スレッド合計時間	スレッドがすべての状態で費やした時間の合計。
合計 CPU 時間	ユーザーモード、カーネルモード、またはトラップモードの CPU での実行に費やされたスレッド時間
ユーザー CPU 時間	ユーザーモードの CPU での実行に費やされたスレッド時間。
システム CPU 時間	カーネルモードの CPU での実行に費やされたスレッド時間。
トラップ CPU 時間	トラップモードの CPU での実行に費やされたスレッド時間。
ユーザーロック時間	同期ロックの待機に費やされたスレッド時間。
データページフォルト時間	データページの待機に費やされたスレッド時間。
テキストページフォルト時間	テキストページの待機に費やされたスレッド時間。
カーネルページフォルト時間	カーネルページの待機に費やされたスレッド時間。
停止時間	停止に費やされたスレッド時間。
CPU 待ち時間	CPU の待機に費やされたスレッド時間。
スリープ時間	スリープに費やされたスレッド時間

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

- ユーザー CPU 時間が大きいということは、その場所で、プログラムが仕事の大半を行なっていることを示します。この情報は、アルゴリズムを再設計することによって特に有益となる可能性があるプログラム部分を見つけるのに使用できます。
- システム CPU 時間が大きいということは、プログラムがシステムルーチンに対する呼び出しで多くの時間を使用していることを示します。
- CPU 待ち時間が大きいということは、使用可能な CPU 以上に実行可能なスレッドが多いか、ほかのプロセスが CPU を使用していることを示します。
- ユーザーロック時間が大きいということは、要求対象のロックをスレッドが取得できないことを示します。
- テキストページフォルト時間が大きいということは、リンカーによって指定されたコードが、多数の呼び出しまたは分岐で新しいページの読み込みが発生するようなメモリー上の配置になることを意味します。

- データページフォルト時間が大きいということは、データへのアクセスによって新しいページの読み込みが発生していること示します。この問題は、プログラムのデータ構造またはアルゴリズムを変更することによって解決できます。

Linux でのクロックプロファイリング

Linux プラットフォームでは、クロックデータは合計 CPU 時間としてのみ表示できます。Linux CPU 時間は、ユーザー CPU 時間とシステム CPU 時間の合計です。

OpenMP プログラム対応のクロックプロファイリング

クロックプロファイリングが OpenMP プログラムで実行される場合は、マスタースレッド時間、OpenMP 作業、および OpenMP 待機という追加メトリックが提供されます。

- Oracle Solaris では、マスタースレッド時間はマスタースレッドで費やされた合計時間であり、時計時間に対応します。このメトリックは Linux では使用できません。
- Oracle Solaris では、OpenMP 作業は、作業が直列または並列に実行されている場合に累積されます。OpenMP 待機は、OpenMP ランタイムが同期化を待機している場合に蓄積し、待機が CPU 時間またはスリーピングを使用しているか、または作業が並列実行中であるがスレッドが CPU 上にスケジュールされていない場合に蓄積します。
- Linux オペレーティングシステムでは、OpenMP 作業および OpenMP 待機は、プロセスがユーザーモードまたはシステムモードでアクティブである場合にのみ累積されます。OpenMP でビジーウェイトを行う必要があるものとして指定しないかぎり、Linux での OpenMP は有用ではありません。

OpenMP プログラムのデータは、3 つの表示モードのいずれかで表示できます。「ユーザー」モードでは、スレーブスレッドが実際にマスタースレッドから複製され、呼び出しスタックがマスタースレッドからのスレーブスレッドに対応しているかのように表示されます。OpenMP 実行時コード (`libmtsk.so`) から来た、呼び出しスタック内のフレームは抑制されます。「上級」ユーザーモードでは、マスタースレッドとスレーブスレッドが異なる方法で表示され、コンパイラで生成された明示的な関数が表示され、OpenMP 実行時コード (`libmtsk.so`) からのフレームは抑制されます。「マシン」モードでは、実際のネイティブなスタックが表示されます。

Oracle Solaris カーネルに対するクロックプロファイリング

`er_kernel` ユーティリティーは、Oracle Solaris カーネルに関するクロックベースのプロファイルデータを収集できます。コマンド行から直接 `er_kernel` ユーティリティーを実行するか、パフォーマンスアナライザの「ファイル」メニューから「カーネルのプロファイル」を選択すると、カーネルのプロファイリングを行えます。

`er_kernel` ユーティリティーはカーネルプロファイルデータを取り込み、そのデータをパフォーマンスアナライザの実験として、ユーザープログラム上で `collect` ユーティリティーによって作成され

た実験と同じ形式で記録します。この実験は、er_print ユーティリティーまたはパフォーマンスアナライザによって処理できます。カーネル実験は、関数データ、呼び出し元-呼び出し先データ、命令レベルのデータ、およびタイムラインを示すことができますが、ほとんどの Oracle Solaris モジュールが行番号テーブルを保持していないため、ソース行データを示すことはできません。

er_kernel は、ユーザーが権限を保持している、その時点で実行中のプロセスでのユーザーレベルの実験も記録できます。このような実験は、collect が作成する実験に似ていますが、ユーザー CPU 時間とシステム CPU 時間に関するデータだけを含み、Java および OpenMP プロファイリングはサポートしません。

詳細は、第9章「カーネルプロファイリング」を参照してください。

MPI プログラム対応のクロックプロファイリング

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

Oracle Message Passing Toolkit は、Oracle Solaris 11 リリースの一部として入手できます。システムにインストールされている場合は、/usr/openmpi にあります。まだ Oracle Solaris 11 システムにインストールされていない場合、システムにパッケージリポジトリが構成されていれば、コマンド pkg search openmpi を使用してこのパッケージを検索できます。Oracle Solaris 11 でのソフトウェアのインストールの詳細は、『*Oracle Solaris 11 ソフトウェアパッケージの追加および更新*』を参照してください。

MPI 実験に関するクロックプロファイリングデータを収集するときに、次の 2 つの追加メトリックが表示されます。

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

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

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

注記 - ほかのバージョンの MPI がサポートされています。完全なリストを表示するには、引数なしで collect と入力します。

注記 - Linux で Oracle Message Passing Toolkit 8.2 または 8.2.1 を使用する場合、回避策が必要になる場合があります。バージョン 8.1 または 8.2.1c では、または Oracle Developer Studio コンパイラを使用している場合はすべてのバージョンで、回避策は必要ありません。

Oracle Message Passing Toolkit のバージョンは、`/opt/SUNWhpc/HPC8.2.1` などのインストールパスで示されています。または、`mpirun –V` と入力して表示される次のような出力では、斜体の部分でバージョンが示されています。

```
mpirun (Open MPI) 1.3.4r22104-ct8.2.1-b09d-r70
```

アプリケーションを GNU または Intel コンパイラでコンパイルし、Oracle Message Passing Toolkit 8.2 または 8.2.1 を MPI 用に使用している場合、MPI の状態データを取得するには、Oracle Message Passing Toolkit の `link` コマンドで `-WI` および `--enable-new-dtags` オプションを使用する必要があります。これらのオプションを使用すると実行可能ファイルで `RPATH` に加えて `RUNPATH` が定義され、MPI 状態ライブラリが `LD_LIBRARY_PATH` 環境変数で有効になります。

ハードウェアカウンタプロファイリングデータ

ハードウェアカウンタは、キャッシュミス、キャッシュストールサイクル、浮動小数点演算、分岐予測ミス、CPU サイクル、および実行対象命令といったイベントの追跡に使用されます。ハードウェアカウンタプロファイリングでは、スレッドが実行されている CPU の指定されたハードウェアカウンタがオーバーフローしたときに、コレクタによってプロファイルパケットが記録されます。この場合、そのカウンタはリセットされ、カウントを続行します。プロファイルパケットには、オーバーフロー値とカウンタタイプが入っています。

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

また、ハードウェアカウンタプロファイリングは、パフォーマンスアナライザおよび `er_kernel` ユーティリティーを使用してカーネルに対して実行することもできます。詳細は、第9章「カーネルプロファイリング」を参照してください。

パフォーマンスアナライザは、ハードウェアカウンタプロファイリングデータをカウントメトリックに変換します。周期的にカウントするカウンタの場合、報告されたメトリックが時間に変換されます。周期的にカウントしないカウンタの場合、報告されるメトリックはイベントカウントです。複数の CPU を搭載したマシンの場合、メトリックの変換に使用されるクロック周波数が個々の CPU のクロック周波数の調和平均となります。プロセッサのタイプごとに専用のハードウェアカウンタセットがあり、またハードウェアカウンタの数が多いため、ハードウェアカウンタメトリックは

ここに記載していません。27 ページの「ハードウェアカウンタリスト」では、どのような種類のハードウェアカウンタがあるかについて調べる方法を説明します。

「cycles」と「insts」の 2 つの特定のカウンタが収集される場合、「CPI」と「IPC」という 2 つのメトリックが使用可能になり、これらはそれぞれ「命令当たりのサイクル数」と「サイクル当たりの命令数」を意味します。これらは常に比率として表示され、時間、カウント、あるいはパーセントで示されません。CPI 値が高いか IPC 値が低ければ、マシン内でのコードの実行が非効率であることを示し、逆に CPI 値が低いか IPC 値が高ければ、パイプライン内で実行中のコードの効率がよいことを示します。

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

ハードウェアカウンタはほかのカウンタと関連する場合があります。たとえば、分岐予測ミスが発生すると、間違った命令が命令キャッシュにロードされるため、分岐予測ミスと命令キャッシュミスが関連付けられることがよくあります。これらは正しい命令に置き換える必要があります。置換により、命令キャッシュミス、命令トランスレーションルックアサイドバッファー (ITLB) ミス、またはページフォルトが発生する可能性があります。

多くのハードウェアカウンタでは、オーバーフローイベントを引き起こした命令のあとの 1 つ以上の命令でオーバーフローが発生することがよくあります。この状況は「スキッド」と呼ばれ、カウンタオーバーフロープロファイルの解釈を困難にすることがあります。パフォーマンスデータは、観測されたイベントを実際に引き起こした命令のあとに実行された命令に関連付けられます。スキッドする命令の数は、多くのチップおよびカウンタでは確定していませんが、一部では正確に決まっています (通常は 0 または 1)。

メモリー関連のカウンタの一部は、スキッドが確定しているため、メモリー領域プロファイリングに使用できます。27 ページの「ハードウェアカウンタリスト」で説明されているリストで、このようなカウンタは「precise load-store」というラベルで示されます。このようなカウンタに対しては、トリガー元の PC および EA を特定でき、メモリー領域とデータ領域のデータがデフォルトで取得されます。詳細は、193 ページの「データ領域プロファイリングとメモリー領域プロファイリング」を参照してください。

ハードウェアカウンタリスト

ハードウェアカウンタはプロセッサ固有であるため、どのカウンタを利用できるかは、使用しているプロセッサによって異なります。パフォーマンスツールには、よく使われると考えられるいくつかのカウンタの別名が用意されています。現在のマシン上でほかのどの引数も指定せずに `collect -h` を実行することによって、現在のマシン上でのプロファイリングのためのハードウェアカウンタ定義の最大数を確認したり、使用可能なハードウェアカウンタの完全なリストやデフォルトのカウンタセットを表示したりすることができます。

プロセッサとシステムがハードウェアカウンタプロファイリングをサポートしている場合、`collect -h` コマンドは、ハードウェアカウンタに関する情報を含む 2 つのリストを出力します。最初のリストには、一般的な名称に別名が設定されたハードウェアカウンタが含まれます。2 番目の

リストには、raw ハードウェアカウンタが含まれます。パフォーマンスカウンタサブシステムにも collect コマンドにも特定システムのカウンタの名前がない場合、各リストは空になります。ただしほとんどの場合、カウンタは数値で指定できます。

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

```
    Aliases for most useful HW counters:

    alias      raw name                    type      units regs description

    cycles     Cycles_user                      CPU-cycles 0123 CPU Cycles
    insts      Instr_all                            events 0123 Instructions Executed
    c_stalls   Commit_0_cyc                     CPU-cycles 0123 Stall Cycles
    loads      Instr_ld       precise load-store    events 0123 Load Instructions
    stores     Instr_st       precise load-store    events 0123 Store Instructions
    dcm        DC_miss_commit precise load-store    events 0123 L1 D-cache Misses
...

Raw HW counters:

    name                                    type      units regs description

    Sel_pipe_drain_cyc                       CPU-cycles 0123
    Sel_0_wait_cyc                           CPU-cycles 0123
    Sel_0_ready_cyc                          CPU-cycles 0123
...
```

別名が設定されたハードウェアカウンタリストの形式

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

2 番目のフィールドは、カウンタの生の名前を示します。たとえば、loads は Instr_ld の省略です

3 番目のフィールドはタイプ情報を示し、空の場合もあります (たとえば、precise load-store)。

種類情報フィールド内の可能性のあるエントリには次のものがあります。

- precise。メモリーベースのカウンタ割り込みが、既知の正確なスキッドを伴って発生し、メモリー領域プロファイリングに対してサポートされます。このようなカウンタに対しては、パフォーマンスアナライザはメモリー領域とデータ領域のデータを収集でき、デフォルトで収集を行います。詳細は、115 ページの「「メモリーオブジェクト」ビュー」の説明を参照してください。
- load、store、または load-store のいずれかである場合、そのカウンタはメモリーに関連したものです。

- not-program-related である場合、カウンタはほかのプログラムによって開始されたイベント、たとえば CPU 対 CPU のキャッシュスヌープなどを取り込みます。プロファイリングにカウンタを使用すると、警告が生成され、プロファイリングで呼び出しスタックが記録されません。

4 番目のフィールドは、カウントされている単位の種類を示します (たとえば、events)。

単位は次のいずれかです。

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

5 番目のフィールドは、カウンタの使用可能なレジスタを示します。たとえば、0123 です。

6 番目のフィールドは、カウンタの簡単な説明を示します (たとえば、Load Instructions)。

raw ハードウェアカウンタリストの形式

raw ハードウェアカウンタリストに含まれる情報は、別名設定されたハードウェアカウンタリストに含まれる情報のサブセットです。raw ハードウェアカウンタリスト内の各行には、cputrack(1) で使用される内部カウンタ名、タイプ情報、CPU-cycles または events のどちらかのカウンタ単位、およびそのカウンタを使用できるレジスタ番号が含まれています。

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

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

同期待ちトレースデータ

マルチスレッドプログラムでは、別のスレッドで実行されるタスクの同期によって、プログラムの実行が遅延することがあります。たとえば、あるスレッドが、別のスレッドでロックされているデータへのアクセスを待機する必要が生じることがあります。これらのイベントは同期遅延イベントと呼ばれ、Solaris または pthread のスレッド関数の呼び出しをトレースすることによって収集されます。同期遅延イベントを収集して記録するプロセスを同期待ちトレースと言います。また、ロック待ちに費やされる時間を同期待ち時間と言います。

ただし、イベントが記録されるのは、その待ち時間がしきい値 (ミリ秒単位) を超えた場合だけです。しきい値 0 は、待ち時間に関係なく、あらゆる同期遅延イベントをトレースすることを意味

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

Java プログラムの場合、同期トレースは、プロファイルされるプログラムでの Java メソッド呼び出し、ネイティブな同期呼び出し、またはその両方を対象にします。

同期待ちトレースデータは、次の表に示すメトリックに変換されます。

表 2 同期待ちトレースメトリック

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

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

ヒープトレース (メモリー割り当て) データ

正しく管理されていないメモリー割り当て関数やメモリー割り当て解除関数を呼び出すと、データの使い方の効率が低下し、プログラムパフォーマンスが低下する可能性があります。ヒープトレースでは、C 標準ライブラリメモリー割り当て関数 `malloc`、`realloc`、`valloc`、`memalign`、および割り当て解除関数 `free` で割り込み処理を行うことによって、コレクタはメモリーの割り当てと割り当て解除の要求をトレースします。`mmap` の呼び出しはメモリー割り当てとして扱われ、これによって Java メモリー割り当てのヒープトレースイベントを記録することが可能になります。Fortran 関数 `allocate` および `deallocate` は C 標準ライブラリ関数を呼び出すため、これらのルーチンは間接的にトレースされます。

Java プログラムのヒーププロファイリングはサポートされません。

ヒープトレースデータは、次のメトリックに変換されます。

表 3 メモリー割り当て (ヒープトレース) メトリック

メトリック	定義
割り当て	メモリー割り当て関数の呼び出し回数
割り当てバイト数	メモリー割り当て関数の呼び出しごとに割り当てられるバイト数の合計

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

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

フィルタが適用された「リーク」ビューが表示された場合、そのリークはフィルタ基準で行われたメモリー割り当てに対するリークであり、常に割り当て解除されません。リークは、フィルタリング基準では割り当て解除されない割り当てに制限されません。

ここでのメモリーリークは、動的に割り当てられたが解放されることのないメモリーブロックと定義されます。そのメモリーブロックへのポインタがプロセスのアドレス空間に存在するかどうかは関係ありません。(ほかのツールでは、解放されていない割り当て済みメモリーブロックを指すポインタがプロセスのアドレス空間に存在しなくなった場合をリークと定義することがあります。)

I/O トレースデータ

I/O データ収集は、読み取りや書き込みを含む入力/出力システムコールをトレースします。呼び出しの時間を測定し、ファイルおよび記述子を追跡し、転送されたデータ量を測定します。I/O メトリックを使用すると、転送バイト数が多く、スレッド合計時間が長いファイル、ファイルハンドル、および呼び出しスタックを識別できます。

表 4 I/O トレースメトリック

メトリック	定義
読み取りバイト数	読み取り関数の各呼び出しで読み取られたバイト数の合計。
書き込みバイト数	書き込み関数の各呼び出しで書き込まれたバイト数の合計。
読み取り数	読み取り呼び出しが行われた回数。
書き込み数	書き込み呼び出しが行われた回数。
その他の I/O 数	その他の I/O 呼び出しが行われた回数
I/O エラー数	I/O 呼び出し中に発生したエラーの数
読み取り時間	データの読み取りにかかった秒数
書き込み時間	データの書き込みにかかった秒数
その他の I/O 時間	その他の I/O 呼び出しの実行にかかった秒数
I/O エラー時間	I/O エラーに費やされた秒数

標本データ

大域データは、コレクタによって*標本パケット*と呼ばれるパケット単位で記録されます。各パケットには、ヘッダー、タイムスタンプ、ページフォルトや I/O データなどのカーネルからの実行統計情

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

標本パケットは、次の状況で記録されます。

- `dbx` でのデバッグ中に、プログラムが何らかの理由により停止したとき (ブレークポイントで停止するようにオプションが設定されている場合にブレークポイントに到達した場合など)。
- 標本収集の間隔の終了時 (定期的な標本収集を選択している場合)。標本収集の間隔は整数値 (秒単位) で指定します。デフォルト値は 1 秒です。
- `dbx collector sample record` コマンドを使用し、標本を手動で記録したとき。
- このルーチンに対する呼び出しがコードに含まれている場合に `collector_sample` を呼び出したとき (50 ページの「libcollector ライブラリを使用したデータ収集のプログラム制御」を参照)。
- `collect` コマンドで `-l` オプションが使用されている場合に指定した信号が送信されたとき (`collect(1)` のマニュアルページを参照)。
- 収集が開始および終了したとき。
- `dbx collector pause` コマンドで収集を一時停止したとき (一時停止の直前)、および `dbx collector resume` コマンドで収集を再開したとき (再開の直後)。
- 派生プロセスが作成される前後。

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

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

MPI トレースデータ

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

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

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

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

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

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

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

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

次のリストに、データが収集される関数を示します。

MPI_Abort	MPI_Accumulate	MPI_Address
MPI_Allgather	MPI_Allgatherv	MPI_Allreduce
MPI_Alltoall	MPI_Alltoallv	MPI_Alltoallw
MPI_Attr_delete	MPI_Attr_get	MPI_Attr_put
MPI_Barrier	MPI_Bcast	MPI_Bsend
MPI_Bsend-init	MPI_Buffer_attach	MPI_Buffer_detach
MPI_Cancel	MPI_Cart_coords	MPI_Cart_create
MPI_Cart_get	MPI_Cart_map	MPI_Cart_rank
MPI_Cart_shift	MPI_Cart_sub	MPI_Cartdim_get
MPI_Comm_compare	MPI_Comm_create	MPI_Comm_dup
MPI_Comm_free	MPI_Comm_group	MPI_Comm_rank

MPI_Comm_remote_group	MPI_Comm_remote_size	MPI_Comm_size
MPI_Comm_split	MPI_Comm_test_inter	MPI_Dims_create
MPI_Errhandler_create	MPI_Errhandler_free	MPI_Errhandler_get
MPI_Errhandler_set	MPI_Error_class	MPI_Error_string
MPI_File_close	MPI_File_delete	MPI_File_get_amode
MPI_File_get_atomicity	MPI_File_get_byte_offset	MPI_File_get_group
MPI_File_get_info	MPI_File_get_position	MPI_File_get_position_shared
MPI_File_get_size	MPI_File_get_type_extent	MPI_File_get_view
MPI_File_iread	MPI_File_iread_at	MPI_File_iread_shared
MPI_File_iwrite	MPI_File_iwrite_at	MPI_File_iwrite_shared
MPI_File_open	MPI_File_preallocate	MPI_File_read
MPI_File_read_all	MPI_File_read_all_begin	MPI_File_read_all_end
MPI_File_read_at	MPI_File_read_at_all	MPI_File_read_at_all_begin
MPI_File_read_at_all_end	MPI_File_read_ordered	MPI_File_read_ordered_begin
MPI_File_read_ordered_end	MPI_File_read_shared	MPI_File_seek
MPI_File_seek_shared	MPI_File_set_atomicity	MPI_File_set_info
MPI_File_set_size	MPI_File_set_view	MPI_File_sync
MPI_File_write	MPI_File_write_all	MPI_File_write_all_begin
MPI_File_write_all_end	MPI_File_write_at	MPI_File_write_at_all
MPI_File_write_at_all_begin	MPI_File_write_at_all_end	MPI_File_write_ordered
MPI_File_write_ordered_begin	MPI_File_write_ordered_end	MPI_File_write_shared
MPI_Finalize	MPI_Gather	MPI_Gatherv
MPI_Get	MPI_Get_count	MPI_Get_elements
MPI_Get_processor_name	MPI_Get_version	MPI_Graph_create
MPI_Graph_get	MPI_Graph_map	MPI_Graph_neighbors
MPI_Graph_neighbors_count	MPI_Graphdims_get	MPI_Group_compare
MPI_Group_difference	MPI_Group_excl	MPI_Group_free
MPI_Group_incl	MPI_Group_intersection	MPI_Group_rank
MPI_Group_size	MPI_Group_translate_ranks	MPI_Group_union
MPI_Ibsend	MPI_Init	MPI_Init_thread
MPI_Intercomm_create	MPI_Intercomm_merge	MPI_Irecv
MPI_Irsend	MPI_Isend	MPI_Issend
MPI_Keyval_create	MPI_Keyval_free	MPI_Op_create
MPI_Op_free	MPI_Pack	MPI_Pack_size
MPI_Probe	MPI_Put	MPI_Recv
MPI_Recv_init	MPI_Reduce	MPI_Reduce_scatter
MPI_Request_free	MPI_Rsend	MPI_rsend_init
MPI_Scan	MPI_Scatter	MPI_Scatterv
MPI_Send	MPI_Send_init	MPI_Sendrecv

MPI_Sendrecv_replace	MPI_Ssend	MPI_Ssend_init
MPI_Start	MPI_Startall	MPI_Test
MPI_Test_cancelled	MPI_Testall	MPI_Testany
MPI_Testsome	MPI_Topo_test	MPI_Type_commit
MPI_Type_contiguous	MPI_Type_extent	MPI_Type_free
MPI_Type_hindexed	MPI_Type_hvector	MPI_Type_indexed
MPI_Type_lb	MPI_Type_size	MPI_Type_struct
MPI_Type_ub	MPI_Type_vector	MPI_Unpack
MPI_Wait	MPI_Waitall	MPI_Waitany
MPI_Waitsome	MPI_Win_complete	MPI_Win_create
MPI_Win_fence	MPI_Win_free	MPI_Win_lock
MPI_Win_post	MPI_Win_start	MPI_Win_test
MPI_Win_unlock		

MPI トレースデータは、次のメトリックに変換されます。

表 5 MPI トレースメトリック

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

MPI 時間は、MPI 関数で費やされたスレッド合計時間です。MPI 状態の時間も収集される場合、MPI_Init および MPI_Finalize 以外のすべての MPI 関数については、MPI 作業時間と MPI 待機時間の合計が MPI 作業時間にほぼ等しくなるはずです。Linux では、MPI 待ちおよび MPI ワークがユーザーとシステムの CPU 時間に基づいているのに対して、MPI 時間は実際の時間に基づいているため、これらの数値は一致しません。

MPI のバイトおよびメッセージカウントは、現在、ポイントツーポイントメッセージについてのみ収集されます。これらは集合通信関数については記録されません。MPI 受信バイト数は、すべてのメッセージで実際に受信したバイト数をカウントします。MPI 送信バイト数は、すべてのメッセージで実際に送信したバイト数をカウントします。MPI 送信数は送信したメッセージの数をカウントし、MPI 受信数は受信したメッセージの数をカウントします。

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

プログラム構造へのメトリックの対応付け

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

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

関数レベルのメトリックス: 排他的、包括的、属性

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

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

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

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

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

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

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

メトリック間の関係は、次の等式で表すことができます。

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

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

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

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

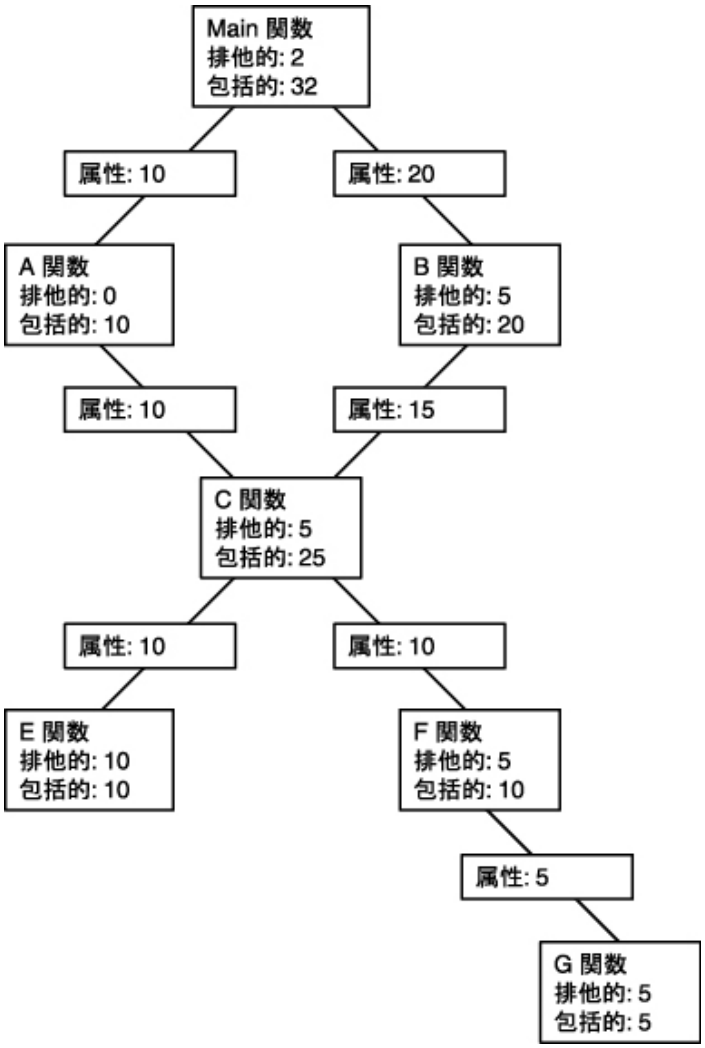
- 排他的メトリックを使用して、メトリック値が大きい関数を見つけます。
- 包括的メトリックを使用して、プログラム内のどの呼び出しシーケンスが大きなメトリック値の原因になっているかを判定します。
- 属性メトリックを使用して、大きなメトリック値の原因になっている 1 つまたは複数の関数に対する特定の呼び出しシーケンスをトレースします。

属性メトリックスの解釈: 例

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

プログラムの擬似コードは、図のあとに示されています。

図 1 排他的、包括的、属性メトリックを示す呼び出しツリー



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

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

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

呼び出し元の属性メトリックは、関数 A と B の包括的メトリックと排他的メトリックの差と等しくなります。これは、関数 A と B は関数 C のみを呼び出すことを意味します。実際、これらの関数がほかの関数を呼び出している場合がありますが、時間が短すぎて、実験に現れることはありません。

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

呼び出し先の属性メトリックと呼び出し先の包括的メトリックは、関数 E と関数 F に対して同じです。つまり、関数 E と関数 F の両方が関数 C からのみ呼び出されています。排他的メトリックと包括的メトリックは関数 E に対して同じですが、関数 F に対しては異なります。これは、関数 F は別の関数である関数 G を呼び出していますが、関数 E は呼び出していないためです。

このプログラムの擬似コードを次に示します。

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

A() {
   C(10);
}

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

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

関数レベルのメトリックに再帰が及ぼす影響

直接または間接のどちらの場合も、再帰関数呼び出しがあると、メトリックの計算が複雑になります。パフォーマンスアナライザは、関数の呼び出しごとのメトリックではなく、関数全体としてのメトリックを表示します。このため、一連の再帰的な呼び出しのメトリックを 1 つのメトリックに

要約する必要があります。この動作によって、呼び出しスタックの最後の関数 (リーフ関数) から計算される排他的メトリックが影響を受けることはありませんが、包括的および属性メトリックは影響を受けます。

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

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

♦ ♦ ♦ 第 3 章

3 パフォーマンスデータの収集

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

データの収集はコマンド行またはパフォーマンスアナライザツールから実行されます。

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

この章では、次の内容について説明します。

- 41 ページの「プログラムのコンパイルとリンク」
- 44 ページの「データ収集と解析のためのプログラムの準備」
- 54 ページの「データ収集に関する制限事項」
- 58 ページの「収集データの格納場所」
- 60 ページの「必要なストレージ容量の概算」
- 62 ページの「データの収集」
- 62 ページの「`collect` コマンドによるデータの収集」
- 82 ページの「`dbx collector` サブコマンドによるデータの収集」
- 89 ページの「Oracle Solaris プラットフォーム上での `dbx` による動作中のプロセスからのデータの収集」
- 91 ページの「スクリプトからのデータの収集」
- 92 ページの「`collect` と `ppgsz` を組み合わせた使用法」
- 92 ページの「MPI プログラムからのデータの収集」

プログラムのコンパイルとリンク

プログラムのコンパイル時には、ほとんどのコンパイラオプションを使用してデータの収集および解析を行うことができますが、収集対象とパフォーマンスアナライザでの表示対象に影響するオプションがいくつかあります。プログラムのコンパイルとリンクする際に考慮する問題は以降のサブセクションで説明します。

ソースコードを分析するためのコンパイル

注釈付きの「ソース」および「逆アセンブリ」ビューにソースコードを表示し、「行」ビューにソース行を表示するには、`-g` コンパイラオプション (C++ でフロントエンドインライン化を有効にするには `-g0`) で対象のソースファイルをコンパイルし、デバッグシンボル情報を作成します。デバッグシンボル情報の形式は、`-xdebugformat=dwarf` によって指定される DWARF2 です。これがデフォルトです。

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

プログラムは Oracle Developer Studio コンパイラまたは GNU コンパイラを使用してコンパイルできます。ただし、GNU コンパイラは OpenMP による再構築された呼び出しスタックなどの一部の機能をサポートできません。

`-g` を指定してコンパイルしても、Oracle Developer Studio コンパイラの `O2` と `O3` の最適化レベルでの末尾呼び出し最適化を除き、最適化に変更はありません。

Java コードのソースレベル情報がサポートされています。Java ソースの場所は、ネイティブ言語の場合とは異なり、実験に記録されません。ソースを指示するために、パスマッピングを使用するか、検索パスを設定することが必要な場合もあります。詳細は、221 ページの「ツールがソースコードを見つけるしくみ」を参照してください。

データ領域およびメモリー領域プロファイリング用のコンパイル

データ領域プロファイリングは、メモリーアクセスをデータ構造要素によるものとします。データ領域プロファイリングを有効にするには、Oracle Developer Studio コンパイラで `-xhwcprof` オプションを使用して、C、C++、および Fortran 実行可能ファイルをコンパイルする必要があります。コンパイルでこのオプションを指定しない場合、「データオブジェクト」ビューと「データレイアウト」ビューには、バイナリのデータは表示されません。

メモリー領域プロファイリングを使用すると、どのメモリーアドレスでパフォーマンスがもっとも低下しているかを確認できます。メモリー領域プロファイリング用のプログラム準備に特別なコンパイラオプションは必要ありませんが、この機能は、Oracle Solaris 10 1/13 を実行している SPARC プラットフォームと、Oracle Solaris 11.2 を実行している Intel プラットフォームでのみ使用できます。詳細は、193 ページの「データ領域プロファイリングとメモリー領域プロファイリング」を参照してください。

静的リンク

ヒープトレースや I/O トレースなどのパフォーマンスデータの一部の種類では、データ収集が動的にリンクされた `libc` に依存します。この機能は静的にリンクした場合に失われるため、Oracle Developer Studio コンパイラで `-dn` や `-Bstatic` などのオプションを使用して `libc` を制御しないでください。

完全に静的にリンクされたプログラム用のデータを収集しようとすると、コレクタはエラーメッセージを出力し、データを収集しません。このエラーが発生する原因は、コレクタを実行したときに、コレクタライブラリが動的に読み込まれるためです。

システムライブラリまたはコレクタライブラリ `libcollector.so` のいずれも静的にリンクしないでください。

共有オブジェクトの処理

通常、`collect` コマンドではターゲットのアドレス空間に含まれているすべての共有オブジェクトのデータが、初期ライブラリリストに含まれているものか、`dlopen()` によって明示的にロードされたものかにかかわらず、収集されます。ただし、特定の条件では一部の共有オブジェクトのプロファイリングが行われないことがあります。

- 遅延ロードによりターゲットプログラムが呼び出された場合。この場合、起動時にはライブラリがロードされず、`dlopen()` の呼び出しによって明示的にもロードされないため、共有オブジェクトが実験に含まれない可能性があり、共有されたオブジェクトからの PC はすべて `<Unknown>` 関数にマップされます。回避策として、環境変数 `LD_BIND_NOW` を設定すると、起動時にライブラリが強制的にロードされます。
- 実行可能ファイルが `-B direct` オプションを指定して構築された場合。この場合、オブジェクトは特に `dlopen()` の動的リンカーのエントリポイントへの呼び出しによって動的にロードされるため、`libcollector` 割り込みがバイパスされます。共有オブジェクト名が実験に含まれない可能性があり、そのオブジェクトからのすべての PC が `<Unknown>()` 関数にマップされます。回避策は、`-B direct` オプションを使用しないことです。プログラムが正常に終了すると、そのような共有オブジェクトが検出され、記録されます。

コンパイル時の最適化

何らかのレベルの最適化を有効にしてプログラムをコンパイルすると、コンパイラにより実行順序が変更され、プログラム内の行の順序と実行順序が厳密に一致しなくなることがあります。この場合、パフォーマンスアナライザは、最適化されたコードについて収集された実験データを解析できますが、しばしば、逆アセンブリレベルでパフォーマンスアナライザが提供するデータを元のソースコード行に対応付けることが困難になります。また、コンパイラが末尾呼び出しの最適化を行う場合には、呼び出しシーケンスが予想とは異なっているように見えることがあります。詳細は、197 ページの「末尾呼び出しの最適化」を参照してください。

Java プログラムのコンパイル

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

データ収集と解析のためのプログラムの準備

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

- シグナルハンドラをインストールする。48 ページの「データ収集とシグナル」を参照してください。
- 関数を動的にコンパイルする。53 ページの「動的な関数とモジュール」を参照してください。
- プロファイルする派生プロセスを作成する。74 ページの「-F *option* を使用してプロセスを追跡する」を参照してください。
- プロファイリングタイマーまたはハードウェアカウンタ API を直接使用する。46 ページの「システムライブラリの使用」を参照してください。
- setuid(2) を呼び出すか、setuid ファイルを実行する。48 ページの「データ収集とシグナル」および50 ページの「setuid と setgid の使用」を参照してください。

また、実行時にプログラムからデータ収集を制御する場合は、50 ページの「libcollector ライブラリを使用したデータ収集のプログラム制御」を参照してください。

動的割り当てメモリーの利用

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

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

メモリー割り当てメソッドが初期値を設定するときに明示的に記述される場合を除いて、動的に割り当てられたメモリーの初期の内容にプログラムが依存しないように注意する必要があります。たとえば、malloc(3C) のマニュアルページにある calloc と malloc の説明を比較してみます。

動的に割り当てられるメモリーを使用するプログラムを単独で実行すると、正常に機能しているように見えることがありますが、パフォーマンスデータの収集を有効にした状態で実行すると、問

題が起きることがあります。この場合、浮動小数点演算の予期しない動作、セグメント例外、アプリケーション固有のエラーメッセージなどが発生する可能性があります。

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

次のツールが、こうした潜在的な問題の発見に役立ちます。

- コードアナライザ。コンパイラやその他のツールと使用したときに次の項目を表示できる Oracle Developer Studio ツールです。

  静的コード検査 — コードアナライザは、Oracle Developer Studio の C または C++ コンパイラでアプリケーションをコンパイルし、`-xanalyze=code` オプションを指定した場合に実行される静的コード検査の結果を表示できます。

  動的メモリーアクセス検査 — コードアナライザは、`discover` で `-a` オプションを使用してバイナリに計測機構を組み込んだあと、その計測機構が組み込まれたバイナリを実行してデータを生成するときに実行される動的メモリーアクセス検査の結果を表示できます。

  詳細は、『Oracle Developer Studio 12.5: コードアナライザユーザーズガイド』を参照してください

- `f95 -xcheck=init_local`

  詳細は、『Oracle Developer Studio 12.5: Fortran ユーザーズガイド』または `f95`(1) のマニュアルページを参照してください。

- `lint` ユーティリティー

  詳細は、『Oracle Developer Studio 12.5: C ユーザーズガイド』または `lint`(1) のマニュアルページを参照してください。

- `dbx` 下での実行時チェック

  詳細は、『Oracle Developer Studio 12.5: dbx コマンドによるデバッグ』マニュアルまたは `dbx`(1) のマニュアルページを参照してください。

システムライブラリの使用

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

- 同期待ちトレースデータの収集。Oracle Solaris では、コレクタは Oracle Solaris C ライブラリ `libc.so` からの関数に割り込みます。
- I/O トレースデータの収集。コレクタは次の関数に割り込みます。

```
/* interposition function handles */
static int (*__real_open)(const char *path, int oflag, ...) = NULL;
#if WSIZE(32)
static int (*__real_open64)(const char *path, int oflag, ...) = NULL;
#endif
static int (*__real_fcntl)(int fildes, int cmd, ...) = NULL;
static int (*__real_openat)(int fildes, const char *path, int oflag, ...) = NULL;
static int (*__real_close)(int fildes) = NULL;
static FILE *(*__real_fopen)(const char *filename, const char *mode) = NULL;
static int (*__real_fclose)(FILE *stream) = NULL;
static int (*__real_dup)(int fildes) = NULL;
static int (*__real_dup2)(int fildes, int fildes2) = NULL;
static int (*__real_pipe)(int fildes[2]) = NULL;
static int (*__real_socket)(int domain, int type, int protocol) = NULL;
static int (*__real_mkstemp)(char *template) = NULL;
static int (*__real_mkstemps)(char *template, int slen) = NULL;
static int (*__real_creat)(const char *path, mode_t mode) = NULL;

#if WSIZE(32)
static int (*__real_creat64)(const char *path, mode_t mode) = NULL;
#endif
static FILE *(*__real_fdopen)(int fildes, const char *mode) = NULL;

static ssize_t (*__real_read)(int fildes, void *buf, size_t nbyte) = NULL;
static ssize_t (*__real_write)(int fildes, const void *buf, size_t nbyte) = NULL;
static ssize_t (*__real_readv)(int fildes, const struct iovec *iov, int iovcnt) = NULL;
static ssize_t (*__real_writev)(int fildes, const struct iovec *iov, int iovcnt) = NULL;
static size_t (*__real_fread)(void *ptr, size_t size, size_t nitems, FILE *stream) = NULL;
static size_t (*__real_fwrite)(void *ptr, size_t size, size_t nitems, FILE *stream) =
 NULL;
static ssize_t (*__real_pread)(int fildes, void *buf, size_t nbyte, off_t offset) = NULL;
static ssize_t (*__real_pwrite)(int fildes, const void *buf, size_t nbyte, off_t offset) =
 NULL;
#if OS(Linux)
static ssize_t (*__real_pwrite64)(int fildes, const void *buf, size_t nbyte, off64_t
 offset) = NULL;
```

```
#endif
static char *(*__real_fgets)(char *s, int n, FILE *stream) = NULL;
static int (*__real_fputs)(const char *s, FILE *stream) = NULL;
static int (*__real_fputc)(int c, FILE *stream) = NULL;
static int (*__real_fprintf)(FILE *stream, const char *format, ...) = NULL;
static int (*__real_vfprintf)(FILE *stream, const char *format, va_list ap) = NULL;

static off_t (*__real_lseek)(int fildes, off_t offset, int whence) = NULL;
static offset_t (*__real_llseek)(int fildes, offset_t offset, int whence) = NULL;
static int (*__real_chmod)(const char *path, mode_t mode) = NULL;
static int (*__real_access)(const char *path, int amode) = NULL;
static int (*__real_rename)(const char *old, const char *new) = NULL;
static int (*__real_mkdir)(const char *path, mode_t mode) = NULL;
static int (*__real_getdents)(int fildes, struct dirent *buf, size_t nbyte) = NULL;
static int (*__real_unlink)(const char *path) = NULL;
static int (*__real_fseek)(FILE *stream, long offset, int whence) = NULL;
static void (*__real_rewind)(FILE *stream) = NULL;
static long (*__real_ftell)(FILE *stream) = NULL;
static int (*__real_fgetpos)(FILE *stream, fpos_t *pos) = NULL;
static int (*__real_fsetpos)(FILE *stream, const fpos_t *pos) = NULL;
#if WSIZE(32)
static int (*__real_fgetpos64)(FILE *stream, fpos64_t *pos) = NULL;
static int (*__real_fsetpos64)(FILE *stream, const fpos64_t *pos) = NULL;
#endif // WSIZE(32)
static int (*__real_fsync)(int fildes) = NULL;
static struct dirent *(*__real_readdir)(DIR *dirp) = NULL;
#if OS(Linux)
static int (*__real_flock)(int fd, int operation) = NULL;
#endif
static int (*__real_lockf)(int fildes, int function, off_t size) = NULL;
static int (*__real_fflush)(FILE *stream) = NULL;
#if OS(Linux) && ARCH(Intel) && WSIZE(32)
static FILE *(*__real_fopen_2_1)(const char *filename, const char *mode) = NULL;
static int (*__real_fclose_2_1)(FILE *stream) = NULL;
static FILE *(*__real_fdopen_2_1)(int fildes, const char *mode) = NULL;
static int (*__real_fgetpos_2_2)(FILE *stream, fpos_t *pos) = NULL;
static int (*__real_fsetpos_2_2)(FILE *stream, const fpos_t *pos) = NULL;
static FILE *(*__real_fopen_2_0)(const char *filename, const char *mode) = NULL;
static int (*__real_fclose_2_0)(FILE *stream) = NULL;
static FILE *(*__real_fdopen_2_0)(int fildes, const char *mode) = NULL;
static int (*__real_fgetpos_2_0)(FILE *stream, fpos_t *pos) = NULL;
static int (*__real_fsetpos_2_0)(FILE *stream, const fpos_t *pos) = NULL;
#endif
```

- ヒープトレースデータの収集。コレクタは、malloc、realloc、memalign、free の関数に割り込みます。これらの関数は、C 標準ライブラリ libc.so のほか、libumen.so、libmalloc.so、libmtmalloc.so などのライブラリにあります。

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

次の状況では割り込み処理は成功しません。

- 割り込み対象関数が入っているライブラリとプログラムを静的にリンクした場合。
- コレクタライブラリが事前読み込みされていない実行中アプリケーションに `dbx` を接続した場合。
- これらのライブラリのいずれか 1 つを動的に読み込み、このライブラリの中でだけ検索することによってシンボルを解決する場合。

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

`er_sync.so`、`er_iotrace.so`、`er_heap.so`、および `er_mpview`*n*`.so` (*n* は MPI バージョンを示す) ライブラリは、それぞれ同期待ちトレースデータ、I/O トレースデータ、ヒープトレースデータ、または MPI トレースデータが要求された場合にのみロードされます。

データ収集とシグナル

シグナルは、クロックプロファイリングとハードウェアカウンタプロファイリングの両方に使用されます。`SIGPROF` は、すべての実験のデータ収集で使用されます。シグナルを生成する期間は、収集されるデータによって異なります。Solaris での `SIGEMT` または Linux での `SIGIO` はハードウェアカウンタプロファイリングに使用されます。オーバーフローの間隔は、プロファイリングのユーザーパラメータによって異なります。プロファイリングシグナルを使用または操作する何らかのユーザーコードが、データ収集の妨げになる可能性があります。コレクタは、そのシグナルハンドラをプロファイルシグナル用にインストールする場合、シグナルを配信するシステムコールの処理が中断されないようにするためのフラグを設定します。この設定により、プロファイリングシグナルをほかの目的に使用しているターゲットプログラムの動作が変更される可能性があります。

コレクタは、そのシグナルハンドラをプロファイルシグナル用にインストールするときに、ターゲットが独自のシグナルハンドラをインストールしたかどうかを記憶しています。コレクタはまた、一部のシグナル処理ルーチンに割り込み、ユーザーがこれらのシグナル用にシグナルハンドラをインストールできないようにします。コレクタは、実験の開始時にユーザーハンドラを置き換える場合と同様に、そのユーザーのハンドラを保存します。

プロファイリングシグナルは、カーネル内のプロファイリングタイマーまたはハードウェアカウンタオーバーフロー処理コードから、または `kill(2)`、`sigsend(2)`、`tkill(2)`、`tgkill(2)`、または `_lwp_kill(2)` のシステムコール、`raise(3C)` および `sigqueue(3C)` のライブラリコール、あるいは `kill` コマンドへの応答として送信されます。コレクタが起点を区別できるように、シグナルコードがシグナルとともに配信されます。シグナルコードは、プロファイリングのために配信される場合はコレクタによって処理されます。プロファイリングのために配信されない場合は、ターゲットのシグナルハンドラに配信されます。

コレクタが `dbx` の下で実行されているときに、配信されたプロファイリングシグナルのシグナルコードが破壊されたり、プロファイルシグナルがシステムまたはライブラリコールあるいはコマンドから生成されたかのように扱われたりする場合があります。その場合は、ユーザーのハンドラに誤って配信されます。ユーザーハンドラが `SIG_DFL` に設定されていた場合は、プロセスが失敗してコアダンプが生成されます。

コレクタがターゲットプロセスへの接続のあとに呼び出された場合、コレクタはそのシグナルハンドラをインストールしますが、シグナル処理ルーチンに割り込むことはできません。これらのユーザーコードが接続のあとにシグナルハンドラをインストールした場合は、コレクタのシグナルハンドラがオーバーライドされ、データは失われます。

どちらかのプロファイリングシグナルを含むシグナルによって、システムコールが途中で終了することがあります。この動作を処理するようにプログラムを準備する必要があります。`libcollector` は、データ収集のシグナルハンドラをインストールするときに、可能であればシステムコールを再開するように指定します。ただし、`sleep`(3C) などの一部のシステムコールはエラーを報告することなく早期に復帰します。

標本シグナルと一時停止/再開シグナル

シグナルは、ユーザーが標本シグナル (-l) または一時停止/再開シグナル (-y) として指定できます。この使用には SIGUSR1 または SIGUSR2 が推奨されますが、ターゲットによって使用されない任意のシグナルを使用できます。

プロファイリングシグナルは、プロセスがそれ以外ではそのシグナルを使用しない場合に使用できますが、ほかのシグナルが使用できない場合にのみ使用するようにしてください。コレクタは、一部のシグナル処理ルーチンに割り込み、ユーザーがこれらのシグナル用にシグナルハンドラをインストールできないようにします。コレクタは、実験の開始時にユーザーハンドラを置き換える場合と同様に、そのユーザーのハンドラを保存します。

コレクタがターゲットプロセスへの接続のあとに呼び出され、ユーザーコードが標本または一時停止/再開シグナル用にシグナルハンドラをインストールした場合、それらのシグナルは指定されたとおりに動作しなくなります。

setuid と setgid の使用

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

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

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

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

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

libcollector ライブラリを使用したデータ収集のプログラム制御

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

API 関数は、次のように定義されます。

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

CollectorAPI クラスに、Java™ プログラム用の類似の機能が用意されており、これについては、51 ページの「Java インタフェース」で説明しています。

C および C++ インタフェース

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

有効な実験がない場合、API 呼び出しは無視されます。

Fortran インタフェース

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

Fortran の場合、API 関数を使用するには、次の文を挿入します。

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

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

Java インタフェース

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

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

Java CollectorAPI メソッドは、次のように定義されます。

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

```
CollectorAPI.terminate()
```

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

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

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

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

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

C/C++、Fortran、および Java API 関数

このセクションでは、API 関数について説明します。

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

  Fortran: `collector_sample(string name)`

  Java: `CollectorAPI.sample(String name)`

  標本パケットを記録し、その標本に指定された名前または文字列をラベルとして付けます。このラベルは、パフォーマンスアナライザの「タイムライン」ビューで標本を選択したときに「選択の詳細」ウィンドウに表示されます。Fortran の引数 `string` の型は、`character` です。

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

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

- C、C++、Fortran: `collector_pause()`

  Java: `CollectorAPI.pause()`

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

- C、C++、Fortran: `collector_resume()`

  Java: `CollectorAPI.resume()`

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

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

  Java: `CollectorAPI.terminate`

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

動的な関数とモジュール

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

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

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

このセクションでは、API 関数について説明します。

`collector_func_load()` 関数

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

表 6　collector_func_load() のパラメータリスト

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

collector_func_unload() 関数

アドレス vaddr にある動的関数がアンロードされたことをコレクタに通知します。

データ収集に関する制限事項

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

異なるデータ型の同時収集について制限はありません。任意のデータ型をそれ以外の任意のデータ型とともに収集できますが、カウントデータは除きます。

デフォルトで、コレクタは深さ上限 256 フレームまでスタックを収集します。スタックがさらに深い場合は、er_print とパフォーマンスアナライザに <Truncated-stack> 関数が表示されることがあります。詳細は、215 ページの「<Truncated-stack> 関数」を参照してください。これよりも深いスタックをサポートするには、SP_COLLECTOR_STACKBUFSZ 環境変数に設定されている値を増やします。

クロックプロファイリングに関する制限事項

プロファイリング間隔の最小値と、プロファイリングに使用するクロック分解能は、オペレーティング環境により異なります。最大値は 1 秒に設定されています。プロファイル間隔の値は、時

間の分解能のもっとも近い倍数に切り捨てられます。最小値と最大値、およびクロック分解能を表示するには、collect コマンドをほかの引数を指定せずに入力します。

プロファイリングタイマーを使用するプログラムではクロックプロファイリングを行えません。コレクタは、プロファイリングクロックパラメータを設定する setitimer(3) の呼び出しをインターセプトし、ほかのプログラムで使用できないようにします。

Linux プラットフォームでは、クロックデータは合計 CPU 時間としてのみ表示できます。Linux CPU 時間は、ユーザー CPU 時間とシステム CPU 時間の合計です。

Linux システムでは、マルチスレッドアプリケーションの時間プロファイリングによって、スレッドの不正確なデータが報告される可能性があります。プロファイルシグナルは、必ずしもカーネルから各スレッドに指定された間隔で配信されるとは限りません。シグナルが間違ったスレッドに配信される場合もあります。使用可能な場合は通常、サイクルカウンタを使用したハードウェアカウンタプロファイリングによって、スレッドのより正確なデータが提供されます。

クロックプロファイリングによる実行時のひずみと実行時間の拡大

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

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

トレースデータの収集に関する制限事項

コレクタライブラリ libcollector.so が事前読み込みされていないかぎり、すでに稼働中のプログラムからはトレースデータを収集できません。詳細は、90 ページの「動作中のプロセスからのトレースデータの収集」を参照してください。

トレースによる実行時のひずみと実行時間の拡大

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

ハードウェアカウンタプロファイリングに関する制限事項

ハードウェアカウンタプロファイリングには、次のような制限があります。

- ハードウェアカウンタデータの収集は、ハードウェアカウンタが用意されていてハードウェアカウンタプロファイリングをサポートしているプロセッサでのみ行えます。その他のシステムでは、ハードウェアカウンタプロファイリングは無効です。Oracle Solaris と、Unbreakable Enterprise Kernel または Red Hat 互換のカーネル 6.0 以降を備えた Oracle Linux は、ハードウェアカウンタをサポートします。
- Oracle Solaris を実行しているシステム上で、`cpustat`(1) コマンドの実行中にハードウェアカウンタデータを収集することはできません。これは、`cpustat` がカウンタを制御しており、ユーザープロセスでカウンタを使用できないためです。データ収集中に `cpustat` を起動すると、ハードウェアカウンタプロファイリングは終了され、実験にエラーが記録されます。`root` がハードウェアカウンタを使用して `er_kernel` 実験を開始する場合にも、同じことが当てはまります。
- ハードウェアカウンタプロファイリングを行う場合、独自のコードでハードウェアカウンタを使用することはできません。コレクタは `libcpc` ライブラリ関数に割り込み、コレクタからの呼び出しではなかった場合、`-1` の戻り値で復帰します。ハードウェアカウンタへのアクセスの取得に失敗した場合に正常に機能するようにプログラムをコーディングする必要があります。この処理を行うようにコーディングされていない場合、ハードウェアカウンタプロファイル時に、スーパーユーザーがカウンタを使用するシステム全体のツールを起動した場合、またはそのシステムでカウンタがサポートされていない場合にプログラムが失敗します。
- `dbx` をプロセスに接続することによって、ハードウェアカウンタライブラリを使用している実行中プログラムのハードウェアカウンタデータを収集しようとすると、実験が壊れたり、プログラムが異常終了したりすることがあります。

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

ハードウェアカウンタプロファイリングによる実行時のひずみと実行時間の拡大

ハードウェアカウンタプロファイリングは、`SIGEMT` シグナル (Oracle Solaris プラットフォームの場合) または `SIGIO` シグナル (Linux プラットフォームの場合) がターゲットへ配信された時点のデータを記録します。それによってシグナルを処理するための実行時間の拡大が発生し、呼び出しスタックが展開されます。クロックプロファイリングと違い、ハードウェアカウンタによっては、プログラムのさまざまな部分がその他の部分より高速にイベントを生成する場合があり、そのコード部分に実行時間の拡大が生じます。そのようなイベントを非常に高速に生成するプログラムの一部で大きなひずみが生じる場合があります。同様に、あるスレッドでは、ほかのスレッドと不均等にイベントが生成されるものがあります。

派生プロセスのデータ収集における制限事項

派生プロセスのデータ収集には、いくつかの制限事項があります。

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

- `-F on` オプションを使用すると、`system`、`popen`、`posix_spawn`(3p)、`posix_spawnp`(3p)、`sh` への呼び出しによるものを含め、`fork` とそのバリアント、および `exec` とそのバリアントへの呼び出しについて、自動的にデータを収集できます。
- `-F all` は `-F on` と同じです。
- `-F '=`*regexp*`'` オプションを使用すると、名前が指定した正規表現と一致するすべての派生プロセスに関するデータを収集できます。

`-F` オプションについては、73 ページの「実験制御オプション」を参照してください。

OpenMP プロファイルに関する制限事項

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

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

Java プロファイルに関する制限事項

Java プログラムのデータ収集には、次の制限事項があります。

- Java 2 Software Development Kit (JDK) の JDK 7、Update 25 (JDK 1.7.0_25) またはそれ以降のバージョンを使用する必要があります。コレクタは、`JDK_HOME` 環境変数または `JAVA_PATH` 環境変数のパスセットに含まれている JDK を最初に探します。これらの変数がいずれも設定されていない場合、使用されている環境の `PATH` に含まれている JDK を探します。`PATH` にも JDK が存在しない場合、`/usr/java/bin/java` の `java` 実行可能ファイルを探します。

  コレクタは、見つかった `java` 実行可能ファイルのバージョンが ELF 実行可能ファイルであることを確認します。異なる場合は、使用された環境変数またはパスと試行されたフルパス名を示すエラーメッセージが出力されます。

- データを収集するには、collect コマンドを使用する必要があります。dbx collector サブコマンドは使用できません。
- 一部のアプリケーションは純粋な Java ではなく、C または C++ アプリケーションで、dlopen() を起動して libjvm.so をロードしてから、その中への呼び出しを行なって JVM ソフトウェアを開始します。このようなアプリケーションをプロファイルするには、SP_COLLECTOR_USE_JAVA_OPTIONS 環境変数を設定します。この場合は、LD_LIBRARY_PATH 環境変数を設定しないでください。

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

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

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

収集データの格納場所

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

コレクタは実験データを記録するばかりでなく、プログラムが使用したロードオブジェクトの独自のアーカイブも作成します。これらのアーカイブには、すべてのオブジェクトファイルとそのロードオブジェクト内のすべての関数のアドレス、サイズ、名前、ロードオブジェクトのアドレス、その最終変更日時を示すタイムスタンプが含まれます。アーカイブには、すべての共有オブジェクトとソースファイルの一部またはすべてのコピーが含まれる場合があります。詳細は、249 ページの「er_archive ユーティリティー」を参照してください。

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

派生プロセスの実験は、親プロセスの実験の内部に格納されます。

実験名

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

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

サブ実験も同様の命名ルールに従います。詳細は、187 ページの「サブ実験」を参照してください。

実験グループ

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

実験グループファイルを手動で作成するには、最初の行が次の内容であるプレーンテキストファイルを作成します。

```
#analyzer experiment group
```

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

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

派生プロセスの実験

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

MPI プログラムの実験

MPI プログラムのデータはデフォルトでは `test.1.er` に収集され、MPI プロセスからのすべてのデータはサブ実験に収集されます (ランクごとに 1 つ)。コレクタは、MPI ランクを使用して、`M_r`*m*`.er` の形式のサブ実験名を構築します。ここで、*m* は MPI ランクです。たとえば、MPI ランク 1 の実験データは、`test.1.er/M_r1.er` ディレクトリ内に記録されます。

カーネルおよびユーザープロセス上の実験

カーネル上の実験には、デフォルトでは `test.1.er` ではなく、`ktest.1.er` という名前が付けられます。データがユーザープロセスでも収集されるときは、カーネル実験には後続のユーザープロセスごとのサブ実験が含まれます。

サブ実験には、*_process-name* `_PID_`*process-id*`.1.er` という形式を使用して名前が付けられます。たとえば、プロセス ID 1264 で実行されている `sshd` プロセス上で実行される実験には、`ktest.1.er/_sshd_PID_1264.1.er` という名前が付けられます。

実験の移動

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

実験には、関数レベルでメトリックを計算してタイムラインを表示するのに必要な情報がすべて含まれています。ただし、注釈付きソースコードや注釈付き逆アセンブリコードを調べるには、ターゲットまたは実験の構築に使用されたものと同じバージョンのロードオブジェクトやソースファイルにアクセスできる必要があります。

実験のソースコードを検索するために使用されるプロセスについては、221 ページの「ツールがソースコードを見つけるしくみ」を参照してください。

プログラムに対応する、正しい注釈付きソースコードと注釈付き逆アセンブリコードが確実に表示されるようにするには、`er_archive` コマンドを使用して、ソースコード、オブジェクトファイル、および実行可能ファイルを実験にコピーしてから、実験の移動やコピーを行います。

詳細は、`er_archive`(1) のマニュアルページを参照してください。

必要なストレージ容量の概算

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

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

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

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

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

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

実験サイズの概算では、アーカイブファイルに使用されるディスク容量も考慮する必要があります (58 ページの「収集データの格納場所」参照)。必要なディスク容量のサイズを確定できない場合は、実験を短時間だけ行なってみてください。この実験からアーカイブファイルのサイズを取得し (データ収集時間とは無関係)、プロファイルファイルのサイズを確認することによって、完全な実験のサイズを概算できます。

アーカイブには、すべての共有オブジェクトとソースファイルの一部またはすべてのコピーが含まれる場合があります。詳細は、249 ページの「`er_archive` ユーティリティー」を参照してください。

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

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

選択的なデータ収集および分析については、Oracle Developer Studio の Web ページ `http://www.oracle.com/technetwork/server-storage/solarisstudio` から、*データ選択性と Oracle Solaris Studio パフォーマンスアナライザ*の記事を参照してください。

データの収集

ユーザーモードのターゲットに関するパフォーマンスデータを収集するには、次のいくつかの方法があります。

- コマンド行から `collect` コマンドを使用する (62 ページの「`collect` コマンドによるデータの収集」および `collect`(1) のマニュアルページを参照)。`collect` コマンド行ツールは `dbx` よりもデータ収集のオーバーヘッドが小さいため、この方法のほうが適切な場合があります。
- パフォーマンスアナライザの「アプリケーションのプロファイル」ダイアログボックスを使用する。(パフォーマンスアナライザヘルプの「アプリケーションのプロファイリング」を参照)。
- `dbx` コマンド行から `collector` コマンドを使用する。(82 ページの「`dbx collector` サブコマンドによるデータの収集」を参照)。

派生プロセスのデータの自動収集は、「アプリケーションのプロファイル」ダイアログボックスと `collect` コマンドからのみ行えます。

Oracle Solaris カーネルに関するパフォーマンスデータは、`er_kernel` ユーティリティーまたは「カーネルのプロファイル」ダイアログボックスを使用して収集できます。詳細は、第9章「カーネルプロファイリング」を参照してください。

`collect` コマンドによるデータの収集

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

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

collect-options は `collect` コマンドのオプションで、*program* はデータ収集対象のプログラム名、*program-arguments* はプログラムの引数です。ターゲットプログラムは通常、バイナリ実行可能ファイルまたはスクリプトです。

引数なしで呼び出された場合、collect は、実験のデフォルトの構成を含む使用法のサマリーを表示します。

プロファイリングに使用可能なオプションの一覧およびすべてのハードウェアカウンタの名前の一覧を取得するには、collect -h コマンドをほかの引数を指定せずに入力します。

```
% collect -h
```

ハードウェアカウンタの一覧については、26 ページの「ハードウェアカウンタプロファイリングデータ」を参照してください。56 ページの「ハードウェアカウンタプロファイリングに関する制限事項」も参照してください。

データ収集オプション

これらのオプションは、収集されるデータの種類を制御します。データの種類については、21 ページの「コレクタが収集するデータ」を参照してください。

データ収集オプションを指定しなかった場合、デフォルトは -p on で、デフォルトのプロファイリング間隔 (約 10 ミリ秒) でクロックプロファイリングが行われます。

-p off でクロックプロファイリングを明示的に無効とし、すべてのトレースとハードウェアカウンタプロファイリングを有効にしなかった場合、collect コマンドは警告メッセージを出力し、大域標本データだけを収集します。

-p *option* を使用したクロックプロファイリング

-p オプションを使用すると、クロックプロファイリングデータを収集できます。*option* に使用できる値は次のとおりです。

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

collect コマンドは、デフォルトでクロックプロファイリングデータを収集します。カウントデータ (-c) またはデータ競合およびデッドロックデータ (-r) を収集しない場合は、-p オプションを指定していなくてもクロックプロファイリングデータが収集されます。

-h high または -h low を指定して、プロセッサのデフォルトのハードウェアカウンタが高周波数または低周波数に設定されるように要求する場合、デフォルトのクロックプロファイリングも高または低に設定されます。-p hi または -p low あるいは -p *n* オプションを使用して明示的に指定することにより、クロックプロファイリングに対し別の周波数を設定できます。

Linux 上でのマルチスレッドアプリケーションのクロックプロファイリングに関する注意事項については、54 ページの「クロックプロファイリングに関する制限事項」を参照してください。

collect -h を使用したハードウェアカウンタプロファイリング

ハードウェアカウンタプロファイリングデータを収集します。一部のハードウェアでは、collect コマンドは、引数を付けずに collect -h を使用して表示できるデフォルトのカウンタセットを定義しています。デフォルトのカウンタセットを使用する代わりに、特定のカウンタを指定することもできます。カウンタの指定には複数の -h 引数を使用できます。

collect -h で表示されるカウンタの形式については、27 ページの「ハードウェアカウンタリスト」を参照してください。

collect -h で使用される *option* で指定できる値は次のとおりです。

off	ハードウェアカウンタプロファイリングを無効にします。-h off を使用する場合は、ほかのオプションを一緒に指定することはできません。
on	特定のハードウェアシステムのデフォルトのカウンタセットを使用してハードウェアカウンタプロファイリングを有効にします。システムにデフォルトのカウンタセットがない場合に -h on を指定するとエラーが発生します。
hi | high	システムのデフォルトカウンタセットに対してハードウェアカウンタプロファイリングを有効にし、高レートでプロファイリングします。システムにデフォルトのカウンタセットがない場合に -h hi を指定するとエラーが発生します。
lo | low	システムのデフォルトカウンタセットに対してハードウェアカウンタプロファイリングを有効にし、低レートでプロファイリングします。システムにデフォルトのカウンタセットがない場合に -h lo を指定するとエラーが発生します。
ctr_def... [,*ctr_n_def*]	指定された 1 つ以上のカウンタを使用してハードウェアカウンタプロファイルを収集します。サポートされるカウンタ (*ctr_def* から *ctr_n_def*) の最大数はプロセッサに依存します。プロファイリング用のハードウェアカウンタ定義の最大数を判断し、使用可能なハードウェアカウンタとデフォルト

のカウンタセットをすべて一覧表示するには、現在のマシン上で引数を付けずに collect -h を実行します。

メモリー関連のカウンタのタイプは、ほかのどのコマンド行引数も指定せずに collect -h コマンドを実行するときにカウンタリストに表示される load、store、または load-store です。このようなカウンタの一部には、precise というラベルも付けられます。precise カウンタの場合は、SPARC または x86 のどちらでも、デフォルトではデータ領域およびメモリー領域データが記録されます。

各 *ctr_def* カウンタ定義の形式を次に示します。

ctr[*~attr=val*]...[*~attrN=valN*][*/reg#*],[*interval*]

カウンタ定義の各オプションの意味は次のとおりです。

ctr
: ほかのコマンド行引数を使用せずに collect -h コマンドを実行して表示されるプロセッサ固有のカウンタ名。ほとんどのシステムでは、カウンタがリストにない場合でも、そのカウンタを引き続き 16 進数 (0x1234) または 10 進数のどちらかの数値で指定できます。古いチップのドライバでは数値がサポートされませんが、最近のチップのドライバではサポートされています。カウンタを数値で指定する場合は、レジスタ番号も指定するようにしてください。使用する数値は、チップ固有の製造元のマニュアルに記載されています。マニュアルの名前は、collect -h の出力で指定されます。一部のカウンタは、ベンダー独自のマニュアルでしか説明されていません。

~attr=val
: 任意の 1 つ以上の属性オプション。一部のプロセッサでは、属性オプションをハードウェアカウンタに関連付けることができます。プロセッサが属性オプションをサポートしている場合は、ほかのどのコマンド行引数も指定せずに collect -h を実行すると、*~attr* に使用する属性名のリストも表示されます。*val* の値は、10 進数または 16 進数の形式で指定できます。16 進数の形式の数値は、数値の前に 0 と小文字の x が付加された C プログラムの形式 (0x*hex_number*) で指定します。カウンタ名に複数の属性が連結されます。それぞれの属性名の前には ~ が必要です。

/reg#
: カウンタに使用するハードウェアレジスタ。指定されていない場合、collect は、カウンタを使用可能な最初のレジスタに配置しようとします。その結

果、レジスタの競合のために以降のカウンタを配置できない可能性があります。複数のカウンタを指定する場合は、各カウンタで異なるレジスタを使用する必要があります。使用可能なレジスタ番号のリストは、ほかのどのコマンド行引数も指定せずに `collect -h` コマンドを実行することによって確認できます。レジスタを指定する場合は、`/` 文字が必要です。

interval

この間隔は標本収集頻度であり、カウンタオーバーフロー値 (ハードウェアイベントの間隔) を定義することによって設定できます。有効な値は次のとおりです。

- `on` – ピーク速度での間隔がデフォルトのクロックプロファイリング速度 (100x/秒) とほぼ一致するようにカウンタオーバーフロー値を調整します。
- `hi` – 選択したカウンタの高分解能値で、`on` の約 1/10 です。
- `lo` – 選択したカウンタの低分解能値で、`on` の約 10 倍です。
- *value* - 特定の値で、10 進数または 16 進数形式の正の整数です。

注記 - 設定する *value* が小さすぎると、アプリケーションや場合によってはシステム全体が過負荷になる可能性があります。スレッドあたり 1000 イベント/秒より少なくなる設定をお勧めします。

間隔を省略した場合、`-h on` の値が使用されます。ただし、間隔を省略した場合でも、`-h` パラメータで最後のカウンタを指定しないかぎり、省略した間隔の指定の前にコンマを付ける必要があります。

例 1 `-h` の有効な使用例

```
-h on
-h lo
-h hi
```

デフォルトのカウンタを、それぞれデフォルトレート、低レート、または高レートで有効にします

```
-h cycles,,insts,,dcm
-h cycles -h insts -h dcm
    どちらも同じ意味です。3 つのカウンタ cycles、insts、およびデータキャッシュミスのデータ領域プロファイリング (SPARC のみ)

-h cycles~system=1
    ユーザーモードとシステムモードの両方でサイクルをカウントします

-h 0xc0/0,10000003
    Nehalem の場合。次と同等です
-h inst_retired.any_p/0,10000003
```

例 2 -h の無効な使用例

```
-h cycles -h off
      off は、ほかの -h 引数と一緒に使用できません
-h cycles,insts
    コンマがありません。「insts」は数値として解析されません
  <interval>
```

-h 引数がハードウェアカウンタの使用を指定していても、コマンドが指定された時点でハードウェアカウンタがルートによって使用されている場合、collect コマンドはエラーを報告し、実験は実行されません。

-h 引数が指定されない場合、ハードウェアカウンタプロファイリングデータは収集されません。実験では、ハードウェアカウンタオーバーフロープロファイリングとクロックベースのプロファイリングの両方を指定できます。ハードウェアカウンタオーバーフロープロファイリングを指定しても、クロックプロファイリングは (デフォルトで有効になっていても) 無効になりません。

-s *option* を使用した同期待ちトレース

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

on
: 同期遅延トレースを有効にし、実行時の調整によってしきい値を設定します。

calibrate
: on と同じです。

off
: 同期遅延トレースを無効にします。

n

n マイクロ秒のしきい値で同期遅延トレースを有効にします。*n* が 0 である場合は、すべてのイベントをトレースします。

`all`

同期遅延トレースを有効にし、すべての同期イベントをトレースします。

デフォルトでは、同期遅延トレースを無効にします。

Java プログラムの場合、同期トレースは、プロファイルされるプログラムでの Java メソッド呼び出し、ネイティブな同期呼び出し、またはその両方を対象にします。

Oracle Solaris では、次の関数がトレースされます。

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

Linux では、次の関数がトレースされます。

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

注記 - `-H` または `-i` オプションでデータをトレースすると、非常に大きな実験が生成される可能性があります。

-H *option* を使用したヒープトレースデータ

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

`on` メモリー割り当てリクエストのトレースを有効にします。

`off` メモリー割り当てリクエストのトレースを無効にします。

デフォルトでは、ヒープのトレースは無効です。

ヒープトレースイベントはネイティブな呼び出しについて記録され、`mmap` の呼び出しは、メモリー割り当てとして扱われます。

ヒーププロファイリングは、Java プログラムに対してはサポートされていません。これを指定すると、エラーとして扱われます。

ヒープトレースにより、非常に大きな実験が生成される可能性があることに注意してください。このような実験は、ロードや参照が非常に遅くなります。

-i *option* を使用した I/O トレース

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

`on` I/O 操作のトレースを有効にします。

`off` I/O 操作のトレースを無効にします。

I/O トレースはデフォルトでは実行されません。I/O トレースは、非常に大きな実験を生成する可能性があり、このような実験ではロードや参照が非常に遅くなります。

-c *option* を使用したカウントデータ

カウントデータを記録します。

option に使用できる値は次のとおりです。

`on` カウントデータを有効にします。

`static` すべての命令が正確に 1 回実行されたという前提に基づいて、シミュレートされたカウントデータを有効にします。

`off` カウントデータを無効にします。

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

カウントデータおよびシミュレートされたカウントデータの場合、実行可能ファイルと、計測され静的にリンクしているすべての共有オブジェクトがカウントされます。シミュレートされたものではないカウントデータの場合、動的にロードされた共有オブジェクトも計測されカウントされます。

Oracle Solaris では、特別なコンパイルは不要ですが、コンパイルフラグ `-p`、`-pg`、`-qp`、`-xpg`、および `--xlinkopt` を使用したカウントオプションは互換性がありません。

Linux では、カウントデータを収集するには実行可能ファイルを `-annotate=yes` フラグでコンパイルする必要があります。

Oracle Linux 5 では、実行時リンカー監査インタフェース (rtld-audit または LD_AUDIT とも呼ばれます) が不安定なため、カウントデータの収集が妨げられる可能性があります。

`-I` *directory* を使用してカウントデータ計測ディレクトリを指定する

カウントデータ計測のディレクトリを指定します。このオプションは Oracle Solaris システム上でのみ使用可能であり、`-c` オプションも同時に指定されている場合にのみ意味があります。

`-N` *library-name* を使用して除外されるライブラリを指定する

カウントデータの計測から除外するライブラリを指定します。ここで指定したものは、ライブラリが実行可能ファイルにリンクされているか、`dlopen()` によってロードされるかにかかわらず、除外されます。このオプションは Oracle Solaris システム上でのみ使用可能であり、`-c` オプションも同時に指定されている場合にのみ意味があります。`-N` オプションは複数指定できます。

`-S` *option* を使用した標本データ

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

`off`	定期的な標本収集を無効にします。
`on`	デフォルトの標本収集間隔 (1 秒) で定期的な標本収集を有効にします。
n	*n* (秒単位) の標本収集間隔で定期的な標本収集を有効にします。*n* は正である必要があります。

デフォルトでは、1 秒間隔による定期的標本収集が有効になります。

データ指定の引数が指定されていない場合は、デフォルトの分解能を使用して、クロックプロファイリングが実行されます。

クロックプロファイリングが明示的に無効になっており、かつハードウェアカウンタオーバーフロープロファイリングまたは何らかの種類のトレースのどちらも有効になってない場合は、collect は、関数レベルのデータが収集されないという警告を表示したあと、ターゲットを実行してグローバルデータを記録します。

-r *option* を使用したデータ競合およびデッドロックの検出

スレッドアナライザ用に、データ競合検出またはデッドロック検出のデータを収集します。

option に使用できる値は次のとおりです。

race
: データの競合を検出するためのデータを収集します。

deadlock
: デッドロックおよび潜在的デッドロックを検出するためのデータを収集します。

all
: データの競合、デッドロック、および潜在的デッドロックを検出するためのデータを収集します。race,deadlock としても指定できます。

off
: データの競合、デッドロック、および潜在的デッドロックのデータ収集を無効にします。

on
: データの競合を検出するためのデータを収集します (race と同じ)。

terminate
: 回復不可能なエラーが検出された場合は、ターゲットプロセスを終了します。

abort
: 回復不可能なエラーが検出された場合は、ターゲットプロセスを終了してコアダンプを出力します。

continue
: 回復不可能なエラーが検出された場合も、プロセスの続行を許可します。

デフォルトでは、スレッドアナライザのすべてのデータの収集を無効にします。terminate、abort、および continue オプションは、どのデータ収集オプションにも追加でき

ます。これらのオプションは、実際の (潜在的でない) デッドロックなどの回復不可能なエラーが発生した場合の動作を管理します。デフォルトの動作は terminate です。

スレッドアナライザのデータは、トレースデータとともに収集することはできませんが、クロックまたはハードウェアカウンタプロファイリングデータと組み合わせて収集できます。スレッドアナライザのデータはターゲットの実行速度を大幅に低下させるため、プロファイルをユーザーコードに適用しても意味がない可能性があります。

スレッドアナライザの実験は、analyzer または tha のどちらかで検査できます。スレッドアナライザ (tha) はデフォルトのデータビューの単純なリストを表示しますが、それ以外は同じです。

データ競合の検出を有効にする前に、コンパイル時に実行可能ファイルを計測するか、ポストプロセッサを呼び出して実行可能ファイルを計測する必要があります。ターゲットが計測されず、かつそのライブラリリスト上のどの共有オブジェクトも計測されない場合は、警告が表示されますが、実験は実行されます。スレッドアナライザのその他のデータには計測は必要ありません。

collect -r コマンドとスレッドアナライザの詳細は、『Oracle Developer Studio 12.5: スレッドアナライザユーザーズガイド』および tha(1) のマニュアルページを参照してください。

-M *option* を使用した MPI プロファイリング

MPI 実験の収集を指定します。collect コマンドのターゲットは mpirun コマンドである必要があります。また、mpirun コマンドのオプションは、-- オプションを使って mpirun コマンドによって実行されるターゲットプログラムと分けて指定されることが必要です。(mpirun コマンドでは常に -- オプションを使用することで、collect コマンドとそのオプションを mpirun コマンド行の先頭に追加し、実験を収集できます。)実験には通常どおりに名前が付けられ、親の実験として参照されます。そのディレクトリには、MPI プロセスそれぞれのサブ実験が含まれ、ランクで名前が付けられます。

option に使用できる値は次のとおりです。

MPI-version

指定された MPI バージョンを前提にして、MPI 実験の収集を有効にします。認識された MPI のバージョンは、引数なしで collect を入力したとき、または -M で指定された認識されないバージョンに応答して出力されます。

off

MPI 実験の収集を無効にします。

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

collect -h コマンドをほかのオプションを指定せずに入力するか、または -M オプションで認識されないバージョンを指定した場合は、MPI のサポートされているバージョンが出力されます。

-m *option* を使用した MPI トレース

MPI トレースデータを収集します。

option に使用できる値は次のとおりです。

`on` MPI トレース情報を有効にします。

`off` MPI トレース情報を無効にします。

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

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

実験制御オプション

これらのオプションは、実験データの収集方法を制御します。

-L *size* を使用して実験サイズを制限する

記録するプロファイリングデータの量を *size*M バイトに制限します。この制限は、すべてのプロファイリングデータとトレースデータの合計に適用されますが、標本ポイントには適用されません。この制限値は概数にすぎないので、この値を超えることは可能です。

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

size に使用できる値は次のとおりです。

`unlimited` または `none`
: 実験にサイズ制限を適用しません。

n
: *n*M バイトの制限を適用します。*n* の値は正である (0 を超えている) 必要があります。

デフォルトでは、記録されるデータサイズに制限はありません。

約 2G バイトの制限を付けるには、たとえば、-L 2000 を指定します。

-F *option* を使用してプロセスを追跡する

派生プロセスのデータを記録するかどうかを制御します。-F 設定と関係なく、常に親プロセスに関するデータが収集されます。*option* に使用できる値は次のとおりです。

on
: すべての子孫プロセスについて実験を記録します。

all
: on と同じです。

off
: どの子孫プロセスについても実験を記録しません。

=*regex*
: 実行可能ファイル名が正規表現に一致する派生プロセスについて実験を記録します。フルパスではなく、実行可能ファイルのベース名のみが使用されます。使用する *regex* に、シェルによって解釈される空白または文字が含まれている場合は、*regex* 引数全体を必ず単一引用符で囲むようにしてください。

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

system(3C)、system(3F)、sh(3F)、posix_spawn(3p)、posix_spawnp(3p)、popen(3C)、および同様の関数の呼び出しによって作成された派生プロセスと、それに関連した派生プロセスも追跡されます。

Linux では、CLONE_VM フラグを使用せずに clone(2) によって作成された派生がデフォルトで追跡されます。CLONE_VM フラグを使用して作成された派生は、プロセスではなくスレッドとして扱われ、-F の設定には関係なく常に追跡されます。

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

正規表現の使用例:

- 親での system の最初の呼び出しからの最初の fork からの最初の exec の派生プロセスに関するデータを取得するには、`collect -F '=_x1_f1_x1'` を使用します。
- exec のすべてのバリアントに関するデータを取得するが、fork については取得しないようにするには、`collect -F '=.*_x[0-9]/*'` を使用します
- system ("goodbye") ではなく system ("echo hello") の呼び出しからデータを取得するには、`collect -F '=echo hello'` を使用します。

派生プロセスの実験を作成し命名する方法の詳細は、187 ページの「サブ実験」を参照してください。

MPI 実験では、デフォルトで派生も追跡されます。

パフォーマンスアナライザと `er_print` ユーティリティーは、親実験が読み取られると、自動的に派生プロセスの実験を読み取り、データ画面に派生を表示します。

コマンド行から特定のサブ実験のデータを表示のために選択するには、そのサブ実験のパス名を `er_print` または `analyzer` コマンドへの引数として明示的に指定します。指定されるパスには、初期の実験の名前と、親ディレクトリの内部にある派生実験の名前が含まれている必要があります。

たとえば、`test.1.er` 実験の 3 回目の fork のデータを表示するには、次のように実行します。

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

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

もう一つの方法として、関心のある派生の実験の明示的な名前を入れた実験グループファイルを用意する方法もあります。詳細は、59 ページの「実験グループ」を参照してください。

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

また、スクリプトのデータを収集して、スクリプトの派生プロセスを追跡することもできます。詳細は、91 ページの「スクリプトからのデータの収集」を参照してください。

`-j` *option* を使用して Java をプロファイリングする

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

on

JVM マシンのプロファイリングデータを記録し、Java HotSpot 仮想マシンでコンパイルされたメソッドを認識するほか、Java 呼び出しスタックも記録します。これはデフォルト値です。

`off`

Java プロファイリングデータの記録を無効にします。ネイティブ呼び出しスタックのプロファイリングデータは引き続き記録されます。

path

JVM のプロファイリングデータを記録し、指定された *path* にインストールされた JVM を使用します。

-J *java-argument* を使用して Java オプションを渡す

プロファイルで使用するために JVM へ渡される追加引数を指定します。`-J` オプションを指定した場合、Java プロファイリング (`-j on`) が有効になります。*java-argument* に複数の引数が含まれる場合、全体を引用符でくくる必要があります。この引数は、空白またはタブで区切られた一連のトークンで構成される必要があります。各トークンは、別々の引数として JVM へ渡されます。JVM へのほとんどの引数は、ダッシュ文字 (-) で始まる必要があります。

-l *signal* による標本収集のシグナルの指定

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

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

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

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

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

-t *duration* を使用して時間範囲を設定する

データ収集の時間範囲を指定します。

duration は単独の数値として指定でき、オプションとして `m` または `s` 接尾辞を付け、実験終了までの時間が分単位か秒単位かを示すこともできます。デフォルトでは、所要時間は秒で

す。*duration* はハイフンで区切られた 2 つの数で指定することもできます。この場合、1 つ目の時間が経過するまでデータ収集は停止します。その時間になるとデータ収集が開始します。2 つ目の時間が経過すると、データ収集が終了されます。2 つ目の時間がゼロの場合、初めてプログラムが停止したあと、そのプログラムの実行の終わりまで、データの収集が実行されます。実験が終了しても、ターゲットプロセスは最後まで実行できます。

-x を使用して dbx attach を許可するためにプロファイリングされたターゲットを停止する

デバッガが接続できるようにするために、exec システムコールからの終了時にターゲットプロセスを停止したままにします。collect コマンドは、プロセス ID を含むメッセージを出力します。

ターゲットが collect によって停止されたあと、そのターゲットにデバッガを接続するには、次の手順に従います。

1. collect -x コマンドによって出力されたメッセージからプロセスの PID を取得します。
2. -P*PID* オプションを指定して collect コマンドを呼び出し、データを収集します。

   あるいは、次の手順を使用して手動で行うこともできます。

   a. デバッガを起動します
   b. SIGPROF と、ハードウェアカウンタデータを収集することを選択した場合は Oracle Solaris では SIGEMT、Linux では SIGIO を無視するようにデバッガを構成します。
   c. dbx attach コマンドを使用してプロセスに接続します。
   d. 収集する実験のためのコレクタのパラメータを設定します。
   e. collector enable コマンドを発行します。
   f. ターゲットプロセスを実行できるように cont コマンドを発行します。

プロセスがデバッガの制御下で実行されると、コレクタは実験を記録します。

別の方法として、プロセスに接続し、collect -P *PID* コマンドを使用して実験を収集することもできます。

-y *signal* [,r] を使用したシグナルの一時停止および再開状態

signal という名前のシグナルで、データの記録を制御します。このシグナルがプロセスに送信されると、一時停止状態 (データは記録されない) と記録状態 (データは記録される) が切り替わります。

デフォルトで、データ収集は一時停止状態から開始します。オプションの ,r フラグを指定した場合、データ収集は再開状態から開始します。つまりプロファイリングが即座に行われます。ただし、-y オプションの状態に関係なく、標本ポイントは常に記録されます。

一時停止/再開シグナルの 1 つの使用法として、データを収集せずにターゲットを起動し、ターゲットが定常状態に到達できるようにしたあと、データ収集を有効にする方法があります。

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

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

-y オプションに r 引数 (オプション) を指定した場合、コレクタは記録状態から起動します。それ以外の場合は一時停止状態から起動します。-y オプションが使用されていない場合、コレクタは記録状態で起動します。

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

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

出力オプション

これらのオプションは、コレクタによって生成される実験を制御します。

-o *experiment-name* を使用して実験名を設定する

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

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

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

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

-d *directory-name* を使用して実験のディレクトリを設定する

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

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

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

-g *group-name* を使用してグループ内に実験を作成する

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

-A *option* を使用してロードオブジェクトを実験にアーカイブする

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

- `on` – ロードオブジェクト (ターゲット、およびターゲットが使用するすべての共有オブジェクト) を実験にコピーします。また、ロードオブジェクトに存在しないスタブまたは DWARF デ

バッグ情報を含むすべての .anc ファイルと .o ファイルをコピーします。これはデフォルト値です。

- src – -A on の場合と同様のロードオブジェクトのコピーに加えて、見つかったすべてのソースファイルと .anc ファイルを実験にコピーします。
- used[src] – -A on の場合と同様のロードオブジェクトのコピーに加えて、記録されたデータで参照されていて、見つかったすべてのソースファイルと .anc ファイルを実験にコピーします。
- off – ロードオブジェクトを実験にコピーすることもアーカイブすることもしません。

実験を別のマシンにコピーするか、または実験を別のマシンから読み取る予定の場合は、-A on を指定します。実験で使用するディスク容量は増えますが、ほかのマシンで実験を読み取ることができます。

-A on でも、ソースファイルまたはオブジェクトファイル (.o) は実験にコピーされません。これらのファイルが、実験の検査に使用するマシンからアクセス可能であることを確認する必要があります。実験が記録されたあとで、ファイルを変更したり、再構築しないでください。

収集時に実験をアーカイブすると、特に多数の派生プロセスが含まれている実験では、非常にコストが高くなる場合があります。このような実験では、-A off を使用してデータを収集してから、実行の終了後に -A フラグを付けて er_archive を実行することをお勧めします。

-A のデフォルト設定は on です。

-O *file* を使用してコマンド出力をファイルに保存する

collect 自体のすべての出力を指定された *file* のあとに付加しますが、生成されたターゲット、(-P オプションで呼び出された) dbx、(-c 引数で呼び出された) カウントデータの記録にかかわるプロセスのいずれの出力もリダイレクトしません。*file* が /dev/null に設定されている場合は、あらゆるエラーメッセージを含む collect のすべての出力が抑制されます。

その他のオプション

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

-P *process-id* を使用してプロセスに接続する

dbx が、指定された *process-id* のプロセスに接続し、プロセスからデータを収集し、そのスクリプトで dbx を起動するためのスクリプトを作成します。

クロックまたはハードウェアカウンタプロファイリングデータは指定できますが、トレースデータもカウントデータもサポートされません。詳細は、collector(1) のマニュアルページを参照してください。

プロセスに接続するとき、ディレクトリは collect -P を実行中のユーザーの umask を使用して作成されますが、実験は dbx の接続先のプロセスを実行するユーザーとして書き込まれます。接続を実行するユーザーが root で、umask がゼロでない場合、実験は失敗します。

-C *comment* を使用して実験にコメントを追加する

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

-n を使用してコマンドを試す

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

-V を使用して collect バージョンを表示する

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

-v を使用して verbose 出力を表示する

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

collect ユーティリティーによる動作中のプロセスからのデータの収集

Oracle Solaris プラットフォームの場合のみ、collect ユーティリティーで -P *pid* オプションを使用して、指定された PID を持つプロセスに接続し、そのプロセスからデータを収集できます。collect コマンドのそのほかのオプションは dbx 用のスクリプトに変換され、そのスクリプトを起動してデータが収集されます。クロックプロファイルデータ (-p オプション) とハードウェアカ

ウンタプロファイルデータ (-h オプション) だけを収集できます。トレースデータはサポートされていません。

-p オプションを明示的に指定せずに -h オプションを使用すると、クロックプロファイリングが無効となります。ハードウェアカウンタデータとクロックデータの両方を収集するには、-h オプションと -p オプションの両方を指定する必要があります。

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

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

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

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

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

   コレクタのオプションについては、63 ページの「データ収集オプション」を参照してください。クロックプロフィリングについては、63 ページの「-p *option* を使用したクロックプロファイリング」を参照してください。ハードウェアクロックプロファイリングについては、64 ページの「collect -h を使用したハードウェアカウンタプロファイリング」を参照してください。

dbx collector サブコマンドによるデータの収集

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

▼ dbx からコレクタを実行する

1. **次のコマンドを使用して、dbx にプログラムをロードします。**

   ```
   % dbx program
   ```

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

```
(dbx) collector subcommand
```

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

```
(dbx) help collector
```

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

3. **dbx のオプションを設定し、プログラムを実行します。**

   指定したサブコマンドに誤りがある場合は、警告メッセージが出力され、サブコマンドは無視されます。以降のセクションでは、`collector` サブコマンドの完全なリストを示します。

データ収集のサブコマンド

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

profile *option*

クロックプロファイリングデータの収集を制御します。*option* に使用できる値は次のとおりです。

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

    デフォルトの設定は約 10 ミリ秒です。

デフォルトでは、hwprofile サブコマンドを使用してハードウェアカウンタプロファイリングデータ収集が有効になっていないかぎり、コレクタはクロックプロファイリングデータを収集します。

hwprofile *option*

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

- on - ハードウェアカウンタプロファイリングを有効にします。デフォルトでは、通常のオーバーフロー値で cycles カウンタのデータが収集されます。
- off - ハードウェアカウンタプロファイリングを無効にします。
- list - 使用可能なカウンタの一覧を返します。一覧の説明については、27 ページの「ハードウェアカウンタリスト」を参照してください。ハードウェアカウンタプロファイリング機能がシステムでサポートされていない場合は、dbx から警告メッセージが返されます。
- counter *counter-definition*... [, *counter-definition*] – カウンタの定義は次の形式です。

  [+]*counter-name*[~ *attribute_1*=*value_1*]...[~*attribute_n* =*value_n*][/ *register-number*] [,*interval*]

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

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

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

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

    デフォルトでは、コレクタは、ハードウェアカウンタプロファイリングデータを収集しません。ハードウェアカウンタプロファイリングが有効になっていて profile コマンドが発行されていない場合、クロックプロファイリングは無効となります。

56 ページの「ハードウェアカウンタプロファイリングに関する制限事項」も参照してください。

synctrace *option*

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

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

    デフォルトでは、コレクタは同期待ちのトレースデータを収集しません。

heaptrace *option*

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

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

デフォルトでは、コレクタはヒープのトレースデータを収集しません。

tha *option*

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

- `off` – スレッドアナライザのデータ収集を無効にします。
- `all` – すべてのスレッドアナライザデータを収集します。

- `race` - データ競合検出データを収集します。
- `deadlock` - デッドロックと潜在的デッドロックのデータを収集します。

スレッドアナライザの詳細は、『Oracle Developer Studio 12.5: スレッドアナライザユーザーズガイド』および `tha`.1 のマニュアルページを参照してください。

sample *option*

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

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

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

dbxsample { **on** | **off** }

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

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

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

実験制御のサブコマンド

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

disable サブコマンド

データ収集を無効にします。プロセスが実行中でデータが収集されている場合、その実験が終了し、データ収集は無効になります。プロセスが動作中でデータ収集が無効になっている場合、

警告が出され、このサブコマンドは無視されます。プロセスが動作していない場合は、以降の実行のデータ収集が無効になります。

enable サブコマンド

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

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

pause サブコマンド

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

resume サブコマンド

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

sample record *name* サブコマンド

name のラベルが付いた標本パケットを記録します。このラベルは、パフォーマンスアナライザの「選択の詳細」タブに表示されます。

出力サブコマンド

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

`archive` *mode* サブコマンド

実験をアーカイブするためのモードを設定します。*mode* に使用できる値は次のとおりです。

- `on` - ロードオブジェクトを通常の方法でアーカイブします。
- `off` - ロードオブジェクトをアーカイブしません。
- `copy` - 通常のアーカイブに加えて、ロードオブジェクトを実験にコピーします。

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

`limit` *value* サブコマンド

記録するプロファイリングデータの量を *value*M バイトに制限します。この制限は、クロックプロファイリングデータ、ハードウェアカウンタプロファイリングデータ、および同期待ちのトレースデータの合計に適用されますが、標本ポイントには適用されません。この制限値は概数にすぎないので、この値を超えることは可能です。

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

デフォルトでは、記録されるデータサイズに制限はありません。

`store` *option* サブコマンド

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

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

情報のサブコマンド

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

show サブコマンド

すべてのコレクタ制御の、現在の設定を表示します。

status サブコマンド

開かれている実験のステータスを報告します。

Oracle Solaris プラットフォーム上での dbx による動作中のプロセスからのデータの収集

Oracle Solaris プラットフォーム上では、コレクタを使用して、動作中のプロセスからデータを収集できます。そのプロセスがすでに dbx の制御下にある場合は、プロセスを一時停止し、以前のセクションで説明された方法を使用してデータ収集を有効にすることができます。実行中のプロセスからデータ収集を開始する操作は、Linux プラットフォームではサポートされていません。

プロセスが dbx の制御下にない場合は、collect –P *pid* コマンドを使用して、実行中のプロセスからデータを収集できます。詳細は、81 ページの「collect ユーティリティーによる動作中のプロセスからのデータの収集」を参照してください。プロセスに dbx を接続し、パフォーマンスデータを収集し、プロセスから切り離して、プロセスの実行を継続することもできます。選択した派生プロセスのパフォーマンスデータを収集するには、各プロセスに dbx を接続する必要があります。

▼ dbx の制御下にない実行中のプロセスからデータを収集する方法

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

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

2. **-P*PID* を指定して `collect` コマンドを呼び出し、データを収集します。**

   ■ **あるいは、次の手順を使用して手動で行うこともできます。**

   a. **プロセスに接続します。**
   dbx から、次のコマンドを入力します。

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

   dbx がまだ実行されていない場合、次のコマンドを入力します。

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

   実行中のプロセスに接続すると、そのプロセスが一時停止します。
   プロセスへの接続についての詳細は、マニュアル『Oracle Developer Studio 12.5: dbx コマンドによるデバッグ』を参照してください。

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

   c. **プロセスから切り離します。**
   データの収集を完了したら、プログラムを一時停止し、dbx からプロセスを切り離します。
   dbx から、次のコマンドを入力します。

   ```
   (dbx) detach
   ```

動作中のプロセスからのトレースデータの収集

いずれかの種類のトレースデータを収集するには、プログラムを実行する前にコレクタライブラリ `libcollector.so` をプリロードする必要があります。ヒープトレースデータまたは同期待ちトレースデータを収集するには、`er_heap.so` および `er_sync.so` もそれぞれプリロードする必要があります。これらのライブラリは、データ収集を実行するための実際の関数へのラッパーを提供します。また、コレクタは、ほかのシステムライブラリの呼び出しにもラッパー関数を追加し、それによって完全なパフォーマンスデータを確保できます。ライブラリをプリロードしないと、これらのラッパー関数を挿入できません。コレクタがシステムライブラリ関数に割り込む方法の詳細については、46 ページの「システムライブラリの使用」を参照してください。

`libcollector.so` をプリロードするには、次の表に示すように、環境変数を使用してライブラリ名とライブラリへのパスを設定する必要があります。`LD_PRELOAD` 環

境変数を使用して、ライブラリ名を設定します。ライブラリへのパスを設定するには、LD_LIBRARY_PATH、LD_LIBRARY_PATH_32、または LD_LIBRARY_PATH_64 環境変数を使用します。LD_LIBRARY_PATH は、_32 および _64 バリアントが定義されていない場合に使用されます。これらの環境変数をすでに定義している場合は、新しい値を追加してください。

表 7 libcollector.so、er_sync.so、および er_heap.so をプリロードするための環境変数設定

環境変数	値
LD_PRELOAD	libcollector.so
LD_PRELOAD	er_heap.so
LD_PRELOAD	er_sync.so
LD_LIBRARY_PATH	/opt/developerstudio12.5/lib/analyzer/runtime
LD_LIBRARY_PATH_32	/opt/developerstudio12.5/lib/analyzer/runtime
LD_LIBRARY_PATH_64	/opt/developerstudio12.5/lib/analyzer/v9/runtime
LD_LIBRARY_PATH_64	/opt/developerstudio12.5/lib/analyzer/amd64/runtime

Oracle Developer Studio ソフトウェアが /opt/developerstudio12.5 にインストールされていない場合は、システム管理者に正しいパスを問い合わせてください。LD_PRELOADにフルパスを設定することもできますが、そのようにすると、SPARC® V9 の 64 ビットアーキテクチャーを使用するときに問題が発生する可能性があります。

注記 - 実行が終了したら LD_PRELOAD と LD_LIBRARY_PATH の設定を削除して、これらの設定が、同じシェルから起動されるほかのプログラムに対して有効なままにならないようにしてください。

スクリプトからのデータの収集

collect コマンドのターゲットとしてスクリプトを指定できます。ターゲットがスクリプトである場合、collect はデフォルトで、そのスクリプトを実行するために起動されたプログラムとすべての派生プロセスに関するデータを収集します。

特定のプロセスに関するデータのみを収集するには、-F オプションを使用して、追跡する実行可能ファイルの名前を指定します。

たとえば、スクリプト start.sh をプロファイルするが、主に実行可能ファイル myprogram からのデータを収集するには、次のコマンドを使用します。

```
$ collect -F =myprogram start.sh
```

start.sh スクリプトを実行するために起動された親プロセスと、そのスクリプトから生成されたすべての myprogram プロセスに関するデータが収集されますが、その他のプロセスのデータは収集されません。

collect と ppgsz を組み合わせた使用法

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

子プロセスは、パスの最初のディレクトリに存在する名前付きターゲットの exec を試み、exec の試みが成功するまで、順に次のディレクトリについて同様な操作を行います。たとえば、3 番目の試みが成功した場合、最初の 2 つの派生実験には _x1_f1_x1.er および _x1_f1_x2.er という名前が付けられますが、これらは両方とも完全に空です。ターゲット上の実験は、成功した 3 回目の exec によるもので、_x1_f1_x3.er という名前が付けられ、親の実験の下に格納されます。これは、test.1.er/_x1_f1_x3.er に対してアナライザまたは er_print ユーティリティーを起動することによって直接処理できます。

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

MPI プログラムからのデータの収集

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

コレクタは、Oracle Message Passing Toolkit 8 (以前の Sun HPC ClusterTools 8) とその更新をサポートしています。コレクタは、その他のバージョンの MPI も認識できます。ほかの引数を指定せずに collect -h を実行すると、MPI の有効なバージョンの一覧が表示されます。

Oracle Message Passing Toolkit MPI ソフトウェアは http://www.oracle.com/us/products/tools/message-passing-toolkit-070499.html で入手して、Oracle Solaris 10 および Linux システムにインストールできます。

Oracle Message Passing Toolkit は、Oracle Solaris 11 リリースの一部として入手可能になりました。システムにインストールされている場合は、/usr/openmpi にあります。まだ Oracle Solaris 11 システムにインストールされていない場合、システムにパッケージリポジトリが構成されていれば、コマンド pkg search openmpi を使用してこのパッケージを検索できます。Oracle Solaris 11 でのソフトウェアのインストールの詳細は、Oracle Solaris 11 ドキュメントライブラリのドキュメント『*Oracle Solaris 11 ソフトウェアパッケージの追加および更新*』を参照してください。

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

MPI ジョブからデータを収集するには、collect コマンドを使用する必要があります。dbx collector サブコマンドは、MPI データ収集を開始するために使用できません。詳細は、93 ページの「MPI 用の collect コマンドの実行」を参照してください。

MPI 用の collect コマンドの実行

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

データを収集するには、次の構文を使用します。

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

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

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

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

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

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

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

MPI 実験の格納

マルチプロセス環境は複雑になることがあるので、MPI プログラムからパフォーマンスデータを収集するときは、MPI 実験の格納に関するいくつかの問題に留意する必要があります。これらの問題は、データの収集と保管の効率、および実験の命名に関連しています。MPI 実験を含む、実験の命名については、58 ページの「収集データの格納場所」を参照してください。

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

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

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

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

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

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

バッファーのサイズを変更するには、環境変数 VT_BUFFER_SIZE を使用します。この変数の最適値は、トレース対象のアプリケーションによって異なります。小さい値を設定すると、アプリケーションで利用できるメモリーが増えますが、MPI API トレースコレクタによってバッファーのフラッシュが頻繁に引き起こされます。このようなバッファーのフラッシュによって、アプリケーションの動作が大幅に変化する可能性があります。一方、2G バイトなどの大きい値を設定すると、MPI API トレースコレクタによるバッファーフラッシュは最小限に抑えられますが、アプリケーションで利用できるメモリーが減少します。バッファーやアプリケーションデータを保持するために十分なメモリーを利用できない場合、アプリケーションの一部がディスクにスワップされて、アプリケーションの動作が大幅に変化する可能性もあります。

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

通常、MPI トレース出力データは、mpirun ターゲットが終了すると後処理されます。処理されたデータファイルが実験に書き込まれ、後処理時間に関する情報が実験のヘッダーに書き込まれます。MPI トレースが -m off により明示的に無効になっている場合、MPI 後処理は行われま

せん。後処理に失敗した場合、エラーが報告され、MPI タブと MPI トレースメトリックスは使用できなくなります。

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

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

注記 - コンピュータまたはノード間で実験のコピーあるいは移動を行うと、ソースファイル、または同じタイムスタンプを持つコピーにアクセスできないかぎり、注釈付きソースコードまたは注釈付き逆アセンブリコードのソース行を表示できません。注釈付きソースを表示するため、元のソースファイルへのシンボリックリンクを現在のディレクトリに置くことができます。「設定」ダイアログボックスで設定を使用することもできます。「検索パス」タブ (137 ページの「検索パス設定」を参照) を使用すると、ソースファイルの検索に使用されるディレクトリのリストを管理できます。「パスマップ」タブ (138 ページの「パスマップ設定」を参照) を使用して、ある場所から別の場所へのファイルパスの先頭部分をマップします。

♦ ♦ ♦ 第 4 章

4 パフォーマンスアナライザツール

この章では、パフォーマンスアナライザグラフィカルデータ解析ツールの次の内容について説明します。

- 97 ページの「パフォーマンスアナライザについて」
- 98 ページの「パフォーマンスアナライザの起動」
- 101 ページの「パフォーマンスアナライザのユーザーインタフェース」
- 103 ページの「パフォーマンスアナライザのビュー」
- 124 ページの「ライブラリとクラスの可視性の設定」
- 124 ページの「データのフィルタリング」
- 127 ページの「パフォーマンスアナライザからのアプリケーションのプロファイリング」
- 128 ページの「実行中のプロセスのプロファイリング」
- 129 ページの「実験の比較」
- 130 ページの「パフォーマンスアナライザのリモートでの使用」
- 131 ページの「構成設定」

パフォーマンスアナライザについて

パフォーマンスアナライザは、コレクタによって収集されたパフォーマンスデータを解析するグラフィカルデータ解析ツールです。コレクタは、パフォーマンスアナライザからアプリケーションをプロファイルするとき、または `collect` コマンドを使用したときに起動します。第3章「パフォーマンスデータの収集」で説明されているように、プロセスの実行中に、コレクタはパフォーマンス情報を収集して実験を作成します。

パフォーマンスアナライザはこれらの実験を読み取り、そのデータを解析して表や図に表示します。

注記 - パフォーマンスアナライザのデモコードは、Oracle Developer Studio 12.5 の Web ページ (`http://www.oracle.com/technetwork/server-storage/solarisstudio`) から、サンプルアプリケーション zip ファイルとしてダウンロードできます。

パフォーマンスアナライザでのサンプルコードの使用方法については、『Oracle Developer Studio 12.5: パフォーマンスアナライザチュートリアル』を参照してください。

パフォーマンスアナライザの起動

パフォーマンスアナライザを起動するには、端末ウィンドウで次のコマンドを入力します。

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

実験名または実験リストを指定できます。*experiment-list* コマンド引数は、実験名、実験グループ名、またはその両方からなるスペース区切りのリストです。実験リストを指定しない場合は、パフォーマンスアナライザが起動し、「ようこそ」ページを開きます。

コマンド行では、複数の実験または実験グループを指定できます。内部に派生実験を含む実験を指定した場合は、すべての派生実験が自動的にロードされ、それらのデータが集約されます。個々の派生実験をロードするには、各実験を明示的に指定するか、または実験グループを作成する必要があります。

派生を含む実験の読み取りでは、パフォーマンスデータをほとんどまたはまったく含まないサブ実験はすべて、パフォーマンスアナライザや `er_print` によって無視されます。

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

```
#analyzer experiment group
```

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

パフォーマンスアナライザに複数の実験が表示される場合、デフォルトでは、すべての実験のデータが集約されます。データはまとめられ、1 つの実験から取得されたものであるかのように表示されます。ただし、`-c` オプションを指定することにより、データを集約するのではなく、実験を比較することも選択できます。129 ページの「実験の比較」を参照してください。

「実験を開く」ダイアログボックスで名前をクリックすることによって、実験または実験グループをプレビューできます。

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

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

パフォーマンスアナライザが起動され、「アプリケーションのプロファイル」ウィンドウに、指定されたターゲットとその引数、およびアプリケーションをプロファイルし、実験を収集するための設定が表示されます。詳細は、127 ページの「パフォーマンスアナライザからのアプリケーションのプロファイリング」を参照してください。

「ライブ」実験 - 引き続き収集が行われている実験を開くこともできます。ライブ実験を開くと、その実験を開いたときにすでに収集が完了しているデータのみが表示されます。新しいデータが入力されても実験が自動的に更新されることはありません。更新するために、実験を再度開くことができます。

analyzer コマンドオプション

これらの analyzer コマンドオプションはパフォーマンスアナライザの動作を制御するものであり、次のグループに分けられます。

- 実験オプション
- Java オプション
- 制御オプション
- 情報オプション

実験オプション

これらのオプションは、ユーザーがコマンド行で指定する実験の処理方法を指定します。

-c *base-group compare-group*

パフォーマンスアナライザを起動し、指定された実験を比較します。

base-group は 1 つの実験か、または複数の実験を指定する *groupname*.erg ファイルのどちらかです。*compare-group* は、ベースグループと比較する 1 つ以上の実験です。

比較グループに複数の実験を指定するには、実験名を空白で区切ります。また、比較グループに複数の実験を指定する *groupname*.erg ファイルを指定することもできます。

例 3 比較モードで実験を開くためのコマンド例

実験 test.1.er を開き、それを test.4.er と比較します。

```
% analyzer -c test.1.er test.4.er
```

実験グループ demotest.erg を開き、それを test.4.er と比較します。

```
% analyzer -c demotest.erg test.4.er
```

analyzer コマンドの Java オプション

これらのオプションは、パフォーマンスアナライザを実行する JVM の設定を指定します。

-j | --jdkhome *jvm-path*

パフォーマンスアナライザを実行するための Java ソフトウェアへのパスを指定します。-j オプションが指定されていない場合、デフォルトのパスは、まず JVM へのパスの環境変数を最初

に JDK_HOME、次に JAVA_PATH の順序で検査することによって取得されます。どちらの環境変数も設定されていない場合は、現在の PATH にある JVM が使用されます。デフォルトのパスをすべて無効にするには、-j オプションを使用します。

-J *jvm-option*

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

- 64 ビットのパフォーマンスアナライザを実行するには、次のように入力します。

  ```
  analyzer -J-d64
  ```

- 最大 2G バイトの JVM メモリーを使用してパフォーマンスアナライザを実行するには、次のように入力します。

  ```
  analyzer -J-Xmx2G
  ```

- 最大 8G バイトの JVM メモリーを使用して 64 ビットのパフォーマンスアナライザを実行するには、次のように入力します。

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

analyzer コマンドの制御オプション

これらの analyzer コマンドオプションは、設定が格納されるユーザーディレクトリの場所を制御したり、ユーザーインタフェースのフォントサイズを設定したり、パフォーマンスアナライザを起動する前にバージョンと実行時の情報を表示したりします。

-f | --fontsize *size*

パフォーマンスアナライザのユーザーインタフェースで使用されるフォントサイズを指定します。

パフォーマンスアナライザを起動し、14 ポイントのフォントでメニューを表示するには、次のように入力します。

```
analyzer -f 14
```

-v | --verbose

バージョン情報と Java 実行時引数を表示してから、パフォーマンスアナライザを起動します。

analyzer コマンドの情報オプション

これらのオプションは、analyzer に関する情報を標準出力に出力します。次の個々のオプションは、スタンドアロンのオプションです。これは analyzer のほかのオプションや、target 引数または experiment-list 引数と組み合わせることはできません。

-V | --version

バージョン情報のみを表示し、パフォーマンスアナライザは起動しません。

-? | -h | --help

使用方法に関する情報を出力して終了します。

パフォーマンスアナライザのユーザーインタフェース

「パフォーマンスアナライザ」ウィンドウには、メニューバー、ツールバー、およびナビゲーションパネルがあります。これらを使用して、パフォーマンスアナライザやデータをやりとりできます。

メニューバー

メニューバーには、次のコマンドメニューが含まれています。

- 「ファイル」には、アプリケーションをプロファイルしたり、実験を開いたり、複数の実験のデータを比較または集約したりするために使用できるいくつかのオプションが用意されています。その他のオプションには、データビューの内容をいくつかの異なる形式にエクスポートしたり、リモートホストに接続し、そこに配置されているファイルやアプリケーションに対してパフォーマンスアナライザを使用する機能が含まれます。
- 「ビュー」を使用すると、パフォーマンスアナライザのナビゲーションバーに表示するデータビューを選択できます。また、「ビュー +」ボタンまたはナビゲーションバーの「詳細ビュー」をクリックしてさらにデータビューを追加することもできます。
- 「メトリック」を使用すると、データビューにメトリックを表示できます。
- 「ツール」には、データのフィルタリング、データビューでのライブラリ関数の非表示化、およびその他の設定のオプションが用意されています。
- 「ヘルプ」には、新機能、クイックリファレンス、キーボードショートカット、およびトラブルシューティング情報へのリンクを含む、パフォーマンスアナライザの統合されたドキュメントへのリンクが用意されています。また、キーボードの F1 キーを押して、パフォーマンスアナライザ内の現在のビューに関する情報を表示することもできます。

ツールバー

ツールバーには、コマンドのショートカットとして使用できるボタン、Java 実験や OpenMP 実験データの表示方法の変更に使用できる表示モードセレクタ、およびデータビュー内のテキストの検索に役立つ「検索」ツールが用意されています。

図 2 パフォーマンスアナライザのツールバー

ナビゲーションパネル

左側にある垂直ナビゲーションパネルを使用すると、パフォーマンスアナライザ内の*データビュー*または単に*ビュー*と呼ばれるさまざまなページを選択できます。ほとんどのビューには、プロファイルされたアプリケーションのパフォーマンスメトリックが表示されます。これらのビューは関連しているため、あるビューでの選択や適用したフィルタはその他のビューにも適用されます。

ナビゲーションパネルの一部のボタンは、このツールを使いやすくし、情報の検索も容易にする「ようこそ」ページや、開かれている実験内のデータに関する情報を表示する「概要」ページなどのページを開きます。

ナビゲーションパネルの上部にある「+」ボタン、またはナビゲーションパネルの下部にある「詳細ビュー」ボタンをクリックすると、ナビゲーションパネルにさらにビューを追加できます。また、「ビュー」メニューを使用して、追加する別のビューも選択できます。

ほとんどのデータビュー内の項目は、クリックするとその項目に関する詳細情報を「選択の詳細」ウィンドウに表示できます。このウィンドウは、次のセクション 102 ページの「「選択の詳細」ウィンドウ」で説明されています。

データビューは、103 ページの「パフォーマンスアナライザのビュー」で説明されています。

「選択の詳細」ウィンドウ

右側にある「選択の詳細」ウィンドウには、選択した項目について記録されたすべてのメトリック (値とパーセンテージの両方) や、選択した関数またはロードオブジェクトに関する情報が表示されます。「選択の詳細」ウィンドウは、いずれかのデータビューで新しい項目を選択するたびに更新されます。

「タイムライン」ビューの「標本」バーで標本を選択すると、「選択の詳細」ウィンドウには、その標本番号、標本の開始時間と終了時間、および各マイクロステートで費やされた時間とカラーコードを含むマイクロステートが表示されます。

「タイムライン」ビューのデータバーでイベントを選択すると、「選択の詳細」ウィンドウには、そのイベントの詳細と呼び出しスタックが表示されます。

「呼び出し元/呼び出し回数」パネル

一部のデータビューの下部にある「呼び出し元/呼び出し回数」パネルを使用すると、呼び出しパスをナビゲートできます。ビュー内の関数を選択してから「呼び出し元/呼び出し回数」パネルを使用すると、呼び出し元の関数または呼び出し先の関数にナビゲートできます。「呼び出し元/呼び出し回数」で関数をクリックすると、データビュー内でその関数が選択されます。

パフォーマンスアナライザのビュー

実験を開いたときにナビゲーションバーにデータビューが表示されるかどうかは、次の要因によって決定されます。

- その実験内のデータの種類によって、どのようなデータビューが表示されるかが決定されます。たとえば、実験に OpenMP データが含まれている場合は、そのデータを表示するために OpenMP のビューが自動的に開かれます。
- パフォーマンスアナライザの起動時に読み取られた構成ファイルによって、表示するデフォルトのデータビューが指定されます。

「ビュー」メニューまたはボタンを使用して「設定」ダイアログボックス
(132 ページの「「ビュー」設定」を参照) を開くことにより、現在のパフォーマンスアナライザセッションに表示するビューを選択できます。

ほとんどのビューには、そのビュー内の項目を右クリックすると開くコンテキストメニューがあります。コンテキストメニューを使用すると、フィルタを追加したり、そのデータビューに関連したその他のアクティビティーを実行したりできます。あるビューでフィルタを適用すると、フィルタできるすべてのビュー内のデータがフィルタされます。

ナビゲーションバーには、次のものが表示されます。

- 104 ページの「「ようこそ」ページ」
- 105 ページの「「概要」画面」
- 107 ページの「「関数」ビュー」
- 107 ページの「「タイムライン」ビュー」
- 109 ページの「「ソース」ビュー」
- 111 ページの「「呼び出しツリー」ビュー」
- 111 ページの「「呼び出し元-呼び出し先」ビュー」
- 113 ページの「インデックスオブジェクトビュー」

- 113 ページの「「スレッド」ビュー」
- 113 ページの「「標本」ビュー」
- 113 ページの「「CPU」ビュー」
- 114 ページの「「秒」ビュー」
- 114 ページの「「プロセス」ビュー」
- 114 ページの「「実験 ID」ビュー」
- 115 ページの「「メモリーオブジェクト」ビュー」
- 116 ページの「「データオブジェクト」ビュー」
- 115 ページの「「データレイアウト」ビュー」
- 116 ページの「「I/O」ビュー」
- 117 ページの「「ヒープ」ビュー」
- 117 ページの「「データサイズ」ビュー」
- 117 ページの「「経過時間」ビュー」
- 118 ページの「「OpenMP 並列領域」ビュー」
- 118 ページの「「OpenMP タスク」ビュー」
- 119 ページの「「行」ビュー」
- 119 ページの「「PC」ビュー」
- 119 ページの「「逆アセンブリ」ビュー」
- 120 ページの「「ソース/逆アセンブリ」ビュー」
- 121 ページの「「競合」ビュー」
- 121 ページの「「デッドロック」ビュー」
- 121 ページの「「デュアルソース」ビュー」
- 122 ページの「「命令頻度」ビュー」
- 121 ページの「「統計」ビュー」
- 121 ページの「「実験」ビュー」
- 122 ページの「「MPI タイムライン」ビュー」
- 123 ページの「「MPI チャート」ビュー」

「ようこそ」ページ

どの実験を開くかをコマンド行で指定せずにパフォーマンスアナライザを起動した場合、表示される最初のページは「ようこそ」ページです。「ようこそ」ページでは、アプリケーションのプロファイリングを開始したり、最近の実験を表示したり、実験を比較したり、ドキュメントを表示したりすることが容易になります。実験を開いたあとも引き続き、パフォーマンスアナライザセッション中にナビゲーションパネルからいつでも「ようこそ」ページを選択できます。

次の図は、「ようこそ」ページを示しています。

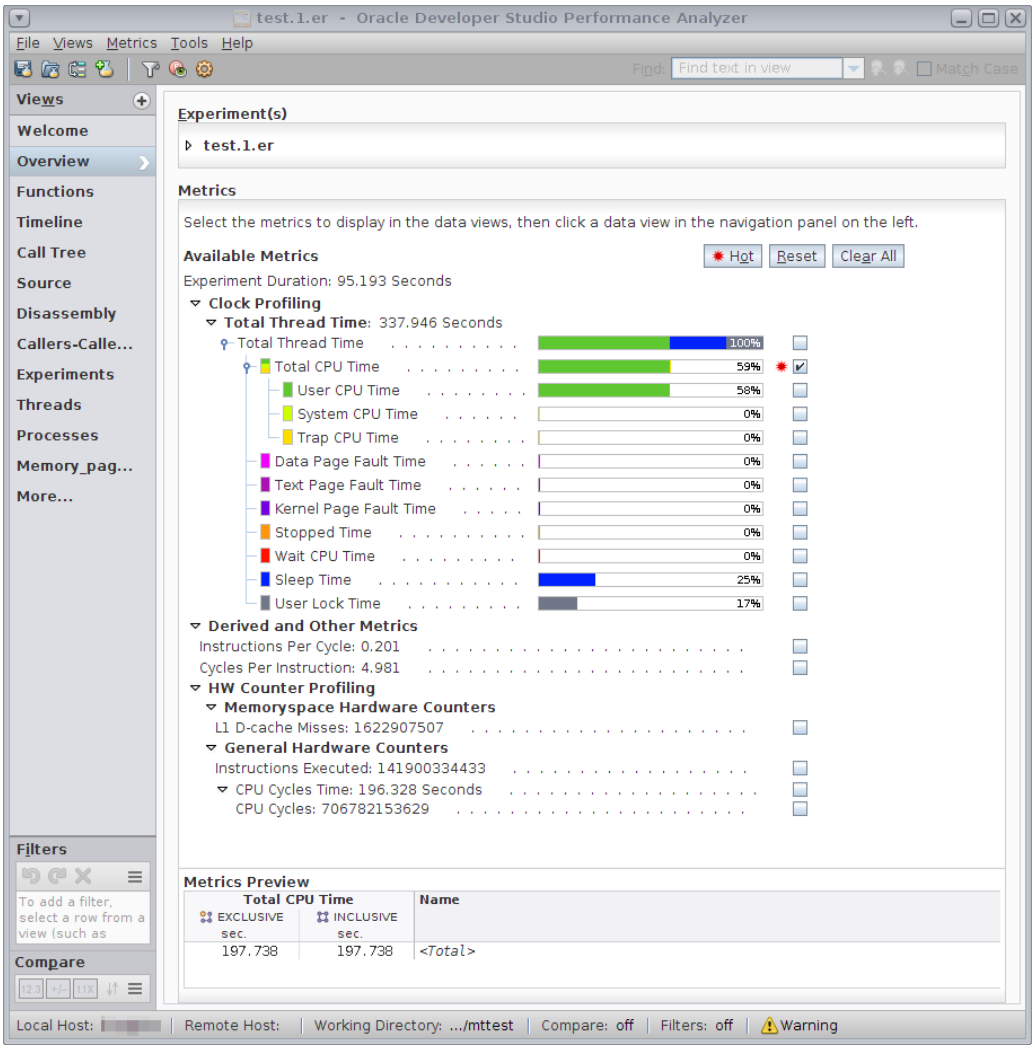
図 3　　パフォーマンスアナライザの「ようこそ」ページ

「概要」画面

実験を開くと、パフォーマンスアナライザには「概要」画面が表示されます。この画面には、ロードされた実験のパフォーマンスメトリックが表示されるため、メトリックのどの部分が高いかをすばやく把握できます。「概要」を使用して、ほかのビューで調べるメトリックを選択します。「概要」に関する詳細を表示するには、「概要」の表示中に F1 キーを押してください。

次の図は、「概要」が表示された「パフォーマンスアナライザ」ウィンドウを示しています。

図 4 パフォーマンスアナライザの「概要」画面



メトリックを選択すると、このウィンドウの下部にある「メトリックのプレビュー」パネルに、このメトリックがデータビューにどのように表示されるかが示されます。選択ボタンを使用するか、関心のあるメトリックを具体的に選択することによって、データビューに表示するメトリックを指定できます。詳細は、ヘルプで「概要」画面のトピックを参照してください。

「関数」ビュー

「関数」ビューには、実験で収集されたデータから派生した、ターゲットプログラムの関数とそのメトリックのリストが表示されます。メトリックスは、排他的メトリックスまたは包括的メトリックスのいずれかになります。排他的メトリックは、その関数自体のみによる使用率を表します。包括的メトリックスは、関数とその関数が呼び出したすべての関数による使用を表します。メトリックの詳細は、36 ページの「関数レベルのメトリックス: 排他的、包括的、属性」を参照してください。

収集されたデータの種類ごとに使用可能なメトリックのリストは、`collect`(1) のマニュアルページとヘルプ、および第2章「パフォーマンスデータ」に記載されています。

時間メトリックは秒で表示され、ミリ秒の精度で提供されます。パーセンテージは、0.01% の精度で表示されます。メトリック値がちょうどゼロの場合、その時間およびパーセンテージは「0」として表示されます。値がちょうどゼロではないが精度より小さい場合、値は「0.000」として、パーセンテージは「0.00」として表示されます。丸めのため、百分率は合計が正確に 100 % にならない場合もあります。カウントメトリックは整数カウントとして示されます。

最初に表示されるメトリックは、収集されたデータに基づいています。複数の種類のデータが収集された場合、各種類のデフォルトのメトリックが表示されます。どのメトリックを表示するかを「概要」ページで選択できます。詳細は、105 ページの「「概要」画面」を参照してください。

関数を検索するには、ツールバーの「検索」ツールを使用します。

関数のソースコードを表示するには、ソースコード内の適切な行でそれをダブルクリックして「ソース」ビューを開きます。

1 つの関数を選択するには、その関数をクリックして、その関数に関する詳細情報を右側にある「選択の詳細」ウィンドウに表示します。

ビュー内に連続して表示されている複数の関数を選択するには、グループの最初の関数を選択したあと、Shift キーを押しながらグループの最後の関数をクリックします。

ビュー内に連続して表示されていない複数の関数を選択するには、グループの最初の関数を選択したあと、Ctrl キーを押しながら各関数をクリックして追加の関数を選択します。

「関数」ビュー内で右クリックしてコンテキストメニューを開き、選択した関数の定義済みフィルタを選択することもできます。フィルタリングの詳細は、パフォーマンスアナライザのヘルプを参照してください。

「タイムライン」ビュー

「タイムライン」ビューには、時間の関数として記録されたイベントおよび標本ポイントのチャートが表示されます。データは、水平バーに表示されます。デフォルトでは、各実験について、CPU 使用率標本のバーが上部に 1 つと、各スレッドに対応する一連のプロファイリングデータのバー

が表示されます。各スレッドに対して表示されるデータは、アプリケーションをプロファイルしたときに収集したデータによって決まります。

「タイムライン」ビューを使用すると、プログラムで次のことを実行できます。

- プログラムのアクティビティーの段階を識別する
- カスタム配色を使用して関数およびメソッドを強調表示する
- 時間軸および垂直軸の方向にズームインして詳細を確認する
- 「選択の詳細」パネルで詳細なイベント情報を確認する
- 大域データを時間範囲、スレッド、および CPU によってフィルタリングする

次のデータバーが表示されることがあります。

CPU 使用率標本

実験に標本データが含まれている場合、上部に表示されるバーは「CPU 使用率標本」です。標本ポイントのデータは、そのポイントと前のポイントの間で費やされた CPU 時間を表します。標本データには、Oracle Solaris システムで使用可能なマイクロステート情報が含まれています。

Oracle Solaris オペレーティングシステムは、マイクロステートアカウンティングというテクノロジを使用して、イベントごとに実行状態に関する統計を収集します。パフォーマンスアナライザによって示されるイベントの時間メトリックは、各状態で費やされた相対的な時間に対応します。CPU 使用率標本には、実験内のすべてのスレッドの時間メトリックのサマリーが表示されます。標本をクリックすると、その標本の時間メトリックが、右側の「選択の詳細」パネルに表示されます。イベント状態を表示することによって、タイムライン内の個々のイベントの時間メトリックを表示できます。

プロファイリングおよびトレースデータバー

クロックプロファイリング、ハードウェアカウンタプロファイリング、およびトレースデータのデータバーには、記録された各イベントのイベントマーカーが表示されます。イベントマーカーは、イベントとともに記録された呼び出しスタックの色分けされた表示で構成されます。

イベントマーカーをクリックすると、そのイベントに関する情報が「選択の詳細」パネルに表示され、呼び出しスタックの関数が「呼び出しスタック」パネルに表示されます。「呼び出しスタック」パネルの関数をダブルクリックすると、「ソース」ビューに移動し、その関数のソースがメトリックとともに表示されます。

データの種類によっては、イベントが重なって見えない場合があります。まったく同じ位置に複数のイベントが表示される場合は、常に 1 つだけが描画されます。1 - 2 ピクセル以内に複数のイベントがある場合は、すべてが描画されますが、見た目には判別できない可能性があります。どちらの場合も、イベントの下にイベントの境界を示す灰色の小さいティックマークが表示されます。イベントはズームインして表示できます。左または右矢印キーを使用してイベントの選択をいずれかの方向に移動すると、表示されていないイベントが表示されます。イベント密度を表示することによって、イベントに関する詳細情報を表示できます。

ヒープサイズ

「ヒープサイズ」データは、`heaptrace` 収集オプションが有効になっている場合にのみ使用できます。「ヒープサイズ」は 2 つのグラフで構成されます。「**正味の割り当てバイト数**」は、

割り当て済みバイト数から解放済みバイト数を引いた値を追跡します。「正味のリークバイト数」は、解放されることのない割り当て済みバイト数の累積を追跡します。

イベント状態

イベント状態は、さまざまな状態で費やされたアプリケーション時間の分布を時間の関数として表示する棒グラフで示されます。

Oracle Solaris に記録されたクロックプロファイリングデータの場合は、イベント状態チャートに Oracle Solaris のマイクロステートが表示されます。イベント状態のカラーコードは、CPU 使用率標本バーの場合と同じです。

イベント状態はデフォルトで表示されます。これらは、「タイムライン」ツールバーの「タイムライン設定」ボタンをクリックし、「設定」ダイアログボックスの「タイムライン」領域で「イベント状態」を選択解除することによって非表示にできます。

イベント密度

イベント密度は、イベントの頻度を時間の関数として表示する青い線で示されます。

イベント密度を表示するには、「タイムライン」ツールバーの「タイムライン設定」ボタン「タイムライン設定」アイコンをクリックし、「設定」ダイアログボックスの「タイムライン」領域で「イベント密度」を選択します。

イベント密度は、各データの種類のタイムラインデータバーの真下に表示されます。イベント密度には、横軸の各タイムスライスで発生したイベントのカウントが表示されます。折れ線チャートの縦軸のスケールは、0 からその特定のデータバーの表示可能な時間範囲内の最大イベントカウントまでです。

表示可能なタイムスライスのそれぞれに多くのイベントが含まれるようにタイムラインのズームを設定すると、イベント密度を使用してイベントの頻度が高い期間を特定できます。このような期間を調べるには、ズームインします。次に、右クリックしてコンテキストフィルタを選択し、表示可能な時間範囲のデータのみが含まれるように設定してから、パフォーマンスアナライザのほかのデータビューを使用してその期間のデータを分析します。

「ソース」ビュー

ソースコードが使用可能な場合、「ソース」ビューには、選択した関数のソースコードを含むファイルが、各ソース行の左側の列にパフォーマンスメトリックの注釈付きで表示されます。

高いメトリックスは黄色で表示され、リソース使用状況のホット領域であるソース行が示されます。また、オレンジ色のナビゲーションマーカーが、各ホットソース行の右側にあるスクロールバーの隣のマージン内に表示されます。ホットしきい値を下回るゼロ以外のメトリックは強調表示されませんが、灰色のナビゲーションマーカーが表示されます。

メトリックス付きのソース行にすばやく移動するには、右側マージン内の黄色のマーカーをクリックしてメトリックス付きの行にジャンプします。メトリックを含む次の行にジャンプするには、そのメトリック自体を右クリックし、「次のホットライン」や「次のゼロ以外のメトリック行」などのオプションを選択します。

「設定」ダイアログボックスの「ソース/逆アセンブリ」タブで、メトリックを強調表示するためのしきい値を設定できます。

「ソース」ビューには、ソースファイルとそれに対応するオブジェクトファイルへのフルパスが表示され、ソースコードの列見出しにはロードオブジェクトの名前が表示されます。複数のオブジェクトファイルをコンパイルするために同じソースファイルが使用されているまれなケースでは、「ソース」ビューには、選択された関数を含むオブジェクトファイルのパフォーマンスデータが表示されます。

パフォーマンスアナライザがソースファイルを見つけることができない場合は、「ソース」ビューの「解決」ボタンをクリックしてソースファイルを参照できます。あるいは、ソースへのパスを入力または参照すると、ソースコードが新しい場所から表示されます。オプションで、「実験内部のソースファイルをアーカイブ」を選択できます。これにより、ソースファイルが実験内にコピーされます。このボタンの詳細については、「ヘルプ」メニューで「ソースファイルの解決」のトピックを参照してください。

実験のソースコードを検索するために使用されるプロセスについては、221 ページの「ツールがソースコードを見つけるしくみ」を参照してください。

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

<Function: f_name>

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

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

別のソースコンテキストを参照するインデックス行をダブルクリックすると、そのソースコンテキストを含むファイルが、インデックスで示す関数に関連付けられた位置から開かれます。

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

ソースコードは、表示のために選択されているすべてのコンパイラ解説とともにインタリーブされます。表示される解説のクラスは、「設定」ダイアログボックスで設定できます。デフォルトのクラスは、`.er.rc` デフォルト値ファイルで設定できます。

「ソース」ビューに表示されるメトリックは、変更または再構成できます。詳細は、「ヘルプ」メニューを参照してください。

「ソース」ビューの内容の詳細は、223 ページの「パフォーマンスアナライザの「ソース」ビューのレイアウト」を参照してください。

「呼び出しツリー」ビュー

「呼び出しツリー」ビューには、プログラムの動的な呼び出しグラフがツリーとして表示されます。各関数呼び出しは、展開したり縮小したりできるノードとして表示されます。展開された関数ノードには、その関数によって行われる関数呼び出しがすべて表示され、加えて、それらの関数呼び出しのパフォーマンスメトリックスも表示されます。

ノードを選択すると、「選択の詳細」ウィンドウに、その関数呼び出しのメトリックとその呼び出し先が表示されます。属性付き測定基準に対して示されるパーセンテージは、プログラムの全測定基準のパーセンテージです。このツリーのデフォルトのルートは、<Total> で、これは関数ではありませんが、プログラムのすべての関数のパフォーマンスメトリックの 100% を表しています。

「呼び出しツリー」ビューは、特定の呼び出しトレースの詳細を表示し、どのトレースがパフォーマンスにいちばん大きく影響しているかを解析するために使用します。高いメトリック値を探しながら、プログラムの構造内を移動できます。

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

選択された分岐または選択された関数の定義済みフィルタを設定するには、「呼び出しツリー」ビュー内で右クリックしてコンテキストメニューを開きます。この方法でフィルタリングすることによって、対象外の領域に対応する、すべてのアナライザビュー内のデータを画面から除外できます。

「呼び出し元-呼び出し先」ビュー

「呼び出し元-呼び出し先」ビューには、コード内の関数間の呼び出し関係がパフォーマンスメトリックとともに表示されます。「呼び出し元-呼び出し先」ビューを使用すると、呼び出しスタックフラグメントを 1 回に呼び出し 1 つずつ構築することによって、コード分岐がないかメトリックを詳細に検査できます。

このビューには、上部に「呼び出し元」パネル、中央に「スタックフラグメント」パネル、下部に「呼び出し先」パネルの 3 つの個別のパネルが表示されます。「呼び出し元-呼び出し先」ビューをはじめて開いたとき、「スタックフラグメント」パネル内の関数は、「関数」ビューや「ソース」ビューなどの、ほかのアナライザビューのいずれかで選択した関数になります。「呼び出し元」パネルには、「スタックフラグメント」パネル上の関数を呼び出す関数が一覧表示され、「呼び出し先」パネルには、「スタックフラグメント」パネル上の関数によって呼び出される関数が一覧表示されます。

呼び出しスタックに呼び出し元または呼び出し先を 1 回に呼び出し 1 つずつ追加することによって、1 つの関数を中心にして呼び出しスタックフラグメントを構築できます。

スタックフラグメントに呼び出しを追加するには、「呼び出し元」ペインまたは「呼び出し先」ペイン内の関数をダブルクリックするか、または関数を選択して「追加」ボタンをクリックします。

関数呼び出しを削除するには、呼び出しスタックフラグメントの上部または下部にある関数をダブルクリックするか、あるいは上部または下部の関数を選択して「削除」をクリックします。

ヒント - コンテキストメニューから「追加」や「削除」のタスクを実行するには、関数を右クリックし、適切なコマンドを選択します。

関数を呼び出しスタックフラグメントの先頭 (上部)、中心、または末尾 (下部) として設定するには、関数を選択して「先頭を設定」、「中心を設定」、または「末尾を設定」をクリックします。この新しい順序により、現在呼び出しスタックフラグメント内にあるほかの関数が「呼び出し元」領域または「呼び出し先」領域内の、スタックフラグメント内の選択された関数の新しい場所を基準とした適切な場所に移動します。

呼び出しスタックフラグメントへの変更履歴をたどるには、「スタックフラグメント」パネルの上にある「戻る」ボタンと「進む」ボタンを使用します。

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

「呼び出し元-呼び出し先」ビューの任意のパネルで関数を選択してから右クリックしてコンテキストメニューを開くと、フィルタを選択できます。データは、このビューおよびアナライザのすべてのデータビューでの選択に従ってフィルタされます。コンテキストフィルタの使用方法の詳細は、オンラインヘルプを参照してください。

「呼び出し元-呼び出し先」ビューには、属性メトリックが表示されます。

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

メトリックスについての詳細は、36 ページの「関数レベルのメトリックス: 排他的、包括的、属性」を参照してください。

インデックスオブジェクトビュー

各インデックスオブジェクトビューには、「スレッド」、「CPU」、「秒」などの、さまざまなインデックスオブジェクトに帰属するデータのメトリック値が表示されます。インデックスオブジェクトは階層構造ではないため、包括的メトリックと排他的メトリックは表示されません。各種類の単一のメトリックだけが表示されます。

「スレッド」、「CPU」、「標本」、「秒」、「プロセス」、「実験 ID」などの、いくつかのインデックスオブジェクトビューが事前に定義されています。これらのビューについては、下で個別に説明します。

カスタムインデックスオブジェクトを定義することもできます。「設定」ダイアログボックスの「カスタムビューを追加」ボタンをクリックし、「インデックスオブジェクトを追加」ダイアログボックスでオブジェクトの値を設定します。

「スレッド」ビュー

「スレッド」ビューには、スレッドとそのメトリックのリストが表示されます。スレッドはプロセスとスレッドのペアで表され、デフォルトでは合計 CPU 時間を示します。読み込まれた実験内にほかのメトリックスが存在する場合は、そのメトリックスもデフォルトで表示される可能性があります。「スレッド」ビューはデフォルトでは表示されません。これは「ビュー」メニューから選択できます。

「フィルタ」ボタンを使用すると、このビューやパフォーマンスアナライザのビューに表示されるデータをフィルタできます。

「CPU」ビュー

「CPU」ビューには、ターゲットのアプリケーションの実行を処理した CPU とそのメトリックのリストが表示されます。CPU は CPU 番号で表され、デフォルトでは合計 CPU 時間を示します。読み込まれた実験内にほかのメトリックスが存在する場合は、そのメトリックスもデフォルトで表示される可能性があります。「CPU」ビューが表示されていない場合は、「ビュー」メニューからこのビューを選択できます。

「フィルタ」ボタンを使用すると、このビューやパフォーマンスアナライザのビューに表示されるデータをフィルタできます。

「標本」ビュー

「標本」ビューには、標本ポイントのリストがそれらのメトリックとともに表示されます。メトリックには、ロードされた実験の各標本ポイントで記録されたマイクロステートが反映されます。標本

は標本番号で表され、デフォルトでは CPU 時間合計が表示されます。「概要」パネルまたは「設定」ダイアログボックスでその他のメトリックを選択した場合は、それらが表示されることもあります。「標本」ビューが表示されていない場合は、「ビュー」メニューからこのビューを選択できます。

「フィルタ」ボタンを使用すると、このビューやパフォーマンスアナライザのビューに表示されるデータをフィルタできます。

標本ポイントの詳細については、パフォーマンスアナライザヘルプの「標本ポイント」のトピックを参照してください。

「秒」ビュー

「秒」ビューには、実験で取り込まれたプログラム実行の各秒が、その秒内に収集されたメトリックとともに表示されます。「秒」ビューは、0 から始まって毎秒発生し、間隔を変更できない定期的な標本を表示する点で、「標本」ビューと異なります。「秒」ビューには、デフォルトでは、実行の秒数が合計 CPU 時間とともに一覧表示されます。読み込まれた実験内にほかのメトリックスが存在する場合は、そのメトリックスも表示される可能性があります。「概要」や「設定」ダイアログでほかのメトリックを選択した場合は、それらのメトリックも表示されます。

「フィルタ」ボタンを使用すると、このビューやパフォーマンスアナライザのビューに表示されるデータをフィルタできます。

「プロセス」ビュー

「プロセス」ビューには、アプリケーションによって作成されたプロセスのリストが、それらのメトリックとともに表示されます。プロセスはプロセス ID (PID) 番号で表され、デフォルトでは合計 CPU 時間メトリックを示します。読み込まれた実験内にほかのメトリックスが存在する場合は、そのメトリックスも表示される可能性があります。「概要」や「設定」ダイアログでほかのメトリックを選択した場合は、それらのメトリックも表示されます。

「プロセス」ビューを使用すると、もっとも多くのリソースを使用したプロセスを検索できます。切り離し、ほかのビューを使用して調査したい特定の一連のプロセスがある場合は、コンテキストメニューで使用可能なフィルタを使用して、ほかのプロセスをフィルタで除外できます。

「実験 ID」ビュー

「実験 ID」ビューには、アプリケーションで作成されたプロセスとそのメトリックのリストが表示されます。実験 ID はプロセス ID (PID) 番号で表され、デフォルトでは合計 CPU 時間のメトリックを示します。読み込まれた実験内にほかのメトリックスが存在する場合は、そのメトリックスも表示される可能性があります。「概要」や「設定」ダイアログでほかのメトリックを選択した

場合は、それらのメトリックも表示されます。これらのメトリックス値には、読み込まれた実験内の各標本ポイントで記録されたマイクロステートが反映されます。これらの値には、ロードされたそれぞれの実験で記録されたメトリックスの値またはパーセンテージが反映されます。

「メモリーオブジェクト」ビュー

各「メモリーオブジェクト」ビューには、ページなどのさまざまなメモリーオブジェクトに帰属するデータ領域メトリックのメトリック値が表示されます。ロードされた 1 つ以上の実験にデータ領域プロファイルが含まれている場合は、「設定」ダイアログボックスの「ビュー」タブで、ビューを表示するメモリーオブジェクトを選択できます。任意の数の「メモリーオブジェクト」ビューを表示できます。

さまざまな「メモリーオブジェクト」ビューが事前に定義されています。メモリーオブジェクトは、仮想ページおよび物理ページに対して、Vpage_8K、Ppage_8K、Vpage_64K などの名前で事前に定義されています。カスタムメモリーオブジェクトを定義することもできます。「設定」ダイアログボックスの「カスタムオブジェクトを追加」ボタンをクリックし、「メモリーオブジェクトを追加」ダイアログボックスでオブジェクトの値を設定します。

「データレイアウト」ビュー

「データレイアウト」ビューには、すべてのプログラムデータオブジェクトの注釈付きデータオブジェクトレイアウトがデータ派生メトリックデータとともに表示されます。このビューは、ハードウェアカウンタオーバーフロープロファイリングの拡張であるデータ領域プロファイリングを含む実験にのみ適用できます。詳細は、193 ページの「データ領域プロファイリングとメモリー領域プロファイリング」を参照してください。

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

「データレイアウト」ビューを表示するには、「設定」ダイアログボックス
(132 ページの「「ビュー」設定」を参照) の「ビュー」タブでこのビューを選択します。「データオブジェクト」ビューと同様に、「データレイアウト」ビューを表示可能にできるのは、ロードされた実験の 1 つ以上にデータ領域プロファイルが含まれている場合だけです。

単一のデータオブジェクトを選択するには、そのオブジェクトをクリックします。

ビュー内に連続して表示されている複数のオブジェクトを選択するには、最初のオブジェクトを選択したあと、Shift キーを押しながら最後のオブジェクトをクリックします。

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

「データオブジェクト」ビュー

「データオブジェクト」ビューには、データオブジェクトのリストとそのメトリックが表示されます。このビューは、ハードウェアカウンタオーバーフロープロファイリングの拡張であるデータ領域プロファイリングを含む実験にのみ適用されます。詳細は、193 ページの「データ領域プロファイリングとメモリー領域プロファイリング」を参照してください。

このビューを表示するには、「設定」ダイアログボックス (132 ページの「「ビュー」設定」を参照) の「ビュー」タブでこのビューを選択します。「データオブジェクト」ビューを表示可能にできるのは、ロードされた実験の 1 つ以上にデータ領域プロファイルが含まれている場合だけです。

このビューには、プログラム内のさまざまなデータ構造や変数に対するハードウェアカウンタメモリー操作のメトリックが表示されます。

単一のデータオブジェクトを選択するには、そのオブジェクトをクリックします。

ビュー内に隣接して表示された複数のオブジェクトを選択するには、最初のオブジェクトを選択したあと、Shift キーを押しながら最後のオブジェクトをクリックします。

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

「I/O」ビュー

「I/O」ビューを使用して、アプリケーション内の I/O パターンを識別し、アプリケーションのパフォーマンスに影響する I/O のボトルネックを特定します。「I/O」ビューは、アプリケーションの I/O トレースデータをプロファイルした場合に使用できます。

I/O データは、次のいずれかのオプションに従って集約できます。

ファイル名
: プログラムがアクセスするファイルの表を表示します。各行が、1 つのファイルを表します。1 つの行のメトリックは、ファイルのすべてのアクセスに関して集計された I/O 統計情報を表します。

ファイル記述子
: プログラムがアクセスするファイルのファイル記述子の表を表示します。各行が、開かれているファイルの単一インスタンスを表します。同じファイルが複数回開かれている場合は、表に同じファイルの行が複数含まれています。1 つの行のメトリックは、開いているファイルの単一インスタンスに適用されます。

呼び出しスタック
: 任意のスタック番号とともに一覧表示された呼び出しスタックの表を表示します。スタックをクリックすると、「呼び出しスタック」パネルにそのスタック内の関数呼び出しが表示されます。メトリックは選択した呼び出しスタックに適用されます。

「ヒープ」ビュー

「ヒープ」ビューには、メモリーリークの可能性を示すメモリー割り当てメトリックを含む呼び出しスタックのリストが表示されます。呼び出しスタックは、任意のスタック番号で識別されます。ピークメモリー使用量を示すために 1 つの呼び出しスタックにラベルが付けられます。

呼び出しスタックをクリックして、メトリックの詳細を「選択の詳細」パネルに表示し、呼び出しスタックの関数呼び出しを「呼び出しスタック」パネルに表示します。「呼び出しスタック」パネルで関数をダブルクリックして、ソースを表示できます。選択した呼び出しスタックをフィルタで除外するか、選択していない呼び出しスタックをフィルタで除外するように、フィルタを設定することもできます。ロードされた実験の 1 つ以上にヒープトレースデータが含まれる場合にのみ、「設定」ダイアログボックスの「ビュー」タブで「ヒープ」ビューを選択できます。「ヒープ」ビューの下部のパネルには、完全なターゲットアプリケーションを表す `<Total>` 擬似関数の詳細データが表示されます。

「データサイズ」ビュー

「データサイズ」ビューは、バイト数などのサイズ要素を持つデータが含まれている実験について使用できます。ヒープトレース、I/O トレース、または MPI トレースを含む実験には「データサイズ」ビューがあります。

「データサイズ」ビューでは、データをいくつかのデータサイズ範囲に整理し、ある範囲に分類されるデータを持つイベントのメトリックを計算します。サイズ要素を持たないデータは、データサイズ 0 に帰属します。

「データサイズ」ビューを使用して、データをフィルタ処理できます。たとえば、ヒープトレースデータを含む実験で、「リークバイト数」について高いメトリックを持つサイズ範囲の行を選択し、「選択した項目を含むイベントのみを含める」というフィルタを追加することができます。別のデータビューに移動すると、選択したサイズ範囲のメモリーリークが発生したイベントのみが表示されるようにデータがフィルタ処理されます。

「経過時間」ビュー

「経過時間」ビューは、期間を持つデータが含まれている実験について使用できます。I/O トレースデータ、MPI トレース、ヒープトレース、および同期トレースデータを含む実験には「経過時間」ビューがあります。

「経過時間」ビューでは、データをいくつかの経過時間範囲に整理し、それぞれの経過時間範囲に分類されるデータを持つイベントのメトリックを計算します。I/O トレース、MPI トレース、および同期トレースについては関数呼び出しの経過時間が記録されます。ヒープトレースの場合、経過時間はメモリーの割り当てから解放までの時間です。経過時間要素を持たないデータは、経過時間 0 に帰属します。

「経過時間」ビューを使用して、データをフィルタ処理できます。たとえば、ヒープトレースデータを含む実験で、「割り当てバイト数」について高いメトリックを持つ経過時間範囲の行を選択し、「選択した項目を含むイベントのみを含める」というフィルタを追加することができます。別のデータビューに移動すると、選択した範囲に一致する経過時間を持つイベントのみが表示されるようにデータがフィルタ処理されるため、経過時間が予想よりも長いメモリー割り当てが見つかることがあります。

「OpenMP 並列領域」ビュー

「OpenMP 並列領域」ビューは、Oracle Developer Studio コンパイラでコンパイルされた OpenMP タスクを使用するプログラムに関して、OpenMP 3.0 コレクタで記録された実験にのみ適用されます。詳細は、57 ページの「OpenMP プロファイルに関する制限事項」を参照してください。

このビューには、プログラムの実行中に発生したすべての並列領域が、同じプロファイリングデータから計算されたメトリック値とともに一覧表示されます。現在の並列領域に対しては、排他的メトリックスが計算されます。包括的メトリックスには、入れ子並列性が反映されます。これらは現在の並列領域と、その作成元である親の並列領域に起因します。この起因関係は、(すべての並列領域の外部での) プログラムの直列実行を表す最上位の暗黙的な OpenMP 並列領域まで再帰的に続きます。プログラムに入れ子の並列領域がない場合、排他的メトリックスと包括的メトリックスは同じ値を持ちます。

並列領域を含む関数が何回も呼び出された場合は、並列領域のすべてのインスタンスが集計され、対応するビュー内に 1 項目として表示されます。

このビューは、ナビゲーションに役立ちます。たとえば、OpenMP 待ち時間がもっとも大きい並列領域といった関心のある項目を選択し、そのソースを分析したり、選択項目に関連するデータのみを含むようにコンテキストフィルタを選択したりできます。そのあと、「関数」、「タイムライン」、「スレッド」などのほかのビューを使用して、これらのデータがほかのプログラムオブジェクトによってどのように表されるかを解析できます。

「OpenMP タスク」ビュー

「OpenMP タスク」ビューには、OpenMP タスクとそれぞれのメトリックの一覧が表示されます。このビュー内のオプションは、Oracle Developer Studio コンパイラでコンパイルされた OpenMP タスクを使用するプログラムについて、OpenMP 3.0 コレクタで記録された実験にのみ適用されます。詳細は、57 ページの「OpenMP プロファイルに関する制限事項」を参照してください。

このビューには、プログラムの実行中に発生したタスクが、プロファイリングデータから計算されたメトリック値とともに一覧表示されます。排他的メトリックスは、現在のタスクのみに適用されます。包括的メトリックスは、OpenMP タスクのメトリックス、および、タスク生成時に構築され

た親子関係を持つ、子タスクのメトリックスを含みます。暗黙並列領域の OpenMP タスクは、プログラムの直列実行を表します。

タスクを含む関数が何回も呼び出された場合は、並列領域のすべてのインスタンスが集計され、対応するビュー内に 1 項目として表示されます。

このビューは、ナビゲーションに役立ちます。OpenMP 待ち時間が最大のタスクなどの関心のある項目を選択し、「ソース」ビューをクリックすることによってそのソースを解析できます。また、右クリックしてコンテキストフィルタを選択し、選択された項目に関連したデータのみを含めることもできます。そのあと、「関数」、「タイムライン」、「スレッド」などのほかのビューを使用して、これらのデータがほかのプログラムオブジェクトによってどのように表されるかを解析できます。

「行」ビュー

「行」ビューには、ソース行とそのメトリックのリストが表示されます。

ソース行は、そのラベルとして、呼び出し元の関数、行番号、およびソースファイル名で示されます。関数の行番号情報が得られない場合、または関数のソースファイルが不明の場合は、その関数のすべてのプログラムカウンタ (PC) が、行の表示でその関数の 1 つのエントリに集計されて表示されます。関数が表示されないロードオブジェクトにある関数の PC は、ロードオブジェクト別に「行」表示で 1 つのエントリにまとめて表示されます。「行」ビューで行を選択すると、特定の行のすべてのメトリックが「選択の詳細」ウィンドウに表示されます。「行」ビューから行を選択したあとに「ソース」または「逆アセンブリ」ビューを選択すると、表示が適切な行に移動します。

「PC」ビュー

「PC」ビューには、プログラムカウンタ (PC) とそのメトリックが一覧表示されます。PC には、呼び出し元の関数、およびその関数内のオフセットがラベルとして表示されます。関数が表示されないロードオブジェクトにある関数の PC は、ロードオブジェクト別に「PC」表示に 1 つのエントリにまとめて表示されます。「PC」ビューで行を選択すると、その PC のすべてのメトリックが「サマリー」タブに表示されます。「PC」ビューから行を選択したあとに「ソース」ビューまたは「逆アセンブリ」ビューを選択すると、表示が適切な行に移動します。

PC の詳細については、195 ページの「呼び出しスタックとプログラムの実行」のセクションを参照してください。

「逆アセンブリ」ビュー

「逆アセンブリ」ビューには、選択した関数を含むオブジェクトファイルの逆アセンブリリストが、各命令のパフォーマンスメトリックの注釈付きで表示されます。逆アセンブリリストを表示するには、ツールバーの「表示モード」リストから「マシン」を選択することが必要になる場合があります。

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

「ソース」ビューと同様に、「逆アセンブリ」ビューにはインデックス行が表示されます。ただし、「ソース」ビューとは異なり、代替ソースコンテキストのインデックス行をナビゲーションの目的に直接使用することはできません。また、代替ソースコンテキストのインデックス行は、「逆アセンブリ」表示の先頭に単に一覧表示されるのではなく、`#include` またはインライン化されたコードの挿入位置の先頭に表示されます。

`#include` されるか、またはほかのファイルからインライン化されたコードは、ソースコードをインタリーブせずに raw の逆アセンブリ命令として表示されます。ただし、これらの命令の 1 つにカーソルを置いて「ソース」ビューを選択すると、`#include` またはインライン化されたコードを含むファイルが開きます。このファイルを表示した状態で「逆アセンブリ」ビューを選択すると、新しいコンテキストで「逆アセンブリ」ビューが開き、インターリーブされたソースコードとともに逆アセンブリコードが表示されます。

表示される解説のクラスは、「設定」ダイアログボックスで設定できます。デフォルトのクラスは、ダイアログボックスの「保存」ボタンをクリックすることによって `.er.rc` デフォルト値ファイルで設定できます。

パフォーマンスアナライザは、重要な行の検索を容易にするために、メトリック固有のしきい値と等しいか、またはそれを超えるメトリックを含む行であるホットラインを強調表示します。このしきい値は、「設定」ダイアログボックスで設定できます。

「ソース」ビューと同様に、黄色のナビゲーションマーカーが、メトリックス付きの各ソース行の右側のスクロールバーの隣のマージン内に表示されます。ホットしきい値を下回るゼロ以外のメトリックは強調表示されませんが、黄色のナビゲーションマーカーが表示されます。メトリックス付きのソース行にすばやく移動するには、右側マージン内の黄色のマーカーをクリックすると、メトリックス付きの行にジャンプできます。また、メトリックス自体を右クリックし、「次のホットライン」や「次のゼロ以外のメトリックライン」などのオプションを選択して、メトリックス付きの次の行にジャンプすることができます。

「逆アセンブリ」ビューの内容の詳細は、229 ページの「注釈付き逆アセンブリコード」を参照してください。

「ソース/逆アセンブリ」ビュー

「ソース/逆アセンブリ」ビューでは、上のペインに注釈付きソースが、下のペインに注釈付き逆アセンブリが表示されます。これらのペインは、あるペインにある行を選択すると、ほかのペインにある関連する行も選択されるように調整されます。このビューは、デフォルトでは表示されません。

「競合」ビュー

「競合」ビューには、データ競合実験で検出されたすべてのデータ競合のリストが表示されます。データ競合をクリックすると、そのデータ競合に関する詳細を右側のパネルにある「競合の詳細」ウィンドウに表示できます。詳細は、『Oracle Developer Studio 12.5: スレッドアナライザユーザーズガイド』を参照してください。

「デッドロック」ビュー

「デッドロック」ビューには、デッドロック実験で検出されたすべてのデッドロックのリストが表示されます。デッドロックをクリックすると、そのデッドロックに関する詳細が右側のパネルにある「デッドロックの詳細」ウィンドウに表示されます。詳細は、F1 キーを押してヘルプを表示するか、または『Oracle Developer Studio 12.5: スレッドアナライザユーザーズガイド』を参照してください。

「デュアルソース」ビュー

「デュアルソース」ビューには、選択したデータ競合またはデッドロックに関連する 2 つのソースコンテキストが表示されます。このビューは、データ競合検出またはデッドロック実験がロードされている場合にのみ表示されます。詳細は、『Oracle Developer Studio 12.5: スレッドアナライザユーザーズガイド』を参照してください。

「統計」ビュー

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

「実験」ビュー

「実験」ビューは 2 つのパネルに分かれています。上のパネルには、読み込まれたすべての実験に含まれるロードオブジェクトのノードと、読み込まれた実験ごとのノードを含むツリーが含まれています。「ロードオブジェクト」ノードを展開すると、すべてのロードオブジェクトのリストが、それらの処理に関するさまざまなメッセージと一緒に表示されます。実験のノードを展開すると、「注記」と「情報」という 2 つの領域が表示されます。

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

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

下のパネルには、パフォーマンスアナライザセッションからのエラーおよび警告メッセージが一覧表示されます。

「命令頻度」ビュー

「命令頻度」ビューには、カウントデータ実験で各種命令が実行された (`collect -c` で収集された) 頻度のサマリーが表示されます。また、ロード、ストア、浮動小数点の各命令の実行頻度に関するデータも示します。無効になった命令や、分岐遅延スロット内の命令に関する情報も表示されます。

「MPI タイムライン」ビュー

「MPI タイムライン」ビューには、一連の水平バーが MPI 実験内のプロセスごとに 1 つずつ表示され、それらをつなぐ対角線がメッセージを示します。各バーには、MPI 機能によって色分けされた領域や、プロセスが MPI に含まれていない (アプリケーションコード内の別の場所にある) ことを示す領域があります。

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

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

MPI タイムラインの画像を `.jpg` ファイルに出力できます。「ファイル」→「エクスポート」を選択し、「エクスポート形式: JPEG」を選択します。

MPI タイムラインコントロール

「MPI タイムラインコントロール」ウィンドウは、「MPI タイムライン」ビューのズーム、パン、イベントステップ、およびフィルタリングをサポートします。このウィンドウには、「MPI タイムライン」に表示される MPI メッセージのパーセンテージを調整するコントロールがあります。

フィルタリングを行うと、ビューの現在のフィールドの外部にあるデータが、「MPI タイムライン」ビューおよび「MPI チャート」ビューに表示されるデータセットから削除されます。フィルタを適用するには、「フィルタ」ボタンをクリックします。最後に適用したフィルタを取り消すにはフィルタの戻るボタンを使用し、フィルタを再適用するにはフィルタの進むボタンを使用します。フィルタは、「MPI タイムライン」ビューと「MPI チャート」ビューの間で共有されますが、その他のデータビューには適用されません。

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

「MPI タイムラインコントロール」ウィンドウには、「MPI タイムライン」ビューでの機能やメッセージの選択に関する詳細を表示することもできます。

「MPI チャート」ビュー

「MPI チャート」ビューには、「MPI タイムライン」ビューに表示される MPI トレースデータのチャートが表示されます。このタブには、MPI の実行に関するデータのプロットが表示されます。「MPI チャート」ビューのコントロールを変更して「変更を表示更新する」をクリックすると、新しいチャートが表示されます。チャートからいずれかの要素を選択すると、その要素に関する詳細情報が「MPI チャートコントロール」ビューに表示されます。

マウスをドラッグすると、ドラッグによって定義された矩形領域上で、「MPI チャート」ビューがズームインします。

MPI チャートの画像は .jpg ファイルに出力できます。「ファイル」→「エクスポート」を選択し、「エクスポート形式: JPEG」を選択します。

MPI チャートコントロール

「MPI チャートコントロール」ウィンドウには、チャートの種類、X 軸と Y 軸のパラメータ、およびデータの集計に使用されるメトリックや演算子を制御するための一連のドロップダウンリストが含まれています。「変更を表示更新する」をクリックすると、新しいグラフが描画されます。

フィルタリングを行うと、ビューの現在のフィールドの外部にあるデータが、「MPI タイムライン」ビューおよび「MPI チャート」ビューに表示されるデータセットから削除されます。フィルタを適用するには、「フィルタ」ボタンをクリックします。最後に適用したフィルタを取り消すにはフィルタの戻るボタンをクリックし、フィルタを再適用するにはフィルタの進むボタンをクリックします。

また、「MPI チャートコントロール」ウィンドウを使用して、「MPI チャート」ビューからの選択に関する詳細を表示することもできます。

ライブラリとクラスの可視性の設定

デフォルトでは、パフォーマンスアナライザには、ターゲットプログラムとそのプログラムが使用するすべての共有ライブラリおよびクラスの関数データが表示されます。「ライブラリとクラスの可視性」ダイアログボックスを使用して、任意のライブラリまたはクラスの関数データを非表示にできます。

このダイアログボックスを開くには、「ツール」>「ライブラリの可視性」を選択するか、「ライブラリの可視性」ツールバーボタンをクリックします。

「ライブラリとクラスの可視性」ダイアログボックスには、実験のすべての共有ライブラリおよびクラスが一覧表示されます。一覧表示される項目ごとに、次のいずれかの可視性レベルを選択できます。

関数	ライブラリまたはクラスのすべての関数が表示されます。該当する場合、各関数のメトリックが表示されます。
API	ライブラリまたはクラスへの呼び出しを表す関数だけが表示されます。これらの関数下の呼び出しは、そのライブラリ内の呼び出しであろうとほかのライブラリへの呼び出し (コールバックを含む) であろうと表示されません。
ライブラリ	ライブラリまたはクラスの名前のみが表示され、すべての内部関数は非表示になります。ライブラリのメトリックは、内部関数で発生するメトリックのアグリゲーションを反映します。リスト内のすべてのエントリに対して「ライブラリ」チェックボックスを選択すると、ライブラリまたはクラスごとに分析を実行できます。

「フィルタ」フィールドでは、ライブラリおよびクラスのサブセットの可視性設定を更新できます。多数のライブラリをロードするプログラムの場合、リストは非常に大きくなる可能性があります。これらのフィルタはこのダイアログでのみ機能し、データビューのフィルタには関連しません。

ライブラリの可視性を設定しても、非表示の関数に対応するメトリックは引き続き、すべての表示に何らかの形式で表されます。これは、表示からデータを削除するデータビューのフィルタとは異なります。このダイアログで実行できるタスクの詳細については、パフォーマンスアナライザのヘルプを参照してください。

データのフィルタリング

実験を開くと、プログラムのすべてのプロファイリングデータが表示されます。フィルタリングでは、プログラムの特定の領域または特性に焦点を当てることができるように、ビューから対象外のデータを一時的に削除できます。

あるビューで適用したフィルタは、すべてのビューに影響します。たとえば、「関数」などのほかのビューには、フィルタ処理された期間に関連したメトリックだけが表示されるように、「タイムライ

ン」ビューで期間を指定できます。ビューで 1 つ以上の項目を選択し、続いて、フィルタを 1 つずつ選択して、ビューに含めるデータを指定します。

複数の方法でフィルタ処理できます。

- 「フィルタ」ボタンをクリックして、現在のデータビューで選択されている項目に適用できるフィルタのリストを開きます。
- データビュー内の項目を右クリックするか、または項目を選択したまま Shift-F10 キーを押してから、適用するフィルタを選択します。
- パフォーマンスアナライザの左下隅にある「フィルタ」パネルを使用して、適用したフィルタを確認し、フィルタを追加または削除します。

フィルタを結合して、プログラム実行の特に注意が必要な領域のメトリックを表示できます。たとえば、「関数」ビューのフィルタと「タイムライン」ビューのフィルタを適用すると、プログラムの実行における一定の期間に特定の関数を含む呼び出しスタックに焦点を当てられます。

フィルタを使用するとき、データはパフォーマンスアナライザのすべてのビューでフィルタ処理されますが、ほかのデータビューとやり取りしない別個のフィルタリングメカニズムを持つ MPI タイムラインは除きます。

パフォーマンスアナライザの経験豊富なユーザーは、「拡張カスタムフィルタ」ダイアログボックスを使用して、フィルタ式を編集してカスタムフィルタを作成し、表示するデータを正確に定義することもできます。

注記 - ここで説明されているフィルタは、122 ページの「MPI タイムラインコントロール」および 123 ページの「MPI チャートコントロール」で説明されている MPI フィルタリングとは関係ありません。これらのフィルタは、「MPI タイムライン」ビューや「MPI チャート」ビューには影響しません。

フィルタの使用

フィルタは、パフォーマンスアナライザ内のほとんどのデータビューで使用できます。フィルタにアクセスするには、ツールバーまたは「アクティブなフィルタ」パネルにある「フィルタ」ボタンを使用します。あるいは、マウスで右クリックするか、キーボードの Shift-F10 キーを押します。フィルタを追加すると、データがただちにフィルタされます。

フィルタは一般に、焦点を当てるビューで 1 つ以上の項目を選択し、適切なフィルタを選択することによって使用します。ほとんどのビューでは、フィルタを使用すると、そのフィルタで指定された条件を満たすデータを含めるようにすることも、含めな*い*ようにすることもできます。これにより、フィルタを使用して、プログラムの特定の領域のデータにフォーカスしたり、特定の領域のデータを除外したりすることができます。

次のいくつかの方法でフィルタを使用できます。

- 複数のフィルタを追加してデータを絞り込みます。フィルタは論理的な AND 関係で結合されるため、データがすべてのフィルタに一致することが必要です。

- あるビューでフィルタを追加したあと、フィルタされたデータを別のビューで検査します。たとえば、「呼び出しツリー」ビューでもっともホットな分岐を検索できたら、「フィルタを追加: 選択した分岐を含むスタックのみを含める」を選択し、「関数」ビューに移動してそのコード分岐で呼び出された関数のメトリックを表示します。
- 複数のビューから複数のフィルタを追加して、非常に具体的な一連のデータを作成します。
- 拡張カスタムフィルタを作成するための基礎としてフィルタを使用します。126 ページの「拡張カスタムフィルタの使用」を参照してください。

拡張カスタムフィルタの使用

パフォーマンスアナライザのデータビューでフィルタを追加すると、フィルタ式が生成され、データのフィルタ処理のためにただちに適用されます。生成されたフィルタ式は、「拡張カスタムフィルタ」ダイアログボックスに表示されます。経験豊富なユーザーは、カスタマイズされたフィルタを作成する開始点としてこれらの生成されたフィルタ式を使用できます。

カスタムフィルタを作成するには、次のようにします。

1. 次のいずれかを行なって「拡張カスタムフィルタ」ダイアログボックスを開きます。
   - 「フィルタ」ボタンをクリックし、「フィルタの追加: 拡張カスタムフィルタ」を選択します。
   - 「ツール」⇒「フィルタ」⇒「フィルタの追加: 拡張カスタムフィルタ」を選択します。
2. 「フィルタ指定」テキストボックスをクリックしてフィルタを編集します。フィルタの詳細は、次を参照してください。
3. 必要に応じて矢印ボタンを使用して、編集を元に戻したり、やり直したりします。
4. 「了解」をクリックして、フィルタ式に応じてデータをフィルタ処理し、ダイアログボックスを閉じます。

「フィルタ指定」パネルには、以前にパフォーマンスアナライザのデータビューで選択し適用したフィルタのフィルタ式が表示されます。これらのフィルタは編集できます。最上部にある矢印ボタンを使用すると、編集を元に戻したり、やり直したりできます。また、テキストエディタの場合と同様に、Ctrl-Z キーを使用して元に戻したり、Shift-Ctrl-Z キーを押してやり直したりすることもできます。「了解」をクリックすると、フィルタはデータビューにのみ影響します。

新しい各フィルタを && (論理「AND」演算子) で始まる新しい行に配置します。実験データが表示されるには、最初のフィルタと 2 番目のフィルタと 3 番目のフィルタのように、すべてのフィルタに一致する必要があります。

データを、たとえば最初のフィルタまたは 2 番目のフィルタに一致させるようにする場合は、&& を || に変更できます。

フィルタ式では、標準的な C の関係演算子 (==、>=、&&、|| など) を実験に固有のキーワードとともに使用します。実験で使用できるキーワードは、「拡張カスタムフィルタ」ダイアログボックスの「キーワード」パネルに表示されます。

フィルタのキーワードおよびフィルタ式の詳細は、パフォーマンスアナライザのヘルプを検索してください。

フィルタ式の構文は、er_print でのフィルタリングに使用される構文と同じです。フィルタ式については、177 ページの「式の文法」を参照してください。

フィルタリングのためのラベルの使用

ラベルは、実験の一部に割り当てることのできる名前です。er_label コマンドを使用すると、実験内のある期間にラベル名を割り当てることができ、そのラベルはその実験とともに維持されます。ラベルを使用すると、er_print コマンドまたはパフォーマンスアナライザで実験データをフィルタして、ラベル付けされた期間中に収集されたデータを含めたり、除外したりできます。

244 ページの「実験へのラベル付け」 ユーティリティーを使用してラベルを作成する方法については、Labeling Experimentsを参照してください。

パフォーマンスアナライザでは、「拡張カスタムフィルタ」ダイアログボックスで、ラベル付けされた期間のデータをフィルタできます。「フィルタ指定」パネルでラベル名を入力し、「適用」をクリックして、そのラベルで指定されたデータをフィルタリングします。このラベルが、TSTAMP キーワードで数値比較を使用するフィルタ式のニックネームとして機能するため、数値比較を使用する必要はありません。ラベルを && で始まる個別の行に追加することによって、そのラベルを「フィルタ指定」パネル内のほかのフィルタと組み合わせることができます。

パフォーマンスアナライザで開かれている実験にラベルが割り当てられているかどうかは、「拡張カスタムフィルタ」ダイアログボックスの「キーワード」パネルで確認できます。また、er_print -describe コマンドを使用しても同じ情報を表示できます。ラベルはこの表示の最初に一覧表示され、そのラベルによって実装された TSTAMP キーワードを含む実際のフィルタ式が含まれています。

ラベルフィルタを適用したあと、「タイムライン」ビューをクリックすると、そのラベルで定義された間隔内のデータが削除されていることを確認できます。また、このデータは、フィルタリングをサポートするほかのデータビューでもフィルタされます。

パフォーマンスアナライザからのアプリケーションのプロファイリング

端末ウィンドウから collect コマンドを使用するか、またはパフォーマンスアナライザを使用してアプリケーションをプロファイルできます。

パフォーマンスアナライザでアプリケーションをプロファイルするには、次のいずれかを実行します。

- 「ようこそ」画面で「アプリケーションのプロファイル」をクリックします。
- 「アプリケーションのプロファイル」ツールバーボタンをクリックします。
- 「ファイル」->「アプリケーションのプロファイル」を選択します。

- `analyzer` コマンドを実行し、コマンド行でターゲットプログラムとその引数を指定します。

それぞれの方法で、「アプリケーションのプロファイル」ダイアログボックスを開きます。このダイアログに関するヘルプ情報を表示するには、F1 キーを押します。

「アプリケーションのプロファイル」ダイアログ内のオプションは、第3章「パフォーマンスデータの収集」で説明されている `collect` コマンドで使用できるオプションに対応しています。

デフォルトのプロファイリングオプションを使用し、クロックプロファイリングデータしか収集しない場合は、ターゲットプログラムが唯一必要なエントリです。それ以外の場合は、「一般」タブで実験オプションを指定したり、「収集するデータ」タブで収集するデータの種類を選択したりできます。

「プレビューコマンド」ボタンをクリックすると、「実行」ボタンをクリックしたときに使用される `collect` コマンドを表示できます。そのあと「実行」をクリックするだけで、プロファイリング、データの収集、および実験の作成が開始されます。

ターゲットプログラムの出力は、デフォルトで、パフォーマンスアナライザによって開かれる別のターミナルウィンドウに表示されます。実験の進行中に「アプリケーションのプロファイル」ダイアログを閉じても、実験は続行されます。ダイアログを再度開くと、実行中にパネルが開いたままであったかのように、実行中の実験が表示されます。実験の実行中にパフォーマンスアナライザを終了しようとすると、実行を終了するか継続するかの確認がダイアログボックスに表示されます。

実験を停止するには、「終了」をクリックします。実験を停止するかどうかを確認する必要があります。

また、次のセクションの説明に従って実行中プロセスをプロファイルしたり、第9章「カーネルプロファイリング」の説明に従ってカーネルをプロファイルすることもできます。

実行中のプロセスのプロファイリング

Oracle Solaris では、パフォーマンスアナライザの実行中のプロセスから、またはコマンド行からデータを収集できます。Linux では、実行中のプロセスのプロファイリングは、シングルスレッドアプリケーションでのみ確実に機能します。JVM はマルチスレッド化されているため、Linux で Java アプリケーションをプロファイルすることはできません。

コマンド行から実行中のプロセスをプロファイルする場合は、`collect(1)` のマニュアルページを参照してください。

パフォーマンスアナライザで実行中プロセスをプロファイルするには、次のいずれかを実行します。

- 「ようこそ」画面で「実行中プロセスのプロファイル」をクリックします。
- 「ファイル」->「実行中プロセスのプロファイル」を選択します。

デフォルトのオプションを使用する場合は、「実行中プロセスのプロファイル」ダイアログボックスでプロファイルするプロセスを選択し、「実行」をクリックします。それ以外の場合は、「一般」タブで実験オプションを指定したり、「収集するデータ」タブで収集するデータの種類を選択したりできます。

このダイアログのヘルプを表示するには、ダイアログを開いている間に F1 キーを押します。

「出力」タブには、コレクタからの出力とプロセスからの出力が表示されます。プロファイリングを停止させるには、「終了」をクリックします。パフォーマンスアナライザから、実験を開くように求められます。

実験の比較

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

パフォーマンスアナライザで 2 つの実験を比較するには、ツールバーの「実験を比較」アイコンをクリックし、比較する 2 つの実験を選択します。

ほとんどのデータビューで実験の比較がサポートされています。比較モードでは、実験または実験グループのデータは隣接する列に表示されます。列は、実験または実験グループの読み込み順に表示され、追加のヘッダー行に実験名または実験グループ名が示されます。

比較形式の設定

「ステータスの比較」パネルの形式ボタンを使用します。比較形式として、次のいずれかを指定できます。

絶対値　　ロードされたすべての実験のメトリック値を表示します。

デルタ　　ベースライン実験とロードされたほかの実験のメトリック間の +/- の差を表示します。

比率　　ベースライン実験とロードされたほかの実験のメトリック間の差を比率として表示します。たとえば、比較実験メトリックは、`x0.994` のように表示され、ベース実験と相対的な値を示します。

「ステータスの比較」パネルに関するヒントと詳細情報については、パフォーマンスアナライザのヘルプを参照してください。

パフォーマンスアナライザのリモートでの使用

Oracle Developer Studio ツールがインストールされているサーバー間で、または Oracle Developer Studio をインストールできないデスクトップクライアント上でも、パフォーマンスアナライザをリモートで使用できます。

デスクトップクライアント上でのパフォーマンスアナライザの使用

デスクトップクライアントシステムに特殊なバージョンのパフォーマンスアナライザをインストールすると、ツールがインストールされているリモートサーバーに接続できます。

クライアントシステムの要件は次のとおりです。

- オペレーティングシステムは、Mac OS X、Windows、Linux、または Oracle Solaris である必要があります。
- パフォーマンスアナライザのバージョンは、リモートシステムにインストールされた Oracle Developer Studio ツールのバージョンと一致する必要があります。
- Java 1.7 または 1.8 がユーザーのパスに存在する必要があります。

リモートシステムの要件は次のとおりです。

- オペレーティングシステムは、Oracle Solaris または Linux である必要があります。
- Secure Shell (SSH) デーモン sshd が実行されている必要があります。
- Oracle Developer Studio ソフトウェアは、リモートホストでアクセスできる必要があり、ソフトウェアまでのパスがわかっている必要があります。
- ホストにユーザーアカウントが必要です。

クライアントシステム上で使用するパフォーマンスアナライザのバージョンは、Mac OS X または Windows マシンを含む任意のシステムにコピーできる tar ファイルで入手できます。このバージョンはリモートパフォーマンスアナライザクライアントとして知られ、Java 1.7 または Java 1.8 を使用できるどのシステムでもパフォーマンスアナライザを使用できるようにします。

リモートパフォーマンスアナライザクライアントは、Oracle Developer Studio インストールの */install-dir*`/lib/analyzer/RemoteAnalyzer.tar` にあります。

デスクトップクライアント上でパフォーマンスアナライザを使用するには、次のタスクの説明に従って、そのクライアントに特殊なリモートバージョンをインストールしておく必要があります。

▼ クライアント上でリモートパフォーマンスアナライザを使用する方法

1. **`RemoteAnalyzer.tar` ファイルを、それを実行するデスクトップクライアントシステムにコピーします。**
2. **ファイル抽出ユーティリティーまたはコマンドを使用して、`RemoteAnalyzer.tar` を抽出します。**

Windows では、ファイルを抽出するために、WinZip や 7-Zip などのアプリケーションのインストールが必要になることがあります。

抽出された RemoteAnalyzer ディレクトリには、Windows、Mac OS X、Linux、および Solaris 用のスクリプトと README.txt ファイルが含まれています。

3. **使用しているシステム用のスクリプトを実行します。**

Windows	RemoteAnalyzer ディレクトリ内の AnalyzerWindows.bat ファイルをダブルクリックします。
Mac	RemoteAnalyzer ディレクトリ内の AnalyzerMacOS.command ファイルを実行します。
Linux	端末ウィンドウで AnalyzerLinux.sh スクリプトを実行します。
Oracle Solaris	端末ウィンドウで AnalyzerSolaris.sh スクリプトを実行します。

「パフォーマンスアナライザ」ウィンドウが開き、リモートで動作する機能だけが有効になっている「ようこそ」画面が表示されます。詳細は、RemoteAnalyzer/README.txt ファイルを参照してください。

パフォーマンスアナライザでのリモートホストへの接続

次の方法でリモートホストに接続できます。

- 「ようこそ」ビューで、「リモートホストに接続」をクリックします。
- 「ファイル」 > 「リモートホストに接続」を選択します。
- 「パフォーマンスアナライザ」ウィンドウの下部にある「リモートホスト」ステータスメッセージをクリックします。

プロンプトに応答する方法については、「リモートホストに接続」ダイアログボックスで F1 キーを押します。

接続が完了すると、パフォーマンスアナライザのメインウィンドウ下部のステータス領域に、ホストに接続したことが示されます。リモートホストに接続すると、パフォーマンスアナライザのファイルブラウザで実験を開くときに、リモートホストのファイルシステムに自動的にアクセスします。

構成設定

データの表示方法やその他の構成設定は、「設定」ダイアログボックスを使用して制御できます。このダイアログボックスを開くには、ツールバーの「設定」ボタンをクリックするか、または「ツール」->「設定」を選択します。

「設定」ダイアログボックスは、次のカテゴリに整理されています。

- 132 ページの「「ビュー」設定」
- 133 ページの「メトリック設定」
- 134 ページの「タイムライン設定」
- 135 ページの「ソース/逆アセンブリ設定」
- 136 ページの「呼び出しツリー設定」
- 136 ページの「書式設定」
- 137 ページの「検索パス設定」
- 138 ページの「パスマップ設定」

「了解」ボタンをクリックすると、現在のセッションについて行なった変更が適用され、ダイアログボックスが閉じます。「適用」ボタンをクリックすると、現在のセッションに対して、変更が適用されますが、ダイアログボックスは開いたままなので、変更を続行できます。「閉じる」ボタンは、変更を保存または適用せずにダイアログボックスを閉じます。

「エクスポート」ボタンは「エクスポート設定」ダイアログを開き、それを使用して、エクスポートする設定とそれらを保存する場所を選択できます。エクスポートされた構成設定は、現在のセッションだけでなく、将来のパフォーマンスアナライザセッションでこの実験をふたたび開いたときにも適用できます。構成はほかの実験にも使用できます。詳細は、138 ページの「パフォーマンスアナライザの構成ファイル」を参照してください。

「ビュー」設定

「ビュー」設定パネルには、現在の実験に適用できるデータビューが一覧表示されます。

標準ビュー

チェックボックスをクリックして、パフォーマンスアナライザに表示する標準データビューを選択または選択解除します。

インデックスオブジェクトビュー

チェックボックスをクリックして、表示する「インデックスオブジェクト」ビューを選択または選択解除します。事前に定義されている「インデックスオブジェクト」ビューには、「スレッド」、「CPU」、「標本」、「秒」、「プロセス」があります。

カスタムインデックスオブジェクト用のビューを追加するには、「カスタムビューを追加」ボタンをクリックして「インデックスオブジェクトを追加」ダイアログを開きます。指定するインデックスオブジェクト名は定義されていないものであり、既存のどのコマンド、メモリーオブジェクトタイプにも一致していないものにする必要があります。この名前は大文字と小文字が区別されず、英数字のみ、または「_」文字を含め、1 文字目は英字で構成する必要があります。式は、177 ページの「式の文法」で説明されている構文に従う必要があります。

インデックスオブジェクトは、er_print コマンドを使用して作成することもできます。159 ページの「インデックスオブジェクトリストを制御するコマンド」を参照してください。

メモリーオブジェクトビュー

チェックボックスをクリックして、表示する事前定義されたメモリーオブジェクトビューを選択または選択解除します。これらのビューは、実験にハードウェアカウンタプロファイリングデータが含まれる場合に使用できます。

メモリーオブジェクトは、キャッシュ行、ページ、メモリーバンクなどの、メモリーサブシステム内のコンポーネントを表します。メモリーオブジェクトは、仮想ページおよび物理ページについて 8K バイト、64K バイト、512K バイト、および 4M バイトのサイズで事前定義されています。

カスタムメモリーオブジェクト用のビューを追加するには、「「カスタムメモリーオブジェクト」ビューを追加」ボタンをクリックして「メモリーオブジェクトを追加」ダイアログボックスを開きます。指定するメモリーオブジェクト名は定義されていないものであり、既存のどのコマンド、またはどのインデックスオブジェクトタイプにも一致していないものにする必要があります。この名前は大文字と小文字が区別されず、英数字のみ、または「_」文字を含め、1 文字目は英字で構成する必要があります。式は、177 ページの「式の文法」で説明されている構文に従う必要があります。

マシンモデルのファイル

特定の SPARC システムアーキテクチャーのメモリーオブジェクトを定義するファイルをロードできます。「マシンモデル」ドロップダウンメニューから、対象のシステムアーキテクチャーを選択します。「適用」または「了解」をクリックすると、オブジェクトの新しいリストが「メモリーオブジェクトビュー」列に表示されます。これらのビューから選択して、関連するデータを表示できます。詳細は、ヘルプで「マシンモデル」を検索してください。

デフォルトでは、パフォーマンスアナライザは、実験が記録されたマシンに適したマシンモデルのファイルをロードします。マシンモデルのファイルでは、メモリーオブジェクトとインデックスオブジェクトの両方を定義できます。

メトリック設定

「メトリック設定」では、「関数」、「呼び出し元-呼び出し先」、「ソース」、「逆アセンブリ」など、ほとんどのアナライザタブに表示されるメトリックを選択できます。一部のメトリックスは選択した時間またはパーセンテージで表示できるのに対して、その他のメトリックスは値として表示されます。メトリックスの一覧には、読み込まれたいずれかの実験で使用可能なすべてのメトリックスが含まれます。

メトリックスごとに、時間とパーセンテージ、または値のチェックボックスが表示されます。パフォーマンスアナライザで表示するメトリックの種類のチェックボックスを選択します。「適用」をクリックして、新しいメトリックでビューを更新します。「概要」ではメトリックを選択することもでき、ここで行なった設定と同期されます。列ヘッダーをクリックすることでメトリックをソートできます。

注記 - 表示することを選択できるのは、排他的メトリックスと包括的メトリックスだけです。属性メトリックは、排他的メトリックまたは包括的メトリックのどちらかが表示されている場合は、常に「呼び出しツリー」ビューに表示されます。

タイムライン設定

「タイムライン設定」を使用すると、107 ページの「「タイムライン」ビュー」に表示される情報を指定できます。

データの種類

表示するデータの種類を選択します。この選択は、すべての実験とすべての表示タイプに適用されます。実験にデータの種類が含まれていない場合、そのデータの種類は、選択可能なデータの種類として設定に表示されません。

CPU 使用率標本

「CPU 使用率標本」バーをプロセスごとに表示する場合に選択します。「標本」バーには、それぞれの定期的な標本についてのマイクロステート情報を要約したグラフが表示されます。

クロックプロファイリング

各 LWP、スレッド、CPU、または実験についてキャプチャーされたクロックプロファイリングデータのタイムラインバーを表示する場合に選択します。各項目のバーには、標本収集されたイベントで実行された関数の色分けされた呼び出しスタックが表示されます。

HW カウンタプロファイリング (キーワード)

HW カウンタプロファイリングデータのタイムラインバーを表示する場合に選択します。

I/O トレース

I/O トレースデータのタイムラインバーを表示する場合に選択します。

ヒープトレース

ヒープトレースデータのタイムラインバーを表示する場合に選択します。

同期トレース

同期トレース呼び出しスタックのタイムラインバーを表示する場合に選択します。

イベント状態

イベントごとにマイクロステートを表示するグラフを各タイムラインバーに追加する場合に選択します。イベント密度- イベントが発生したときに表示するグラフを各タイムラインバーに追加する場合に選択します。

データのグループ化

プロセスごとにタイムラインバーを整理する方法 (LWP 別、スレッド別、CPU 別、またはプロセス全体) を指定します。「タイムライン」ツールバーの「データをグループ化」リストを使用して、グループ化を設定することもできます。

スタック配置方法

タイムラインイベントマーカーに表示される呼び出しスタックが、リーフ関数またはルート関数のどちらに配置されるかを指定します。最後に呼び出された関数をスタックの最下部に表示する場合、リーフを選択します。この設定は、「選択の詳細」パネルに表示されるデータには影響しません。このパネルの最上部には常にリーフ関数が表示されます。

呼び出しスタックの拡大

呼び出しスタック内の各関数を表示するときに使用されるピクセル数を指定します。デフォルト値は 3 です。この設定と、行ごとに使用可能なスペースを制御するタイムライン垂直ズームとによって、深い呼び出しスタックが切り捨てられるか、完全に表示されるかが決定されます。

ソース/逆アセンブリ設定

「ソース/逆アセンブリ設定」では、「ソース」ビュー、「逆アセンブリ」ビュー、および「ソース/逆アセンブリ」ビューに表示される情報を選択できます。

コンパイラ解説

「ソース」ビューと「逆アセンブリ」ビューに表示されるコンパイラ解説のクラスを選択します。

ホットライン強調表示しきい値

「ソース」ビューと「逆アセンブリ」ビューで、高メトリックの行を強調表示するためのしきい値。このしきい値は、ソースまたは逆アセンブリが表示されているファイル内のいずれかの行に帰属する最大のメトリック値のパーセンテージです。このしきい値は、各メトリックに独立して適用されます。

ソースコード

「逆アセンブリ」ビューにソースコードを表示します。「逆アセンブリ」ビューでソースコードを表示した場合は、有効にしたクラスについてコンパイラ解説も表示されます。

ソース行のメトリック

「逆アセンブリ」ビューにソースコードのメトリックを表示します。

16 進命令

「逆アセンブリ」ビューに 16 進数で命令を表示します。

現関数のデータのみを表示する

別のビューで選択された現在の関数の命令についてのみメトリックを表示します。このオプションを選択した場合、ほかのすべての命令についてメトリックが非表示になります。

コンパイラのコマンド行フラグを表示します

ターゲットプログラムのコンパイルに使用されるコンパイラコマンドおよびオプションを表示します。コマンド行を表示するには、「ソース」ビューの最終行までスクロールします。

関数の開始行を表示します

関数の開始行をオンまたはオフに切り替えます。

呼び出しツリー設定

「呼び出しツリー設定」(「メトリックのパーセントがこのしきい値 (%) を超えるときに分岐を展開します」) では、「呼び出しツリー」ビュー内の分岐を展開するためのトリガーを設定します。呼び出しツリーの分岐で使用されているメトリックが指定されたパーセンテージ以下の場合は、「分岐を展開」や「すべての分岐を展開」などの展開アクションを選択しても、自動的には展開されません。このパーセンテージを超えている場合は展開されます。

書式設定

「書式設定」を使用すると、データビューのその他の書式設定を指定できます。

関数名の書式

関数名を C++ 関数名および Java メソッド名の長い形式、短い形式、符号化された形式のどれで表示するかを指定します。

関数名に SO 名を付加

関数またはメソッドが配置されている共有オブジェクトの名前を関数名またはメソッド名のあとに付加するには、このチェックボックスにチェックマークを付けます。

表示モード

表示モードのツールバー設定の初期値を設定します。これは、Java 実験と OpenMP 実験に対してのみ有効になります。「ユーザー」、「上級」、「マシン」の各表示モードによって、実験を表示するためのデフォルトモードが設定されます。ツールバーの「表示モード」リストを使用して、現在のビューを切り替えることができます。

Java 実験の場合:

- 「ユーザー」モードでは、インタプリタされたメソッドおよび呼び出された任意のネイティブメソッドのメトリックを表示します。特殊関数 `<no Java call stack recorded >` は、Java プログラムが実行されていたにもかかわらず、Java 仮想マシン (JVM) ソフトウェアが Java 呼び出しスタックを報告しなかったことを示します。
- 「上級」モードでは、インタプリタされたメソッドおよび呼び出された任意のネイティブメソッドについてメトリックを表示し、さらに、JVM で動的にコンパイルされたメソッドを一覧表示します。

- 「マシン」モードでは、複数の JVM コンパイルを、名前は同じであっても完全に独立した関数として表示します。このモードでは、JVM ソフトウェアのすべての関数はそのように表示されます。

Java 実験の表示モードの詳細は、200 ページの「Java プロファイリングの表示モード」を参照してください。

OpenMP 実験の場合:

- 「ユーザー」モードでは、OpenMP を使用せずにプログラムがコンパイルされるときに取得されるものに類似した、再構築済みの呼び出しスタックを表示します。OpenMP ランタイムが特定の操作を実行しているときは、<OMP-*> という形式の名前を持つ特殊関数が表示されます。
- 「上級」モードでは、「ユーザー」モードではユーザー関数と集約された、並列化されたループやタスクなどを表すコンパイラ生成の関数を示します。OpenMP ランタイムが特定の操作を実行しているときは、<OMP-*> という形式の名前を持つ特殊関数が表示されます。
- 「マシン」モードでは、特殊関数 <OMP-*> を使用せずに、すべてのスレッドのマシン呼び出しスタックを表示します。

OpenMP 実験の表示モードの詳細は、201 ページの「OpenMP実行の概要 」を参照してください。

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

比較形式

実験を比較するときのデータの表示方法を指定します。たとえば、比較実験メトリックは、x0.994 のように表示され、ベース実験と相対的な値を示します。

絶対値	ロードされたすべての実験のメトリック値を表示します。
デルタ	ベースライン実験とロードされたほかの実験のメトリック間の +/- の差を表示します。
比率	ベースライン実験とロードされたほかの実験のメトリック間の差を比率として表示します。

検索パス設定

「検索パス設定」では、「ソース」および「逆アセンブリ」ビューで注釈付きソースデータを表示するために、ロードされた実験に関連付けられたソースおよびオブジェクトファイルの検索に使用されるパスを指定します。この検索パスはまた、システム上の Java Runtime Environment の .jar ファイルを検索するためにも使用されます。特殊なディレクトリ名 $expts は、読み込まれた順序での、現在の一連の実験を指します。$expts を検索すると、親実験のみが検索され、派生実験は検査されません。

デフォルトでは、検索パスは $expts および . (現在のディレクトリ) に設定されます。パスを入力または参照し、「追加」をクリックすることによって、検索するほかのパスを追加できます。リ

スト内のパスを編集するには、パスを選択し、それを「パス」フィールド内で編集して「更新」をクリックします。検索順序を変更するには、リスト内のパスを選択し、「上へ移動」/「下へ移動」ボタンをクリックします。

検索パスの使用方法の詳細は、221 ページの「ツールがソースコードを見つけるしくみ」を参照してください。

パスマップ設定

「パスマップ設定」では、パフォーマンスアナライザでソースファイルを検索しやすくするために、ファイルパスの先頭の部分をある場所から別の場所にマップできます。パスマップは、記録されたときの元の場所から移動された実験に役立ちます。ソースを見つけることができた場合、パフォーマンスアナライザは、注釈付きソースデータを「ソース」ビューおよび「逆アセンブリ」ビューに表示できます。

ソースパス
: 実験で使用されたソースへのパスの先頭を入力します。パフォーマンスアナライザで実験を開いているときに、「選択の詳細」パネルを表示するとこのパスを見つけられます。

ターゲットパス
: パフォーマンスアナライザを実行している現在の場所からソースへのパスの先頭を入力または参照します。

  たとえば、実験に `/a/b/c/d/sourcefile` として指定されたパスが含まれており、`soucefile` が現在 `/x` 内に配置されている場合は、「パスマップ設定」を使用して `/a/b/c/d/` を `/x/` にマップできます。複数のパスマップを指定でき、各パスマップがファイルを検索するために試行されます。

パスマップの使用方法の詳細は、221 ページの「ツールがソースコードを見つけるしくみ」を参照してください。

パフォーマンスアナライザの構成ファイル

パフォーマンスアナライザでは、ユーザーがこのツールを終了すると、構成設定が自動的に保存されます。ユーザーが同じ実験をふたたび開くと、これらの設定は、前に実験を閉じたときの状態のまま構成されます。

一部の設定は名前が `config.xml` で終わる構成ファイルに保存し、「実験を開く」ダイアログからその構成ファイルを開いたときに任意の実験に適用できます。構成は、自分だけが使用する場所に保存することも、ほかのユーザーが使用できるように共有の場所に保存することもできます。

実験を開くと、パフォーマンスアナライザは使用可能な構成ファイルのデフォルトの場所を検索し、開いている実験にどの構成を適用するかをユーザーが選択できるようにします。

また、「ツール」>「設定を .er.rc としてエクスポート」を使用して、er_print で読み取り可能な .er.rc ファイルに設定をエクスポートすることもできます。これにより、er_print とパフォーマンスアナライザで同じメトリックを有効にしておくことができます。

♦ ♦ ♦ 第 5 章

5 er_print コマンド行パフォーマンス解析ツール

この章では、er_print ユーティリティーを使用してパフォーマンス解析を行う方法を説明します。

この章では、次の内容について説明します。

- 142 ページの「er_print について」
- 142 ページの「er_print の構文」
- 143 ページの「メトリックリスト」
- 146 ページの「関数リストを制御するコマンド」
- 149 ページの「呼び出し元 - 呼び出し先リストを管理するコマンド」
- 151 ページの「呼び出しツリーリストを制御するコマンド」
- 151 ページの「トレースデータに共通のコマンド」
- 152 ページの「リークリストと割り当てリストを管理するコマンド」
- 153 ページの「ソースリストと逆アセンブリリストを管理するコマンド」
- 156 ページの「PC と行を制御するコマンド」
- 157 ページの「ソースファイルの検索を制御するコマンド」
- 158 ページの「データ領域リストを制御するコマンド」
- 159 ページの「インデックスオブジェクトリストを制御するコマンド」
- 162 ページの「OpenMP インデックスオブジェクトのコマンド」
- 163 ページの「スレッドアナライザをサポートするコマンド」
- 163 ページの「実験、標本、スレッド、および LWP を一覧表示するコマンド」
- 165 ページの「実験データのフィルタリングを制御するコマンド」
- 167 ページの「ロードオブジェクトの展開と短縮を制御するコマンド」
- 169 ページの「メトリックを一覧するコマンド」
- 170 ページの「出力を制御するコマンド」
- 172 ページの「その他の情報を出力するコマンド」
- 174 ページの「.er.rc ファイル内のデフォルトの設定」
- 176 ページの「その他のコマンド」
- 177 ページの「式の文法」
- 181 ページの「er_print コマンドの例」

er_print について

er_print ユーティリティーは、パフォーマンスアナライザがサポートするさまざまなデータビューをテキスト形式で出力します。これらの情報は、ファイルにリダイレクトしないかぎり、標準出力に書き出されます。er_print には、引数として、コレクタが生成した実験名または実験グループ名を指定する必要があります。

er_print ユーティリティーは、Oracle Solaris Studio 12.3、Oracle Solaris Studio 12.4、および Oracle Developer Studio 12.5 で記録された実験でのみ動作します。その他のバージョンで記録された実験を使用すると、エラーが報告されます。古い実験がある場合、その実験が記録されたリリースの er_print のバージョンを使用する必要があります。

er_print ユーティリティーを使用すると、関数、呼び出し元と呼び出し先、ソースコードリスト、逆アセンブリリスト、標本収集情報、データ領域データ、スレッド解析データ、および実行統計のパフォーマンスメトリックを表示できます。

er_print を複数の実験または実験グループで起動した場合、デフォルトでは実験データが集計されますが、実験を比較することもできます。詳細は、172 ページの「compare { on | off | delta | ratio }」を参照してください。

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

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

er_print の構文

er_print ユーティリティーのコマンド行構文は、次のとおりです。

er_print [-script *script* | –*command* | - | -V] *experiment-list*

er_print ユーティリティーのオプションは、次のとおりです。

–

キーボードから入力された er_print コマンドを読み取ります。

-script *script*

script ファイルからコマンドを読み取ります。script ファイルは、一連の er_print コマンドを含むファイルで、1 行に 1 つの er_print コマンドがあります。-script オプションを指定しない場合、er_print は、端末またはコマンド行からコマンドを読み取ります。

-*command* [*argument*]

指定されたコマンドを処理します。

-V

バージョン情報を表示して終了します。

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

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

er_print ユーティリティーで使用できるコマンドについては、以降のセクションで示します。

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

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

引数なしで呼び出された場合、er_print はヘルプメッセージを出力します。

メトリックリスト

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

metric-keyword-1[:*metric-keyword2*…]

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

flavorvisibilityname

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

[*visibility*]*name*

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

指定可能なメトリックフレーバー文字を表8「メトリックフレーバー文字」に示します。複数のフレーバー文字を含むメトリックキーワードが展開されて、メトリックキーワードリストになります。たとえば、ie.user は、展開されて i.user:e.user になります。

表 8 メトリックフレーバー文字

文字	説明
e	排他的メトリック値を表示します。
i	包括的メトリック値を表示します。

指定可能なメトリック表示形式文字を表9「メトリック表示形式文字」に示します。表示形式文字列の文字の順序は、対応するメトリックの表示順序には影響しません。たとえば、i%.user と i.%user はともに i.user:i%user と解釈されます。

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

表 9 メトリック表示形式文字

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

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

静的メトリックの場合、ソート順序の定義という観点からは、表示形式文字のピリオド (.)、正符号 (+)、パーセント記号 (%) は同等と見なされます。つまり、sort i%user、sort i.user、sort i+user はすべて、「どのような形式で表示するにせよ、パフォーマンスアナライザは包括的ユーザー CPU 時間を基準にソートする」ことを意味します。また、sort i!user は、「表示するかどうかに関係なく、パフォーマンスアナライザは包括的ユーザー CPU 時間を基準にソートする」という意味になります。

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

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

表10「メトリック名文字列」に、タイミングメトリック、同期遅延メトリック、メモリー割り当てメトリック、MPI トレースメトリック、および 2 つの一般的なハードウェアカウンタメトリックに指定可能な `er_print` メトリック名文字列を示します。ほかのハードウェアカウンタメトリックの場合、メトリック名文字列はカウンタ名と同じです。ロードされた実験に適用できるすべてのメトリック名文字列のリストは、`metric_list` コマンドで表示できます。カウンタ名を一覧表示するには、`collect -h` コマンドをほかの引数を指定せずに発行します。ハードウェアカウンタについての詳細は、26 ページの「ハードウェアカウンタプロファイリングデータ」を参照してください。

表 10 メトリック名文字列

カテゴリ	文字列	説明
クロックプロファイリングメトリック	`total`	スレッド合計時間
	`totalcpu`	合計 CPU 時間
	`user`	ユーザー CPU 時間
	`system`	システム CPU 時間
	`trap`	トラップ CPU 時間
	`lock`	ユーザーロック時間
	`datapfault`	データページフォルト時間
	`textpfault`	テキストページフォルト時間
	`kernelpfault`	カーネルページフォルト時間
	`stop`	停止時間
	`wait`	CPU 待ち時間
	`sleep`	スリープ時間
ハードウェアカウンタメトリック	`insts`	発行された命令 サポートするすべてのシステムで使用できます。
	`cycles`	CPU サイクル サポートするほとんどのシステムで使用できます。さらに、それぞれのプロセッサには独自のカウンタセットがあります。システムのすべてのカウンタのリストを表示するには、`collect -h` を使用します。
	`CPI`	`cycles` および `insts` メトリックから計算された命令当たりのサイクル数。両方のカウンタが記録される場合にのみ使用できます。
	`IPC`	`cycles` および `insts` メトリックから計算されたサイクル当たりの命令数。両方のカウンタが記録される場合にのみ使用できます。
OpenMP プロファイリングメトリック	`ompwork`	作業を直列または並列で実行する時間
	`ompwait`	OpenMP ランタイムが同期を待っている時間
	`masterthread`	マスタースレッド時間は、マスタースレッドで費やされた合計時間です。これは、Oracle Solaris 実験からのみ使用できます。これは時計時間に対応します。
同期遅延メトリック	`sync`	同期待ち時間
	`syncn`	同期待ち回数
ヒープトレースメトリック	`heapalloccnt`	割り当て数

カテゴリ	文字列	説明
	heapallocbytes	割り当てバイト数
	heapleakcnt	リーク数
	heapleakbytes	リークバイト数
I/O トレースメトリック	ioreadbytes	読み取りバイト数
	ioreadcnt	読み取り数
	ioreadtime	読み取り時間
	iowritebytes	書き込みバイト数
	iowritecnt	書き込み数
	iowritetime	書き込み時間
	ioothrcnt	その他の I/O 数
	iootherttime	その他の I/O 時間
	ioerrornt	I/O エラー数
	ioerrortime	I/O エラー時間
スレッドアナライザのメトリック	raccesses	データ競合のアクセス
	deadlocks	デッドロック
MPI トレースメトリック	mpitime	MPI 呼び出しに費やされた時間
	mpisend	開始された MPI ポイントツーポイント送信数
	mpibytessent	MPI で送信されるバイト数
	mpireceive	完了した MPI ポイントツーポイント受信数
	mpibytesrecv	MPI で受信されるバイト数
	mpiother	その他の MPI 関数の呼び出し数
MPI プロファイリングメトリック	mpiwork	MPI ランタイムで作業する (処理の要求やメッセージなど) 時間
	mpiwait	MPI ランタイムで、イベント、バッファー、またはメッセージを待っている時間

表10「メトリック名文字列」に示した名前文字列のほかに、2 つの名前文字列をデフォルトのメトリックリスト内でのみ使用できます。この 2 つの文字列は、任意のハードウェアカウンタ名に一致する `hwc` と、任意のメトリック名文字列に一致する `any` です。また、`cycles` と `insts` は、SPARC® プラットフォームと x86 プラットフォームに共通のものですが、それ以外のアーキテクチャー固有のフレーバーも存在します。

ロードした実験から使用可能なメトリックを確認するには、`metric_list` コマンドを発行します。

関数リストを制御するコマンド

ここでは、関数情報の表示を制御するコマンドを説明します。

functions

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

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

書き込む行数は、limit コマンドを使用して制限できます (170 ページの「出力を制御するコマンド」を参照)。

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

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

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

metrics *metric-spec*

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

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

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

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

- 排他的ユーザー CPU 時間 (百分率)

デフォルトでは、174 ページの「.er.rc ファイル内のデフォルトの設定」の説明のように、.er.rc ファイルから処理された dmetrics コマンドに基づいたメトリック設定が使用されます。metrics コマンドで明示的に *metric-spec* を default に設定した場合は、記録するデータに適したデフォルト設定が復元されます。

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

metric-spec を省略した場合は、現在のメトリックの設定が表示されます。

metrics コマンドは、関数リスト用のメトリックを設定するほか、呼び出し元-呼び出し先、データ派生出力、およびインデックスオブジェクトのメトリックを設定します。呼び出し元-呼び出し先メトリックは、関数リストに包括的メトリックまたは排他的メトリックが表示されるメトリックに対応する属性メトリックと、静的メトリックを表示します。

データ領域メトリックは、データが使用可能で、かつ関数リストに包括的メトリックまたは排他的メトリックが表示されるメトリックに対応するデータ領域メトリックと、静的メトリックを表示します。

インデックスオブジェクトメトリックは、関数リストに包括的メトリックまたは排他的メトリックが表示されるメトリックに対応するインデックスオブジェクトメトリックと、静的メトリックを表示します。

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

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

メトリックリストの構文については、143 ページの「メトリックリスト」を参照してください。指定可能なメトリックを一覧表示するには、metric_list コマンドを使用します。

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

sort *metric_spec*

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

デフォルトでは、174 ページの「.er.rc ファイル内のデフォルトの設定」の説明のように、メトリックソート設定は、.er.rc ファイルから処理された dsort コマンドに基づいています。sort コマンドで明示的に *metric_spec* を default に設定した場合は、デフォルトの設定が使用されます。

文字列 *metric-spec* は、次の例に示すように、143 ページの「メトリックリスト」に示すメトリックキーワードの 1 つです。

```
sort i.user
```

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

fsummary

関数リスト内の各関数について、サマリーパネルを出力します。出力するパネル数は、limit コマンドを使用して制限できます (170 ページの「出力を制御するコマンド」を参照)。

サマリーメトリックパネルには、関数またはロードオブジェクトの名前、アドレス、およびサイズが表示されます。関数の場合、ソースファイル、オブジェクトファイル、およびロードオブジェクトの名前が含まれます。パネルには、選択した関数またはロードオブジェクトについて、記録された排他的メトリックと包括的メトリックの両方の値とパーセンテージが表示されます。

fsingle *function-name* [*N*]

指定された関数のサマリーパネルを出力します。複数の関数が同じ名前を持つ場合には、オプションのパラメータ *N* が必要です。指定の関数名を持つ *N* 番目の関数について、サマリーメトリックパネルが出力されます。コマンド行でコマンドを入力する場合、*N* を必ず指定する必要があります。不要な場合は無視されます。*N* が必要であるときに *N* を使用しないでコマンドを対話的に入力すると、対応する *N* 値を持つ関数のリストが出力されます。

関数のサマリーメトリックについては、fsummary コマンドの解説を参照してください。

呼び出し元 - 呼び出し先リストを管理するコマンド

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

callers-callees

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

各呼び出し元 - 呼び出し先レポート内では、呼び出し元 - 呼び出し先のソートメトリック (csort) に従って呼び出し元と呼び出し先がソートされます。出力するパネル数は、limit コマンドを使用して制限できます (170 ページの「出力を制御するコマンド」を参照)。選択されている関数 (中央の関数) は、次のようにアスタリスクで示されます。

```
Attr.       Name
User CPU
 sec.
4.440          commandline
0.            *gpf
4.080          gpf_b
0.360          gpf_a
```

この例では、関数 gpf が選択されています。この関数は commandline によって呼び出され、gpf_a と gpf_b を呼び出します。

呼び出し元-呼び出し先を使用すると、単一の関数だけでなく、スタックフラグメントのデータが低速になる可能性があります。フラグメントは cprepend、cappend、crmfirst、および crmlast コマンドで制御できます。

csingle *function-name* **[***N***]**

指定された関数の呼び出し元 - 呼び出し先パネルを出力します。複数の関数が同じ名前を持つ場合には、オプションのパラメータ *N* が必要です。指定の関数名を持つ *N* 番目の関数について、呼び出し元 - 呼び出し先パネルが出力されます。コマンド行でコマンドを入力する場合、*N* を必ず指定する必要があります。不要な場合は無視されます。*N* が必要であるときに *N* を使用しないでコマンドを対話的に入力すると、対応する *N* 値を持つ関数のリストが出力されます。

cprepend *function-name [N | ADDR]*

呼び出しスタックを構築する際、現在の呼び出しスタックフラグメントの先頭に名前付き関数を追加します。関数名があいまいな場合、オプションパラメータが必要になります。パラメータの指定方法についての詳細は、153 ページの「source|src { *filename* | *function-name* } [*N*]」を参照してください。

cappend *function-name [N | ADDR]*

呼び出しスタックを構築する際、現在の呼び出しスタックフラグメントの末尾に名前付き関数を追加します。関数名があいまいな場合、オプションパラメータが必要になります。パラメータの

指定方法についての詳細は、153 ページの「source|src { *filename* | *function-name* } [*N*]」を参照してください。

crmfirst

呼び出しスタックを構築する際、呼び出しスタックセグメントから最初のフレームを削除します。

crmlast

呼び出しスタックを構築する際、呼び出しスタックセグメントから最後のフレームを削除します。

呼び出しツリーリストを制御するコマンド

ここでは、呼び出しツリー用のコマンドについて説明します。

calltree

実験から動的コールグラフを表示します。各レベルに履歴メトリックが表示されます。

トレースデータに共通のコマンド

このセクションでは、トレースデータを含む実験で使用できるコマンドについて説明します。

datasize

トレースデータ内で参照されるデータのサイズの分布を対数スケールで書き込みます。ヒープトレースの場合、サイズは割り当てサイズまたはリークサイズです。I/O トレースの場合、サイズは転送済みのバイト数です。

duration

トレースデータ内でのイベントの経過時間の分布を対数スケールで書き込みます。同期トレースの場合、経過時間は同期遅延です。I/O トレースの場合、経過時間は I/O 操作に費やした時間です。

リークリストと割り当てリストを管理するコマンド

ここでは、メモリーの割り当てと割り当て解除に関するコマンドについて説明します。

leaks

共通呼び出しスタックによって集計されたメモリーリークのリストを表示します。各エントリは、リーク総数、および指定の呼び出しスタックでリークした総バイト数を示します。このリストは、リークしたバイト数を基準としてソートされます。

allocs

共通呼び出しスタックによって集計されたメモリー割り当てのリストを表示します。各エントリは、割り当ての数、および指定の呼び出しスタックに割り当てられた総バイト数を示します。このリストは、割り当てられたバイト数を基準としてソートされます。

heap

共通呼び出しスタックによって集計された割り当ておよびリークのリストを書き込みます。

heapstat

アプリケーションのピークメモリー使用量を含む、ヒープ使用量に関する全体的な統計を書き込みます。

I/O アクティビティーレポートを制御するコマンド

このセクションでは、I/O アクティビティーに関連するコマンドについて説明します。

ioactivity

ファイルでソートされた、すべての I/O アクティビティーのレポートを書き込みます。

iodetail

仮想ファイル記述子でソートされた、すべての I/O アクティビティーのレポートを書き込みます。ファイルを開いたときに同じファイル記述子が返された場合でも、ファイルを開くたびに異なる仮想ファイル記述子が生成されます。

iocallstack

呼び出しスタックでソートされ、同じ呼び出しスタックですべてのイベントにわたって集約された、すべての I/O アクティビティーのレポートを書き込みます。集約された呼び出しスタックごとに、呼び出しスタックトレースを含めます。

iostat

すべての I/O アクティビティーに関するサマリー統計情報を書き込みます。

ソースリストと逆アセンブリリストを管理するコマンド

ここでは、注釈付きソースおよび逆アセンブリコードの表示を制御するコマンドを説明します。

source|src { *filename* | *function-name* } [*N*]

指定したファイル、または指定した関数を含むファイルの注釈付きソースコードを書き出します。いずれの場合も、指定したファイルはパスの通っているディレクトリに存在する必要があります。ソースが GNU Fortran コンパイラでコンパイルされている場合は、ソースに表示されるように、関数名のあとにアンダースコアを 2 つ追加する必要があります。

省略可能なパラメータ *N* (正の整数) は、ファイル名または関数名があいまいでない場合にだけ使用します。このパラメータを指定した場合は、*N* 番目の候補が使用されます。数値指定子を使用せずにあいまいな名前を指定した場合、`er_print` ユーティリティーは可能性のあるオブジェクトファイル名のリストを出力します。指定した名前が関数の場合、関数の名前がオブジェクトファイル名に付加され、そのオブジェクトファイルの *N* の値を表す数字も出力されます。

関数名は *function"file"* としても指定できます。この場合、*file* は、関数の代替ソースコンテキストを指定するために使用されます。最初の命令の直後にその関数のインデックス行が追加されます。インデックス行は、次の書式で山括弧内にテキストとして表示されます。

```
<Function: f_name>
```

関数のデフォルトソースコンテキストは、その関数の最初の命令が帰するソースファイルとして定義されます。これは通常、関数を含むオブジェクトモジュールを生成するためにコンパイルされたソースファイルです。代替ソースコンテキストは、関数に属する命令を含むほかのファイルから構成されます。このようなコンテキストには、インクルードファイルの命令と、指定の関数にインライン化された関数の命令が含まれます。代替のソースコンテキストが存在する場合、デフォルトのソースコンテキストの冒頭に、代替ソースコンテキストが置かれている場所を示す拡張インデックス行のリストを次の形式で組み込みます。

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

注記 - コマンド行から `er_print` ユーティリティーを起動するときに `-source` 引数を使用する場合は、file の引用符の前にバックスラッシュのエスケープ文字を付加する必要があります。つまり、関数名の形式は、`function\"file\"` となります。`er_print` ユーティリティーが対話モードにあるときは、バックスラッシュは不要です。使用しないでください。

通常、デフォルトのソースコンテキストが使用された場合は、そのファイルに入っているすべての関数についてメトリックが表示されます。ファイルを明示的に指定した場合は、指定した関数についてのみ、メトリックが表示されます。

disasm|dis { *filename* **|** *function-name* **} [** *N***]**

指定したファイル、または指定した関数を含むファイルの注釈付き逆アセンブリコードを出力します。指定したファイルは、パスの通っているディレクトリに存在する必要があります。

省略可能なパラメータ *N* の意味は、`source` コマンドと同じです。

scc *com-spec*

注釈付きソースのリストに含めるコンパイラの解説クラスを指定します。クラスリストはコロンで区切ったクラスのリストであり、次のメッセージクラスがゼロ個以上含まれています。

表 11 コンパイルコメントメッセージクラス

クラス	意味
`b[asic]`	基本的なレベルのメッセージを表示します。
`v[ersion]`	ソースファイル名、最終修正日付、コンパイラコンポーネントのバージョン、コンパイル日付とオプションなどのバージョンメッセージを表示します。

クラス	意味
pa[rallel]	並列化に関するメッセージを表示します。
q[uery]	最適化に影響するコードに関する問い合わせメッセージを表示します。
l[oop]	ループの最適化と変換に関するメッセージを表示します。
pi[pe]	ループのパイプライン化に関するメッセージを表示します。
i[nline]	関数のインライン化に関するメッセージを表示します。
m[emops]	ロード、ストア、プリフェッチなどのメモリー操作に関するメッセージを表示します。
f[e]	フロントエンドメッセージを表示します。
co[degen]	コードジェネレータのメッセージを表示します。
cf	ソースの下部にコンパイラのフラグを表示します。
all	すべてのメッセージを表示します。
none	メッセージを表示しません。

all および none クラスは常に単独で指定します。

scc コマンドを省略した場合は、basic がデフォルトのクラスになります。*class-list* が空の状態で scc コマンドを入力した場合、コンパイラのコメントは出力されません。通常、scc コマンドは、.er.rc ファイルでのみ使用します。

sthresh *value*

注釈付きソースコードでのメトリックの強調表示に使用するしきい値の百分率を指定します。ファイル内のソース行で、メトリック値が、そのメトリックの最大値の *value* % 以上である場合、そのメトリックが発生する行の先頭に ## が挿入されます。

dcc *com-spec*

注釈付き逆アセンブリリストに含めるコンパイラの解説クラスを指定します。クラスリストは、コロンで区切られたクラスのリストです。利用可能なクラスのリストは、表11「コンパイルコメントメッセージクラス」に示す注釈付きソースコードリストのクラスリストと同じです。次の表のオプションをクラスリストに追加できます。

表 12 dcc コマンドの追加オプション

オプション	意味
h[ex]	命令の 16 進値を示します。
noh[ex]	命令の 16 進値を示しません。

オプション	意味
s[rc]	ソースリストと注釈付き逆アセンブリリストをインタリーブします。
nos[rc]	ソースリストと注釈付き逆アセンブリリストをインタリーブしません。
as[rc]	注釈付きソースコードと注釈付き逆アセンブリリストをインタリーブします。

dthresh *value*

注釈付き逆アセンブリコードでのメトリックの強調表示に使用するしきい値の百分率を指定します。ファイル内の命令行で、メトリック値が、そのメトリックの最大値の *value* % 以上である場合、そのメトリックが発生する行の先頭に ## が挿入されます。

cc *com-spec*

注釈付きのソースと逆アセンブリリストに含めるコンパイラの解説クラスを指定します。クラスリストは、コロンで区切られたクラスのリストです。利用可能なクラスのリストは、表11「コンパイルコメントメッセージクラス」に示す注釈付きソースコードリストのクラスリストと同じです。

PC と行を制御するコマンド

次のコマンドは、プログラムカウンタおよび行の情報を表示する方法を制御します。

pcs

現在のソートメトリックで整列されたプログラムカウンタ (Program Counter、PC) と、そのメトリックのリストを出力します。このリストには、`object_select` コマンドで関数を非表示にした各ロードオブジェクトのメトリックを集計した行が含まれています。

psummary

PC リスト内の各 PC のサマリーメトリックパネルを現在のソートメトリックで指定された順序で出力します。

lines

現在のソートメトリックで整列されたソース行と、そのメトリックのリストを書き込みます。このリストには、行番号情報を持っていない各関数またはソースファイルが未知である各関数のメトリックを集計した行と、object_select コマンドで関数を非表示している各ロードオブジェクトのメトリックを集計した行が含まれています。

lsummary

行リスト内の各行のサマリーメトリックパネルを現在のソートメトリックで指定された順序で出力します。

ソースファイルの検索を制御するコマンド

er_print ユーティリティーは、実験で参照されているソースファイルおよびロードオブジェクトファイルを探します。このセクションで説明されているディレクティブを使用すると、er_print が実験で参照されているファイルを見つけやすくなります。

これらのディレクティブの使用方法を含め、実験のソースコードを見つけるために使用される処理については、221 ページの「ツールがソースコードを見つけるしくみ」を参照してください。

setpath *path-list*

ソースファイルおよびオブジェクトファイルの検索に使用するパスを設定します。*path-list* は、コロンで区切られたディレクトリ、jar ファイル、または zip ファイルのリストです。ディレクトリ名にコロン文字がある場合は、バックスラッシュでコロンをエスケープします。特殊なディレクトリ名 $expts は、ロードされた順序での、現在の一連の実験を指します。単一の $ 文字を使用すると短縮できます。

デフォルトパスは $expts:.. で、これは読み込まれた実験のディレクトリと現在の作業ディレクトリです。

現在のパスを表示するには、setpath を引数なしで使用します。

.er.rc ファイルでは、setpath コマンドを使用しないでください。

addpath *path-list*

現在の `setpath` の設定に *path_list* を付加します。

`addpath` コマンドは、`.er.rc` ファイル内で使用することができ、連結されます。

pathmap *old-prefix new-prefix*

`addpath` または `setpath` で設定された *path-list* を使用してファイルが見つからなかった場合、`pathmap` コマンドを使用して、1 つまたは複数のパスの再マッピングを指定できます。ソースファイル、オブジェクトファイル、または共有オブジェクトのパス名が *old-prefix* で指定した接頭辞で始まる場合、古い接頭辞は *new-prefix* で指定した接頭辞に置き換えられます。結果のパスは、ファイルの検索に使用されます。複数の `pathmap` コマンドが提供されており、それぞれが、ファイルが見つかるまで試行されます。

データ領域リストを制御するコマンド

データ領域のコマンドは、Oracle Solaris x86 または SPARC システム上で正確なカウンタに対してデフォルトで、または明示的にメモリー領域およびデータ領域データが記録されたハードウェアカウンタ実験にのみ適用できます。詳細は、`collect`(1) のマニュアルページを参照してください。

データ領域データは、`-xhwcprof` フラグでコンパイルされた関数内で発生したプロファイルヒットでのみ使用できます。`-xhwcprof` フラグは、C、C++、および Fortran コンパイラでのコンパイルに適用でき、SPARC プラットフォームでのみ意味があります。その他のプラットフォームでは無視されます。

これらのデータの種類についての詳細は、26 ページの「ハードウェアカウンタプロファイリングデータ」を参照してください。ハードウェアカウンタオーバーフロープロファイリングの実行に使用するコマンドについては、64 ページの「`collect -h` を使用したハードウェアカウンタプロファイリング」を参照してください。

`-xhwcprof` コンパイラオプションについては、『Oracle Developer Studio 12.5: Fortran ユーザーズガイド』、『Oracle Developer Studio 12.5: C ユーザーズガイド』、または『Oracle Developer Studio 12.5: C++ ユーザーズガイド』を参照してください。

data_objects

データオブジェクトのリストを、それらのメトリックとともに書き込みます。

data_single *name* **[***N***]**

指定されたデータオブジェクトのサマリーメトリックパネルを書き込みます。オブジェクト名があいまいな場合には、省略可能なパラメータ N が必要です。ディレクティブがコマンド行にある場合には N は必要です。不要な場合は無視されます。

data_layout

データ派生メトリックデータを持つすべてのプログラミングデータオブジェクトについて、注釈付きのデータオブジェクトレイアウトを書き込みます。データは、各構造をひとまとめにして、現在のデータソートメトリック値によってソートされます。各集約データオブジェクトをそれに帰属する合計メトリックとともに表示したあと、そのすべての要素をオフセット順に表示します。各要素にはそれぞれのメトリックと、32 バイトブロックを基準にしたサイズと場所を示すインジケータが含まれます。ここでは:

<	要素は、ブロック全体に一致します。
/	要素は、ブロックを開始します。
|	要素は、ブロックの内部にあります。
¥	要素は、ブロックを完了します。
#	要素サイズには、複数のブロックが必要です。
X	要素は複数のブロックにわたっていますが、1 つのブロック内に適合することがあります。
?	未定義。

インデックスオブジェクトリストを制御するコマンド

インデックスオブジェクトのコマンドは、すべての実験に使用できます。インデックスオブジェクトリストは、記録されたデータからインデックスを計算できるオブジェクトのリストです。特に、スレッド、CPU、標本、および秒のインデックスオブジェクトが事前に定義されています。`indxobj_list` コマンドで完全なリストを取得できます。その他のインデックスオブジェクトは、`indxobj_define` コマンドで定義できます。

次のコマンドは、インデックスオブジェクトのリストを制御します。

indxobj *indxobj-type*

所定のタイプのインデックスオブジェクトとそのメトリックのリストを出力します。インデックスオブジェクトのメトリックとソートは、排他的メトリックだけが含まれる点を除き、関数リストと同じです。名前 *indxobj-type* を直接、コマンドとして使用することもできます。

indxobj_list

`indxobj` コマンド内で *indxobj-type* に使用する、既知のタイプのインデックスオブジェクトのリストを書き込みます。特に、スレッド、CPU、標本、および秒のインデックスオブジェクトが事前に定義されています。

indxobj_define *indxobj-type index-exp*

新しいタイプのインデックスオブジェクトを、*index-exp* で指定されたオブジェクトへのパケットのマッピングを使用して定義します。式の構文については、177 ページの「式の文法」に説明があります。

indxobj-type は、定義されていないものにする必要があります。その名前は、大文字と小文字が区別され、すべて英数字または「_」文字で構成されている必要があり、1 文字目は英字である必要があります。

index-exp の構文が正しくなかった場合は、エラーが返され、定義が無視されます。*index-exp* に空白が含まれている場合は、それを二重引用符 (") で囲む必要があります。

`<Unknown>` インデックスオブジェクトのインデックスは -1 です。また、新しいインデックスオブジェクトを定義するために使用される式では、`<Unknown>` の認識をサポートしている必要があります。

たとえば、仮想 PC また物理 PC に基づくインデックスオブジェクトの場合、式は次の形式になっている必要?詰縦介麿リストの例を示します。

```
VIRTPC>0?VIRTPC:-1
```

メモリーオブジェクトリストを制御するコマンド

メモリーオブジェクトのコマンドは、Solaris X86 または SPARC システム上で正確なカウンタに対してデフォルトで、または明示的にメモリー領域データが記録されたハードウェアカウンタ実験にのみ適用できます。詳細は、`collect`(1) のマニュアルページを参照してください。

メモリーオブジェクトは、キャッシュライン、ページ、メモリーバンクなどの、メモリーサブシステム内のコンポーネントです。このオブジェクトは、記録されている仮想または物理アドレスから計算されたインデックスから決定されます。仮想および物理ページに対して、8K バイト、64K バイト、512K バイト、および 4M バイトのサイズのメモリーオブジェクトが事前に定義されています。その他のオブジェクトは、`memobj_define` コマンドで定義できます。

memobj *mobj-type*

所定のタイプのメモリーオブジェクトと現在のメトリックのリストを出力します。メトリックはデータ領域のリストとして、ソートされ表示されます。名前 `mobj_type` を直接、コマンドとして使用することもできます。

mobj_list

`memobj` コマンド内で *mobj-type* に使用する、既知のタイプのメモリーオブジェクトのリストを書き込みます。

mobj_define *mobj-type index-exp*

新しいタイプのメモリーオブジェクトを、*index-exp* で指定されたオブジェクトへの VA/PA のマッピングを使用して定義します。式の構文については、177 ページの「式の文法」に説明があります。

mobj-type は定義されていないものであり、既存のどのコマンド、またはどのインデックスオブジェクトタイプ (下記を参照) にも一致していないものにする必要があります。その名前は、すべて英数字または「_」文字で構成されている必要があり、1 文字目は英字である必要があります。

index-exp は、構文的に正しいものである必要があります。構文的に正しくない場合はエラーが返され、定義は無視されます。

`<Unknown>` メモリーオブジェクトのインデックスは -1 です。また、新しいメモリーオブジェクトを定義するために使用される式では、`<Unknown>` の認識をサポートしている必要があります。たとえば、VADDR ベースのオブジェクトの場合、式は次の形式である必要があります。

`VADDR>255?`*expression* `:-1`

PADDR ベースのオブジェクトの場合、式は次の形式である必要があります。

`PADDR>0?`*expression*`:-1`

memobj_drop *mobj_type*

指定されたタイプのメモリーオブジェクトを削除します。

machinemodel *model_name*

指定されたマシンモデルで定義されているメモリーオブジェクトを作成します。*model_name* は、ユーザーの現在のディレクトリまたはユーザーのホームディレクトリ内のファイル名か、あるいはこのリリースで定義されているマシンモデルの名前です。マシンモデルのファイルは、`.ermm` の接尾辞で格納されています。`machinemodel` コマンドにある *model_name* がその接尾辞で終わらない場合は、`.ermm` が付加された *model_name* が使用されます。*model_name* が `/` で始まる場合、それは絶対パスであると見なされ、そのパスのみが (必要に応じて、`.ermm` を付加して) 試行されます。*model_name* に `/` が含まれている場合は、現在のディレクトリまたはユーザーのホームディレクトリを基準にしたそのパス名のみが試行されます。

マシンモデルのファイルには、コメントと `mobj_define` コマンドを含めることができます。その他のコマンドはすべて無視されます。`machinemodel` コマンドは、`.er.rc` ファイル内に指定できます。マシンモデルが (コマンドによって、またはマシンモデルが中に記録されている実験を読み取ることによって) ロードされた場合、以降の `machinemodel` コマンドは以前のマシンモデルからの定義をすべて削除します。

model_name がない場合は、すべての既知のマシンモデルのリストを出力します。*model_name* が長さ 0 の文字列である場合は、ロードされたマシンモデルをすべてアンロードします。

OpenMP インデックスオブジェクトのコマンド

次のコマンドを使用して、OpenMP インデックスオブジェクトの情報を出力します。

OMP_preg

実験で実行された OpenMP 並列領域とそれぞれのメトリックのリストを出力します。このコマンドは、OpenMP 3.0 または 3.1 のパフォーマンスデータを使用した実験でのみ使用できます。

OMP_task

実験で実行された OpenMP タスクとそれぞれのメトリックのリストを出力します。このコマンドは、OpenMP 3.0 または 3.1 のパフォーマンスデータを使用した実験でのみ使用できます。

スレッドアナライザをサポートするコマンド

次のコマンドはスレッドアナライザに対応しています。取得されるデータや表示されるデータについての詳細は、『Oracle Developer Studio 12.5: スレッドアナライザユーザーズガイド』を参照してください。

races

実験におけるすべてのデータ競合リストを書き出します。データ競合レポートは、データ競合検出データを使用した実験でのみ利用できます。

rdetail *race-id*

指定された *race-id* の詳細情報を書き込みます。*race-id* を `all` に設定した場合は、すべてのデータ競合の詳細情報が表示されます。データ競合レポートは、データ競合検出データのある実験からのみ取得できます。

deadlocks

実験で検出されたすべての実際のデッドロックと潜在的なデッドロックのリストを書き出します。デッドロックレポートは、デッドロック検出データを使用した実験でのみ利用できます。

ddetail *deadlock-id*

指定された *deadlock-id* の詳細情報を書き込みます。*deadlock-id* を `all` に設定した場合は、すべてのデッドロックの詳細情報が表示されます。デッドロックレポートは、デッドロック検出データを使用した実験でのみ利用できます。

実験、標本、スレッド、および LWP を一覧表示するコマンド

ここでは、実験、標本、スレッド、および LWP を一覧表示するコマンドについて説明します。

experiment_list

読み込まれている実験をそれぞれの ID 番号とともにすべて一覧表示します。各実験は、標本、スレッド、または LWP を選択する際に使用されるインデックス、および拡張フィルタリングに使用できる PID とともに一覧表示されます。

次に、実験リストの例を示します。

```
(er_print) experiment_list
 ID Sel   PID Experiment
=== === ===== ================
  1 yes 13493 test.1.er
  2 yes 24994 test.2.er
  3 yes 25653 test.2.er/_f8.er
  4 yes 25021 test.2.er/_x5.er
```

sample_list

解析の対象として現在選択されている標本の一覧を表示します。

次に、標本リストの例を示します。

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

lwp_list

解析の対象として現在選択されている LWP の一覧を表示します。

thread_list

解析の対象として現在選択されているスレッドの一覧を表示します。

cpu_list

解析の対象として現在選択されている CPU の一覧を表示します。

実験データのフィルタリングを制御するコマンド

実験データのフィルタリングは、次の 2 つの方法で指定できます。

- フィルタ式を指定する。このフィルタ式はデータレコードごとに評価され、そのレコードを含めるかどうかを決定します。
- フィルタリング用の実験、標本、スレッド、CPU、および LWP を選択する。

フィルタ式の指定

フィルタ式は `filters` コマンドで指定できます。

filters *filter-exp*

filter-exp は式であり、この式が真と評価されたデータレコードは含まれ、偽と評価されたデータレコードは含まれません。式の文法については、177 ページの「式の文法」に説明があります。

フィルタ式のキーワードの一覧表示

実験のフィルタ式で使用できるオペランドまたはキーワードの一覧を表示できます。

describe

フィルタ式の構築に使用できキーワードのリストを出力します。フィルタ式の一部のキーワードと文法については、177 ページの「式の文法」を参照してください。

フィルタリング用の標本、スレッド、LWP、および CPU の選択

次に、選択の構文の例を示します。この構文はコマンドの説明で使用されます。

[*experiment-list*:]*selection-list*[+[
experiment-list:]*selection-list* …]

選択リスト

各選択リストの前には、空白なしの 1 つのコロンで区切って実験リストを指定できます。複数の選択を指定するには、選択リストを正符号 (+) で結合します。

実験リストと選択リストの構文は同じで、次の例に示すように、キーワード `all`、または空白なしのコンマで区切った番号または番号の範囲 (*n-m*) のリストを指定できます。

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

実験番号は `experiment_list` コマンドを使用して特定できます。

次に選択の例をいくつか示します。

```
1:1-4+2:5,6
all:1,3-6
```

1 つ目の例では、実験 1 からオブジェクト 1 ～ 4、実験 2 からオブジェクト 5 ～ 6 を選択しています。2 つ目の例では、すべての実験からオブジェクト 1 と 3 ～ 6 を選択しています。オブジェクトは、LWP、スレッド、または標本のいずれかです。

選択用のコマンド

LWP、標本、CPU、およびスレッドを選択するためのコマンドは相互に依存しています。コマンドの実験リストの内容が、直前のコマンドのリストの内容と異なる場合は、最新のコマンドの実験リストの内容が、次のようにして 3 つのタイプの選択ターゲット (LWP、標本、スレッド) のすべてに適用されます。

- 最新の実験リストにない実験に対する既存の選択内容は無効になります。
- 最新の実験リストに含まれている実験に対する既存の選択内容は維持されます。
- 選択が行われていないターゲットに対しては `all` が適用されます。

sample_select *sample-spec*

情報を表示する標本を選択します。コマンドが終了すると、選択された標本が一覧表示されます。

lwp_select *lwp-spec*

情報を表示する LWP を選択します。コマンドが終了すると、選択された LWP が一覧表示されます。

thread_select *thread-spec*

情報を表示するスレッドを選択します。コマンドが終了すると、選択されたスレッドが一覧表示されます。

cpu_select *cpu-spec*

情報を表示する CPU を選択します。コマンドが終了すると、選択された CPU が一覧表示されます。

ロードオブジェクトの展開と短縮を制御するコマンド

これらのコマンドは、er_print ユーティリティーによるロードオブジェクトの表示方法を決定します。

object_list

すべてのロードオブジェクトのステータスと名前を示す 2 列のリストを表示します。最初の列には各ロードオブジェクトの表示/非表示/API のステータスが示され、2 番目の列にはオブジェクトの名前が示されます。各ロードオブジェクトの名前の前には、そのオブジェクトの関数が関数リストに表示される (展開される) ことを示す show、そのオブジェクトの関数が関数リストに表示されない (短縮される) ことを示す hide、またはロードオブジェクトへのエントリポイントを表す関数のみが表示される場合は API-only が付きます。短縮されたロードオブジェクトのすべての関数は、そのロードオブジェクト全体を表す関数リスト内の単一の項目へマップされます。

ロードオブジェクトリストの表示例を次に示します。

```
(er_print) object_list
Sel  Load Object
==== ==================
hide <Unknown>
show <Freeway>
show <libCstd_isa.so.1>
show <libnsl.so.1>
show <libmp.so.2>
show <libc.so.1>
show <libICE.so.6>
show <libSM.so.6>
show <libm.so.1>
show <libCstd.so.1>
show <libX11.so.4>
```

```
show <libXext.so.0>
show <libCrun.so.1>
show <libXt.so.4>
show <libXm.so.4>
show <libsocket.so.1>
show <libgen.so.1>
show <libcollector.so>
show <libc_psr.so.1>
show <ld.so.1>
show <liblayout.so.1>
```

object_show *object1,object2,...*

すべての名前付きロードオブジェクトを、それらのすべての関数が表示されるように設定します。オブジェクトの名前は、フルパス名またはベース名で指定できます。名前にコンマ文字が含まれている場合は、その名前を二重引用符で囲む必要があります。ロードオブジェクトの名前に「all」という文字列が使用されている場合は、すべてのロードオブジェクトの関数が表示されます。

object_hide *object1,object2,...*

すべての名前付きロードオブジェクトを、それらのすべての関数が表示されないように設定します。オブジェクトの名前は、フルパス名またはベース名で指定できます。名前にコンマ文字が含まれている場合は、その名前を二重引用符で囲む必要があります。ロードオブジェクトの名前に「all」という文字列が使用されている場合は、すべてのロードオブジェクトの関数が表示されます。

object_api *object1,object2,...*

すべての名前付きロードオブジェクトを、ライブラリへのエントリポイントを表す関数のみがすべて表示されるように設定します。オブジェクトの名前は、フルパス名またはベース名で指定できます。名前にコンマ文字が含まれている場合は、その名前を二重引用符で囲む必要があります。ロードオブジェクトの名前に「all」という文字列が使用されている場合は、すべてのロードオブジェクトの関数が表示されます。

objects_default

すべてのロードオブジェクトを `.er.rc` ファイルの処理からの初期デフォルト値に従って設定します。

object_select *object1,object2,...*

関数の情報を表示するロードオブジェクトを選択します。すべての名前付きロードオブジェクトの関数が表示され、その他すべてのロードオブジェクトの関数は表示されません。*object-list* は、空白なしのコンマで区切ったロードオブジェクトのリストです。ロードオブジェクトの関数が表示される場合、ゼロではないメトリックを保持するすべての関数が関数リストに表示されます。ロードオブジェクトの関数が表示されない場合、そのオブジェクトの関数は短縮され、そのロードオブジェクト全体のメトリックが入った単一行のみが表示され、個々の関数は表示されません。

ロードオブジェクト名は、フルパス名またはベース名で指定します。オブジェクト名そのものにコンマが含まれている場合は、名前を二重引用符で囲む必要があります。

メトリックを一覧するコマンド

ここでは、現在選択されているメトリックと使用可能なメトリックキーワードを一覧表示するコマンドを説明します。

metric_list

関数リストで現在選択されているメトリックと、関数リスト内のさまざまな種類のメトリックを参照する際にほかのコマンド (`metrics`、`sort` など) で使用可能なメトリックキーワードの一覧を表示します。

cmetric_list

現在選択されている呼び出し元 - 呼び出し先属性メトリック、および現在ソートに使用されているメトリックを表示します。

data_metric_list

現在選択されているデータ派生メトリック、およびすべてのデータ派生レポートのメトリックとキーワード名のリストを表示します。`metric_list` コマンドの出力と同じ方法でリストを表示しますが、データ派生フレーバーを持つメトリックと静的メトリックだけを含めます。

indx_metric_list

現在選択されているインデックスオブジェクトメトリック、およびすべてのインデックスオブジェクトレポートのメトリックとキーワード名のリストを表示します。metric_list コマンドと同じ方法でリストを表示しますが、排他的フレーバーを持つメトリックと静的メトリックだけを含めます。

出力を制御するコマンド

ここでは、er_print の出力を制御するコマンドを説明します。

outfile {*filename*|-|--}

開いている出力ファイルを閉じ、以降の出力先として *filename* を開きます。*filename* を開くときに、既存のコンテンツをクリアします。*filename* の代わりにハイフン (-) を指定した場合は、標準出力に出力されます。*filename* の代わりに 2 つのハイフン (--) を指定した場合は標準エラーに書き出されます。

appendfile *filename*

開いている出力ファイルを閉じ、*filename* を開きます。その際、既存のコンテンツがあれば残し、後続の出力がファイルの末尾に付加されるようにします。*filename* が存在しない場合、appendfile コマンドの機能は outfile コマンドと同じです。

limit *n*

すべての出力をレポートの最初の *n* 個のエントリに制限します。*n* は符号なしの整数です。*n* がゼロの場合、すべての制限を削除します。*n* が省略されている場合は、現在の制限を出力します。

name { long | short } [:{ *shared-object-name* **|** *no-shared-object-name* **}]**

長短どちらの形式の関数名を使用するかを指定します (C++ および Java のみ)。*shared-object-name* が指定されている場合は、関数名のあとに共有オブジェクト名を付加します。

viewmode { user| expert | machine }

モードを次のいずれかに設定します。

`user`
Java 実験の場合は、Java スレッドの Java 呼び出しスタックを表示し、ハウスキーピングスレッドを表示しません。関数リストには、Java 以外のスレッドからの集計時間を表す関数 `<JVM-System>` が含まれます。JVM ソフトウェアが Java 呼び出しスタックを報告しない場合、時間は関数 `<no Java callstack recorded>` に報告されます。

OpenMP 実験の場合は、OpenMP を使用せずにプログラムがコンパイルされたときに取得されたものと同様の、再構築された呼び出しスタックが表示されます。また、OpenMP ランタイムが特定の操作を実行しているときは、`<OMP-*>` という形式の名前を持つ特殊関数を追加します。

`expert`
Java 実験の場合は、ユーザーの Java コードの実行中には Java スレッドの Java 呼び出しスタックを表示し、JVM コードの実行中または JVM ソフトウェアが Java 呼び出しスタックを報告しないときにはマシン呼び出しスタックを表示します。ハウスキーピングスレッドについてはマシン呼び出しスタックを表示します。

OpenMP 実験の場合は、コンパイラによって生成された関数を表示します。これらの関数は、ユーザー関数によってユーザーモードで集計される、並列化されたループやタスクなどを表します。また、OpenMP ランタイムが特定の操作を実行しているときは、`<OMP-*>` という形式の名前を持つ特殊関数を追加します。OpenMP 実行時コード `libmtsk.so` からの関数は抑制されます。

`machine`
Java 実験と OpenMP 実験の場合は、すべてのスレッドの実際のネイティブ呼び出しスタックを表示します。

Java 実験と OpenMP 実験を除くすべての実験の場合は、3 つのモードすべてに同じデータを表示します。

compare { on | off | delta | ratio }

比較モードを、オフ (compare off、デフォルト)、オン (compare on)、デルタ (compare delta)、または比率 (compare ratio) に設定します。比較モードがオフである場合は、複数の実験が読み取られると、それらのデータは集約されます。比較が有効になっている場合は、複数の実験がロードされたときに、各実験のデータに関するメトリックの個別の列が表示されます。比較モードがデルタである場合、ベース実験には絶対メトリックが表示されますが、比較実験にはベースとの間の差分が表示されます。比較モードが比率である場合、比較実験にはベースとの間の比率が表示されます。

比較モードでは、それぞれの実験または実験グループを別の比較グループとして扱います。最初の実験または実験グループ引数が、ベースグループです。複数の実験を比較グループに含める場合は、実験グループファイルを作成して、er_print への単一の引数として使用する必要があります。

printmode *string*

string から出力モードを設定します。*string* が text である場合、出力は表形式で実行されます。*string* が 1 文字である場合、出力は、その 1 文字を区切り文字として区切られたリストとして実行されます。*string* が html である場合、出力は HTML 表の形式になります。その他の *string* はすべて無効であり、そのコマンドは無視されます。

printmode 設定は、functions、memobj、indxobj など、表を生成するコマンドに対してのみ使用されます。この設定は、source や disassembly などのほかの出力コマンドでは無視されます。

その他の情報を出力するコマンド

次の er_print サブコマンドは、実験についてのその他の情報を表示します。

header *exp-id*

指定した実験に関する説明情報を表示します。*exp-id* は、exp_list コマンドを使用して取得することができます。*exp-id* として all を指定するか、exp_id を省略した場合は、ロードされた実験すべての情報が表示されます。

エラーや警告が発生した場合には、各ヘッダーのあとに表示されます。各実験のヘッダーは、ハイフン (-) で区切られます。

実験ディレクトリに notes という名前のファイルがある場合は、このファイルの内容がヘッダー情報の先頭に付加されます。notes ファイルは、collect コマンドに -C "*comment*" 引数を付けて、手動で追加、編集、または指定できます。

exp-id はコマンド行では必要ですが、スクリプトや対話モードでは不要です。

ifreq

測定されたカウントデータから命令頻度のリストを書き込みます。命令頻度のレポートは、カウントデータからのみ生成できます。

objects

パフォーマンス解析の目的でロードオブジェクトを使用した結果として生じるエラーメッセージや警告メッセージのないロードオブジェクトを一覧表示します。表示されるロードオブジェクトの数は、limit コマンドを使用して制限できます (170 ページの「出力を制御するコマンド」を参照)。

overview *exp_id*

すべての実験にわたって集計されたすべてのデータの概要を書き込みます。ホットメトリックに値を強調表示するアスタリスクが付いている間は、関数リストメトリックに [X] が付いて示されます。

sample_detail [*exp_id*]

指定された実験に関する詳細な標本情報を書き込みます。*exp_id* は、experiment_list コマンドで指定されている実験の数値識別子です。*exp_id* が省略されているか、または all である場合は、すべての実験内のすべての標本の合計と統計情報を書き込みます。

現在 sample_detail で生成されるレポートは、以前のリリースでは overview コマンドで出力されていました。

statistics *exp_id*

指定した実験の現在の標本セット全体にわたって集計された実行統計情報を書き出します。実行統計値の定義と意味については、`getrusage`(3C) と `proc`(4) のマニュアルページを参照してください。実行統計には、コレクタがデータをまったく収集しないシステムスレッドからの統計が含まれます。

exp_id は、`experiment_list` コマンドを使用して取得することができます。*exp_id* が指定されていない場合、各実験の標本セットを対象に集計された、すべての実験のデータの合計が表示されます。*exp_id* が `all` である場合、各実験の合計と個々の統計が表示されます。

実験のためのコマンド

これらのコマンドは、スクリプトと対話型モードでのみ使用されます。コマンド行では許可されていません。

add_exp *exp_name*

指定された実験または実験グループを現在のセッションに追加します。

drop_exp *exp_name*

現在のセッションから指定された実験を削除します。

open_exp *exp_name*

ロードされているすべての実験をセッションから削除したあと、指定された実験または実験グループをロードします。

`.er.rc` ファイル内のデフォルトの設定

`er_print` ユーティリティーは、複数の場所から `.er.rc` というリソースファイル内の設定を読み取って、デフォルト値を判断します。ファイルは次の順序で読み取られます。

1. システムリソースファイル */Studio-installation-dir*/lib/analyzer/lib/er.rc
2. ユーザーの .er.rc リソースファイル (ユーザーのホームディレクトリに存在する場合)。
3. .er.rc リソースファイル (er_print コマンドの実行元の現在のディレクトリに存在する場合)。

各ファイルの設定は、その前に読み取られていたファイルの設定をオーバーライドします。ホームディレクトリの .er.rc ファイル内のデフォルト値はシステムのデフォルト値よりも優先され、現在のディレクトリの .er.rc ファイル内のデフォルト値は、ユーザーのホームおよびシステムのデフォルト値よりも優先されます。

ソースおよび逆アセンブリのコンパイラの解説に適用する .er.rc 内の設定は、er_src ユーティリティーによっても使用されます。

.er.rc ファイルで次のコマンドを使用して、er_print と er_src のデフォルト値を設定できます。これらのコマンドは、デフォルト値を設定するためにのみ使用でき、er_print ユーティリティーの入力には使用できません。

.er.rc デフォルト値ファイルは、すべての実験のデフォルト値を設定するためにホームディレクトリに置くか、デフォルト値をローカルに設定するためにそれ以外のディレクトリに置くことができます。er_print ユーティリティー、er_src ユーティリティー、またはパフォーマンスアナライザのいずれかを起動すると、現在のディレクトリとユーザーのホームディレクトリに .er.rc ファイルがあるかどうかが調べられます。これらのファイルが存在する場合は読み取られ、システムのデフォルト値ファイルも読み取られます。ホームディレクトリの .er.rc ファイル内のデフォルト値はシステムのデフォルト値よりも優先され、現在のディレクトリの .er.rc ファイル内のデフォルト値は、ユーザーのホームおよびシステムのデフォルト値よりも優先されます。

注記 - 実験が格納されているディレクトリからデフォルト値ファイルを読み取るには、そのディレクトリから er_print ユーティリティーを起動する必要があります。

この章ですでに説明したように、これらのファイルには、scc、sthresh、dcc、dthresh、addpath、pathmap、name、mobj_define、indxobj_define、object_show、object_hid および viewmode のコマンドが含まれます。また、コマンド行やスクリプトでは使用できない、次のコマンドも含めることができます。

dmetrics *metric-spec*

関数リストに表示または印刷するデフォルトのメトリックを指定します。メトリックリストの構文と使用方法については、143 ページの「メトリックリスト」で説明しています。リストでのメトリックキーワードの順序によって、メトリックが表示される順序が決まります。

呼び出し元 - 呼び出し先リストのデフォルトのメトリックは、このリスト内の各メトリック名の最初の名前の前に対応する属性メトリックを追加することによって得られます。

dsort *metric-spec*

関数リストの内容をソートするときの基準として、デフォルトで使用するメトリックを指定します。ソート基準メトリックは、このリスト内で、読み込まれている実験内のメトリックと一致する最初のメトリックです。このとき、次の条件が適用されます。

- *metric-spec* のエントリに表示文字列の感嘆符「!」が含まれている場合、表示されているかどうかに関係なく、一致する名前を持つメトリックの中で最初のメトリックが使用されます。
- *metric-spec* のエントリにほかの表示文字列が含まれている場合、一致する名前を持つメトリックの中の最初の表示メトリックが使用されます。

メトリックリストの構文と使用方法については、143 ページの「メトリックリスト」で説明しています。

呼び出し元 - 呼び出し先リストのデフォルトソート基準メトリックは、関数リストのデフォルトソート基準メトリックに対応する属性メトリックです。

en_desc { on | off | =*regexp***}**

派生実験を読み取るためのモードを `on` (すべての派生実験を有効にする) か `off` (すべての派生実験を無効にする) に設定します。=*regexp* を使用した場合、実行可能ファイル名が正規表現と一致する実験のデータが有効になります。デフォルト設定は `on` で、すべての派生を追跡します。

派生を含む実験の読み取りでは、パフォーマンスデータをほとんどまたはまったく含まないサブ実験はすべて、パフォーマンスアナライザや `er_print` によって無視されます。

その他のコマンド

次のコマンドにより、`er_print` ユーティリティーでさまざまなタスクが実行されます。

procstats

処理データから蓄積された統計を出力します。

script *filename*

filename に指定したスクリプトファイル内の追加コマンドを処理します。

version

現在の `er_print` ユーティリティーのバージョン情報を出力します。

quit

現在のスクリプトの処理を打ち切るか、対話モードを終了します。

exit

`quit` の別名。

help

`er_print` コマンドの一覧を表示します。

 # ...

コメント行。スクリプトまたは `.er.rc` ファイルで使用されます。

式の文法

フィルタを定義する式と、メモリーオブジェクトインデックスを算出するために使用される式には、共通の文法が使用されます。

文法では、式を演算子とオペランドまたはキーワードの組み合わせとして指定します。フィルタの場合は、式が真と評価されるとパケットが包含され、式が偽と評価されるとパケットが除外され

ます。メモリーオブジェクトまたはインデックスオブジェクトの場合、式は、パケット内で参照される特定のメモリーオブジェクトまたはインデックスオブジェクトを定義するインデックスへと評価されます。

式のオペランドにはラベル、定数、またはデータレコード内のフィールドのリストとして、`describe` コマンドで指定できます。オペランドには、`THRID`、`LWPID`、`CPUID`、`USTACK`、`XSTACK`、`MSTACK`、`LEAF`、`VIRTPC`、`PHYSPC`、`VADDR`、`PADDR`、`DOBJ`、`TSTAMP`、`SAMPLE`、`EXP` またはメモリーオブジェクトの名前が含まれます。オペランドの名前は大文字と小文字が区別されません。

`USTACK`、`XSTACK`、および `MSTACK` は、それぞれユーザー表示、上級表示、マシン表示の関数呼び出しスタックを表します。

`VIRTPC`、`PHYSPC`、`VADDR`、および `PADDR` は、ハードウェアカウンタプロファイリングまたは時間プロファイリングで「+」が指定された場合のみゼロ以外になります。さらに、`VADDR` は、実際の仮想アドレスが決定できなかった場合、256 未満になります。`VADDR` を決定できなかった場合、または、仮想アドレスを物理アドレスにマップできなかった場合、`PADDR` はゼロになります。同様に、バックトラッキングが失敗した場合、または要求されなかった場合、`VIRTPC` はゼロになり、`VIRTPC` がゼロか、`VIRTPC` が物理アドレスにマップできなかった場合、`PHYSPC` はゼロになります。

演算子は、C の表記法と C の優先順位規則に従った通常の論理演算子と算術 (シフトを含む) 演算子、要素が集合に含まれるかどうかを決める演算子 (`IN`)、要素集合の一部または全部が 1 つの集合に含まれるかどうかを決める演算子 (それぞれ、`SOME IN` か `IN`) のいずれかです。追加の演算子 `ORDERED IN` は、左側のオペランドの全要素が、右側のオペランドに同じ順序で現れているかどうかを判断します。`IN` 演算子は、左側のオペランドの全要素が右側のオペランドに現れることを要求しますが、順序については強制しません。

If-then-else 構造は、C のように `?` 演算子と `:` 演算子で指定されます。すべての式が正しく構文解析されるよう、括弧を使用してください。`er_print` のコマンド行では、複数の行にまたがって式を分割することはできません。スクリプト内またはコマンド行では、式に空白文字が含まれる場合、その式を二重引用符で囲む必要があります。

フィルタ式はブール値がパケットを包含する場合は真、除外する場合は偽と評価します。スレッド、CPU、実験 id、プロセス pid、および標本のフィルタリングは、適切なキーワードと 1 つの整数の間の関係式、または `IN` 演算子とコンマ区切りの整数リストを使用した関係式に基づきます。

時間フィルタリングを使用するには、`TSTAMP` と時間を結び付ける 1 つ以上の関係式を指定し、時間は、現在パケットが処理されている実験の開始以降のナノ秒数 (整数) で指定します。標本の時間を取得するには、`overview` コマンドを使用します。`overview` コマンドでは時間が秒単位で与えられるため、時間フィルタリングに使用するには、ナノ秒に変換する必要があります。時間は、パフォーマンスアナライザの「タイムライン」表示から取得することもできます。

関数フィルタリングは、リーフ関数に基づいて行うか、スタック内の任意の関数に基づいて行うことができます。リーフ関数によるフィルタリングは、`LEAF` キーワードと整数関数 ID の間の関係式、または `IN` 演算子と構造 `FNAME("` *regexp*`")` を使用した関係式を使用することで指定され

ます (*regexp* は regexp(5) のマニュアルページで説明されている正規表現)。*name* の現在の設定で指定されている関数の名前全体が一致する必要があります。

呼び出しスタック内の関数に基づいたフィルタリングは、構造 FNAME("*regexp*") 内の関数が、式 (FNAME("myfunc") SOME IN USTACK) を使用してキーワード USTACK で表現された関数配列に含まれるかどうかを判定することによって指定されます。FNAME は、スタックのマシン表示 (MSTACK) および上級表示 (XSTACK) を同じ方法でフィルタリングするためにも使用できます。

データオブジェクトのフィルタリングは、スタック関数のフィルタリングに似ていて、DOBJ キーワードと構造 DNAME("*regexp*") を括弧で囲んで使用します。

メモリーオブジェクトのフィルタリングを指定するには、mobj_list コマンドに示すようなメモリーオブジェクトの名前とオブジェクトの整数インデックス、または一連のオブジェクトのインデックスを使用します。(<Unknown> メモリーオブジェクトのインデックスは -1 です)。

インデックスオブジェクトのフィルタリングを指定するには、indxobj_list コマンドに示すようなインデックスオブジェクトの名前とオブジェクトの整数インデックス、または一連のオブジェクトのインデックスを使用します。(<Unknown> インデックスオブジェクトのインデックスは -1 です)。

データオブジェクトのフィルタリングとメモリーオブジェクトのフィルタリングは、データ領域データを持つハードウェアカウンタパケットについてのみ意味があり、ほかのすべてのパケットは、そのようなフィルタリングでは除外されます。

仮想アドレスまたは物理アドレスの直接フィルタリングを指定するには、VADDR または PADDR とアドレスの間の関係式を使用します。

メモリーオブジェクトの定義 (161 ページの「mobj_define *mobj-type index-exp*」を参照) では、VADDR キーワードまたは PADDR キーワードのどちらかを使用して整数インデックスに評価する式を使用します。それらの定義は、メモリーカウンタとデータ領域データについてのハードウェアカウンタパケットにのみ適用できます。この式は整数か、または <Unknown> メモリーオブジェクトの場合は -1 を返すべきです。

インデックスオブジェクトの定義 (160 ページの「indxobj_define *indxobj-type index-exp*」を参照) では、1 つの整数インデックスに評価する式が使用されます。この式は整数か、または <Unknown> インデックスオブジェクトの場合は -1 を返す必要があります。

フィルタ式の例

このセクションでは、er_print -filters コマンドとともに使用でき、「拡張カスタムフィルタ」ダイアログボックスで使用できるフィルタ式の例を示します。

er_print -filters コマンドでは、フィルタ式は、次の例のように単一引用符で囲まれます。

```
er_print -filters 'FNAME("myfunc") SOME IN USTACK' -functions test.1.er
```

例 4 名前とスタックによる関数のフィルタリング

ユーザー関数スタックから myfunc という名前の関数をフィルタリングするには、次のようにします。

```
FNAME("myfunc") SOME IN USTACK
```

例 5 スレッドと CPU によるイベントのフィルタリング

CPU 2 上でのみ実行したスレッド 1 からのイベントを表示するには、次のようにします。

```
THRID == 1 && CPUID == 2
```

例 6 インデックスオブジェクトによるイベントのフィルタリング

インデックスオブジェクト THRCPU が "CPUID<<16|THRID" として定義されている場合、次のフィルタは、CPU 2 上で実行したスレッド 1 からのイベントを表示する場合のフィルタと等価です。

```
THRCPU == 0x10002
```

例 7 指定の時間内に発生したイベントのフィルタリング

5 秒と 9 秒の間に発生した実験 2 のイベントをフィルタリングするには、次のようにします。

```
EXPID==2 && TSTAMP >= 5000000000 && TSTAMP < 9000000000
```

例 8 特定の Java クラスのイベントのフィルタリング

スタック内の特定の Java クラスのメソッドを持つイベントをフィルタリングするには、次のようにします。(ユーザー表示モードの場合)

```
FNAME("myClass.*") SOME IN USTACK
```

例 9 内部関数 ID と呼び出し順序によるイベントのフィルタリング

関数 ID が既知である (パフォーマンスアナライザに表示されている) 場合、マシン呼び出しスタック内の特定の呼び出し順序を含むイベントをフィルタリングするには、次のようにします。

```
(314,272) ORDERED IN MSTACK
```

例 10 状態または期間によるイベントのフィルタリング

describe コマンドにより、時間プロファイリング実験の次のプロパティーが一覧表示された場合は、

```
MSTATE    UINT32  Thread state
NTICK     UINT32  Duration
```

次のフィルタを使用して、特定の状態のイベントを選択できます。

```
MSTATE == 1
```

または、次のフィルタを使用して、特定の状態にあり、期間が 1 クロック刻みより長いイベントを選択できます。

```
MSTATE == 1 && NTICK > 1
```

er_print コマンドの例

ここでは、er_print コマンドの使用例を示します。

例 11 関数にかかる時間のサマリーを表示

```
er_print -functions test.1.er
```

例 12 呼び出し元-呼び出し先関係を表示

```
er_print -callers-callees test.1.er
```

例 13 ホットなソース行を表示

ソース行情報は、コードのコンパイルとリンクで -g が指定されていることを前提にしています。Fortran の関数とルーチンの場合は、関数名の最後に下線を付けてください。関数名のあとの 1 は、myfunction の複数インスタンスを区別するためのものです。

```
er_print -source  myfunction 1 test.1.er
```

例 14 ユーザー関数スタックから myfunc という名前の関数をフィルタリング

```
er_print -filters 'FNAME("myfunc") SOME IN USTACK' -functions test.1.er
```

例 15 gprof に似た出力を生成

次の例は、実験から gprof 形式に似た一覧を生成します。出力は er_print.out というファイルで、先頭の 100 個の関数と、関数ごとの属性ユーザー時間でソートされた呼び出し元-呼び出し先データの一覧です。

```
er_print -outfile  er_print.out -metrics e.%user -sort e.user \
```

```
-limit 100 -func -callers-callees test.1.er
```

この例のコマンドを分解して、次の独立した 2 つのコマンドにすることもできます。ただし、大規模な実験またはアプリケーションでは、er_print の呼び出しのたびに、かなりの時間がかかることがありますので注意してください。

```
er_print -metrics  e.%user -limit 100  -functions test.1.er

er_print -metrics  e.%user -callers-callees test.1.er
```

例 16 コンパイラ解説のみを表示

このコマンドを使用するためにプログラムを実行する必要はありません。

```
er_src -myfile.o
```

例 17 時計時間プロファイリングを使用して、関数および呼び出し元-呼び出し先を一覧表示

```
er_print -metrics  ei.%wall -functions test.1.er

er_print -metrics aei.%wall  -callers-callees test.1.er
```

例 18 er_print コマンドを含むスクリプトを実行

```
er_print -script myscriptfile test.1.er
```

myscriptfile スクリプトに er_print コマンドが含まれます。スクリプトファイルの例を次に示します。

```
## myscriptfile

## Send script output to standard output
outfile -

## Display descriptive information about the experiments
header

## Write out the sample data for all experiments
overview

## Write out execution statistics, aggregated over
## the current sample set for all experiments
statistics

## List functions
functions

## Display status and names of available load objects
object_list
```

```
## Write out annotated disassembly code for systime,
## to file disasm.out
outfile disasm.out
disasm systime


## Write out annotated source code for synprog.c
## to file source.out
outfile source.out
source synprog.c

## Terminate processing of the script
quit
```

♦ ♦ ♦ 第 6 章

パフォーマンスアナライザとそのデータについて

パフォーマンスアナライザは、コレクタが収集したイベントデータを読み取り、そのデータをパフォーマンスメトリックに変換します。メトリックは、ターゲットプログラムの構造内の、命令、ソース行、関数、ロードオブジェクトなどのさまざまな要素について計算されます。タイムスタンプ、スレッド ID、LWP ID、および CPU ID を含むヘッダーに加えて、各イベントについて次の 2 つの部分からなるデータが収集され、記録されます。

- メトリックの計算に使用されるイベント固有のデータ
- プログラム構造へのメトリックの関連付けに使用されるアプリケーションの呼び出しスタック

プログラム構造にメトリックを関連付ける処理は、常に簡単にできるとはかぎりません。これは、コンパイラによって、コードの挿入や変換、最適化が行われるためです。この章では、この処理を説明するとともに、パフォーマンスアナライザの表示にそのことがどのように反映されるのかという問題を取り上げます。

この章では、次の内容について説明します。

- 185 ページの「データ収集の動作」
- 189 ページの「パフォーマンスメトリックの解釈」
- 195 ページの「呼び出しスタックとプログラムの実行」
- 208 ページの「プログラム構造へのアドレスのマッピング」
- 216 ページの「インデックスオブジェクトへのパフォーマンスデータのマッピング」
- 216 ページの「メモリーオブジェクトへのパフォーマンスデータのマッピング」
- 217 ページの「プログラムデータオブジェクトへのデータアドレスのマッピング」

データ収集の動作

データ収集の実行による出力は実験であり、さまざまな内部ファイルとサブディレクトリを持つディレクトリとしてファイルシステム内に格納されます。

実験の形式

すべての実験には、次の 3 つのファイルが含まれています。

- ログファイル (`log.xml`)。収集されたデータの種類、各種コンポーネントのバージョン、ターゲットが存続している間の各種イベントのレコード、およびターゲットのワードサイズに関する情報が含まれている XML ファイル。
- マップファイル (`map.xml`)。ターゲットのアドレス空間にどのようなロードオブジェクトがロードされたかに関する時間依存の情報、およびそれらのロードオブジェクトがロードまたはアンロードされた時間を記録した XML ファイル。
- 概要ファイル。実験内のあらゆる標本ポイントで記録された使用情報を含むバイナリファイル。

また、実験にはプロセスが存続している間のプロファイルイベントを表すバイナリデータファイルがあります。189 ページの「パフォーマンスメトリックの解釈」で説明されているように、各データファイルには一連のイベントが含まれています。データの種類ごとに個別のファイルが使用されますが、各ファイルはターゲット内のすべてのスレッドで共有されます。

クロックプロファイリングまたはハードウェアカウンタオーバーフロープロファイリングの場合、データは、クロックティックまたはカウンタオーバーフローによって呼び出されたシグナルハンドラで書き込まれます。同期トレース、ヒープトレース、I/O トレース、MPI トレース、または OpenMP トレースの場合、データは、通常ユーザーによって呼び出されたルーチンで `LD_PRELOAD` 環境変数によって割り込まれた `libcollector` ルーチンから書き込まれます。そのような割り込み処理ルーチンは部分的にデータレコードを記入したあと、通常のユーザー呼び出しルーチンを呼び出し、ルーチンが復帰したときにデータレコードの残りの部分を記入し、データファイルにレコードを書き込みます。

すべてのデータファイルはメモリーマップされ、ブロック単位で書き込まれます。レコードは常に有効なレコード構造を持つように記入されるので、実験は書き込み中に読み取ることができます。このバッファー管理の方針は、スレッド間の競合や直列化を最小限に抑えるように設計されています。

オプションで、`notes` という名前の ASCII ファイルを実験に含めることができます。このファイルは、`collect` コマンドに `-C` *comment* 引数を使用すると、自動的に作成されます。実験の作成後、ファイルを手動で編集または作成できます。ファイルの内容は、実験のヘッダーの先頭に付加されます。

`archives` ディレクトリ

各実験には `archives` ディレクトリがあり、このディレクトリには、`map.xml` ファイル内で参照されている各ロードオブジェクトについて記述したバイナリファイルがあります。これらのファイルは、データ収集の終了時に実行される `er_archive` ユーティリティーによって作成されます。プロセスが異常終了すると、`er_archive` ユーティリティーが呼び出されないことがあります。その場合

は、er_print ユーティリティーまたはパフォーマンスアナライザが、その実験で最初に呼び出されたときにアーカイブファイルを書き込みます。

アーカイブディレクトリにはまた、実験をアーカイブするために使用されるオプションに応じて、共有オブジェクトまたはソースファイルのコピーを含めることもできます。

サブ実験

サブ実験は、派生プロセスをたどったり、MPI 実験を収集したり、ユーザープロセスでカーネルをプロファイルしたりする場合など、複数のプロセスがプロファイルされるときに作成されます。

派生プロセスは、その実験を親実験のディレクトリ内のサブディレクトリに書き込みます。これらの新しいサブ実験には、次のように、その系統を示す名前が付けられます。

- 作成者の実験名にアンダースコアが付加される。
- fork には f、exec には x、そのほかの派生実験には c のコード文字が追加される。Linux では、C は clone(2) によって生成された子孫に使用されます。
- コード文字のあとに、fork または exec のインデックスを示す数字が追加される。
- 実験接尾辞 .er を付加して実験名が完成する。

ユーザープロセスでは、親プロセスの実験名が test.1.er である場合、その 3 番目の fork によって作成された派生プロセスの実験は test.1.er/_f3.er になります。その派生プロセスが新しいイメージを実行した場合、対応する実験名は test.1.er/_f3_x1.er になります。派生実験は親の実験と同じファイルから構成されていますが、派生実験を持たず (すべての派生は親の実験内のサブディレクトリで表される)、アーカイブサブディレクトリを持っていません (すべてのアーカイブが親の実験内へ行われる)。

カーネル上の実験には、デフォルトでは test.1.er ではなく、ktest.1.er という名前が付けられます。データがユーザープロセスでも収集されるときは、カーネル実験には後続のユーザープロセスごとのサブ実験が含まれます。カーネルサブ実験には、_*process-name*_PID_*process-id*.1.er の形式を使用して名前が付けられます。たとえば、プロセス ID 1264 で実行されている sshd プロセス上で実行される実験には、ktest.1.er/_sshd_PID_1264.1.er という名前が付けられます。

MPI プログラムのデータはデフォルトでは test.1.er に収集され、MPI プロセスからのすべてのデータはサブ実験に収集されます (ランクごとに 1 つ)。コレクタは、MPI ランクを使用して、M_r*m*.er の形式のサブ実験名を構築します。ここで、*m* は MPI ランクです。たとえば、MPI ランク 1 の実験データは、test.1.er/M_r1.er ディレクトリ内に記録されます。

動的関数

ターゲットが動的関数を作成する実験では、map.xml ファイル内にこれらの関数を記述した追加レコードが存在します。追加ファイル dyntext には、動的関数の実際の命令のコピーが含まれています。動的な関数の注釈付き逆アセンブリを生成するには、このコピーが必要です。

Java 実験

Java 実験では、map.xml ファイル内には、その内部処理用に JVM ソフトウェアで作成された動的関数用と、ターゲット Java メソッドの動的にコンパイルされた (HotSpot) バージョン用の追加レコードがあります。

さらに、Java 実験には、ユーザーの呼び出されたすべての Java クラスに関する情報を含む JAVA_CLASSES ファイルが含まれています。

Java トレースデータは、libcollector.so の一部である JVMTI エージェントを使用して記録されます。エージェントは、記録されたトレースイベントへマップされるイベントを受け取ります。このエージェントはまた、JAVA_CLASSES ファイルの書き込みや、Java でコンパイルされたメソッドレコードの map.xml ファイルへの書き込みに使用される、クラスのロードや HotSpot コンパイルのためのイベントも受信します。

実験の記録

ユーザーモードのターゲット上の実験は、次の 3 種類の方法で記録できます。

- collect コマンド
- dbx によるプロセスの生成
- dbx による実行中のプロセスからの実験の作成

パフォーマンスアナライザの「アプリケーションのプロファイル」ダイアログでは、collect 実験を実行します。

collect 実験

collect コマンドを使用して実験を記録する場合、collect ユーティリティーは実験ディレクトリを作成して LD_PRELOAD 環境変数を設定することにより、libcollector.so やその他の libcollector モジュールが確実にターゲットのアドレス空間に事前にロードされるようにします。collect ユーティリティーは、その後、libcollector.so に実験名を知らせるための環境変数とデータ収集オプションを設定し、ターゲットをその上で実行します。

libcollector.so および関連モジュールが、すべての実験ファイルを書き込みます。

プロセスを作成する dbx 実験

データ収集を有効にした状態で dbx を使用してプロセスを起動すると、dbx は実験ディレクトリも作成し、libcollector.so が事前に読み込まれるようにします。dbx は最初の命令の前のブレークポイントでプロセスを停止し、次にデータ収集を開始するために libcollector.so 内の初期化ルーチンを呼び出します。

dbx はデバッグに Java Virtual Machine Debug Interface (JVMDI) エージェントを使用するため、dbx では Java 実験を収集できません。そのエージェントは、データ収集に必要な Java Virtual Machine Tools Interface (JVMTI) エージェントと共存できません。

実行中プロセス上の dbx 実験

dbx は実行中プロセスでの実験開始に使用すると実験ディレクトリを作成しますが、LD_PRELOAD 環境変数は使用できません。dbx はターゲットへの対話式関数呼び出しを行なって libcollector.so を開いたあと、プロセスを作成する場合と同様に libcollector.so の初期化ルーチンを呼び出します。データは、collect による実験の場合と同様に libcollector.so とそのモジュールによって書き込まれます。

このプロセスが開始されたときに libcollector.so はターゲットのアドレス空間に存在しなかったため、ユーザー呼び出し可能関数に対する割り込みに依存するデータ収集 (同期トレース、ヒープトレース、MPI トレース) はどれも機能しない可能性があります。一般に、シンボルは基盤となる関数にすでに解決されているため、割り込み処理は行えません。さらに、派生プロセスの次も割り込み処理に依存し、実行中プロセスで dbx により作成された実験に対して適切に機能しません。

dbx でプロセスを開始する前、または dbx を使用して実行中プロセスに接続する前に明示的に libcollector.so を事前ロードしている場合は、トレースデータを収集できます。

パフォーマンスメトリックの解釈

各イベントのデータには、高精度のタイムスタンプ、スレッド ID、および CPU ID が含まれています。これらを使用すると、パフォーマンスアナライザ内のメトリックを時間、スレッド、または CPU で絞り込むことができます。CPU ID については、getcpuid(2) のマニュアルページを参照してください。getcpuid を利用できないシステムでのプロセッサ ID は -1 であり、Unknown にマップされます。

各イベントでは、共通データ以外に、以降のセクションで説明する固有の raw データが生成されます。これらのセクションではまた、raw データから得られるメトリックの精度と、データ収集がメトリックに及ぼす影響についても説明しています。

クロックプロファイリング

クロックプロファイリングのイベント固有のデータは、プロファイリング間隔カウントの配列で構成されています。Oracle Solaris では、間隔カウンタが提供されます。プロファイル間隔の最後で適切な間隔カウンタが 1 増分され、別のプロファイル信号がスケジューリングされます。配列が記録およびリセットされるのは、Solaris スレッドが CPU ユーザーモードに入った場合だけです。配列のリセット時には、ユーザー CPU 状態の配列要素が 1 に設定され、ほかの全状

態の配列要素が 0 に設定されます。配列データが記録されるのは、配列がリセットされる前にユーザーモードに入るときです。そのため、この配列には、Solaris スレッドごとにカーネルによって保持されている 10 マイクロステートのそれぞれについて、前回ユーザーモードに入ってから開始されたマイクロステートごとのカウントの累計値が含まれます。Linux オペレーティングシステムでは、マイクロステートは存在しません。唯一の間隔カウンタは「ユーザー CPU 時間」です。

呼び出しスタックは、データと同時に記録されます。プロファイル間隔の最後に Solaris スレッドがユーザーモードになっていない場合は、そのスレッドが再びユーザーモードに入るまで、呼び出しスタックを変更できません。そのため、呼び出しスタックには、各プロファイリング間隔の最後のプログラムカウンタの位置が常に正確に記録されます。

Oracle Solaris で各マイクロステートが関連しているメトリックを表13「カーネルのマイクロステートとメトリックとの対応関係」に示します。

表 13 カーネルのマイクロステートとメトリックとの対応関係

カーネルのマイクロステート	説明	メトリック名
LMS_USER	ユーザーモードで動作	ユーザー CPU 時間
LMS_SYSTEM	システムコールまたはページフォルトで動作	システム CPU 時間
LMS_TRAP	上記以外のトラップで動作	システム CPU 時間
LMS_TFAULT	ユーザーテキストページフォルトでスリープ	テキストページフォルト時間
LMS_DFAULT	ユーザーデータページフォルトでスリープ	データページフォルト時間
LMS_KFAULT	カーネルページフォルトでスリープ	ほかの待ち時間
LMS_USER_LOCK	ユーザーモードロック待ちのスリープ	ユーザーロック時間
LMS_SLEEP	ほかの理由によるスリープ	ほかの待ち時間
LMS_STOPPED	停止 (/proc、ジョブ制御、lwp_stop のいずれか)	ほかの待ち時間
LMS_WAIT_CPU	CPU を待機中	CPU 待ち時間

タイミングメトリックの精度

タイミングデータは統計データとして収集されるため、あらゆる統計的な標本収集手法のすべての誤差の影響を受けます。プログラムの実行時間が非常に短い場合は、少数のプロファイルパケットしか記録されず、多くのリソースを消費するプログラム部分が、呼び出しスタックに反映されないことがあります。このため、目的の関数またはソース行について数百のプロファイルパケットを累積するのに十分な時間または十分な回数で、プログラムを実行するようにしてください。

統計的な標本収集の誤差のほかに、データの収集?関連付け方法、システムにおけるプログラムの実行の進み具合を原因とする誤差もあります。次に示す環境などでは、タイミングメトリックでデータに不正確さやひずみが生じる可能性があります。

- スレッドが作成されるとき、最初のプロファイルパケットが記録される前に費やされる時間はプロファイリング間隔より短くなりますが、プロファイリング間隔全体は最初のプロファイルパケット内に記録されるマイクロステートに従います。多数のスレッドが作成された場合、エラーはプロファイル間隔の何倍にもなることがあります。

- スレッドが破棄されたとき、一部の時間は、最後のプロファイルパケットが記録されたあとに費やされます。多数のスレッドが破棄された場合、エラーはプロファイル間隔の何倍にもなることがあります。
- プロファイル間隔中に、スレッドの再スケジュールが発生する場合があります。その結果、そのスレッドの記録された状態が、プロファイル間隔のほとんどが費やされたマイクロステートを表していない可能性があります。これらのエラーは、スレッドを実行するプロセッサより、実行されるスレッドの方が多い場合にさらに大きくなる可能性があります。
- プログラムがシステムクロックと相関関係を持つ形で動作することがあります。この場合は、スレッドが、費やされた時間のごく小さい部分を表しているような状態であると、プロファイリング間隔が常に期限切れになるため、プログラムの特定部分に対して記録された呼び出しスタックは過剰に表されます。マルチプロセッサシステムでは、プロファイリングシグナルによって相互関係が引き起こされる場合があります。プログラムのスレッドを実行中にプロファイリングシグナルによって中断されたプロセッサは、マイクロステートが記録されるときに Trap-CPU マイクロステートにある可能性があります。
- カーネルは、プロファイル間隔の時間切れになったときにマイクロステート値を記録します。システムが過負荷状態の場合、この値に、プロセスの本当の状態が反映されないことがあります。Oracle Solaris では、この状況によって、Trap-CPU または Wait-CPU マイクロステートの過剰アカウンティングが発生する可能性があります。
- システムクロックが外部ソースと同期されている場合、プロファイルパケット内に記録されるタイムスタンプにはプロファイリング間隔が反映されませんが、クロックに対して行われた調整はすべて含まれます。システムクロック調整の結果、プロファイルパケットが失われたかのように見える可能性があります。その時間は通常数秒間であり、調整は一定の増分単位で行われます。
- 動作クロック周波数が動的に変わるマシンで記録された実験には、プロファイリングの不正確さが反映されていることがあります。

今説明した不正確さに加えて、データ収集のプロセスによってタイミングメトリックにひずみが生じます。記録はプロファイリングシグナルによって開始されるため、プロファイルパケットの記録に費やされた時間がプログラムのメトリックに現れることはありません。これは、相関関係の別の例です。記録に費やされたユーザー CPU 時間は、記録されるあらゆるマイクロステート値に配分されます。この結果、ユーザー CPU 時間のメトリックが実際より小さくなり、その他のメトリックが実際より大きくなります。デフォルトのプロファイル間隔の場合、一般に、データの記録に費やされる時間は CPU 時間の 2、3% 未満です。

タイミングメトリックの比較

時間ベースの実験のプロファイリングで得られたタイミングメトリックと、その他の方法で得られた時間を比較する場合は、次の点に注意する必要があります。

シングルスレッドアプリケーションの場合、1 つのプロセスについて記録されたスレッド合計時間は通常、同じプロセスについて `gethrtime`(3C) によって返された値と比較して数十分の 1 パーセントまで正確です。CPU 時間の場合は、同じプロセスについて `gethrvtime`(3C) によって返される値と比較して、数パーセント程度異なることがあります。負荷が大きい場合は、差がさらに大きくなることがあります。ただし、CPU 時間の差は体系的なひずみを表しません。さまざまな関数やソース行などについて報告される相対的な時間に大きなひずみは生じません。

パフォーマンスアナライザの報告するスレッド時間は、vmstat が CPU 全体にまたがって集計した時間を報告するため、vmstat の報告する時間とかなり異なることがあります。たとえば、ターゲットプロセスの LWP 数が、そのプロセスが動作するシステムの CPU 数よりも多い場合、アナライザは、vmstat が報告する時間よりもずっと長い待ち時間を報告します。

パフォーマンスアナライザの「統計」ビューや er_print 統計の表示に現れるマイクロステートタイミングは、プロセスファイルシステムの /proc 使用報告に基づいており、マイクロステートで費やされた時間が高い精度で記録されます。詳細は、proc(4) のマニュアルページを参照してください。これらのタイミングを、全体としてのプログラムを表す <Total> 関数のメトリックと比較することによって、集約されたタイミングメトリックのおよその精度を知ることができます。ただし、「統計」ビューに表示される値には、<Total> のタイミングメトリック値には含まれないその他の関連要素が含まれることがあります。その原因は、データ収集が一時停止される期間によるものです。

ユーザー CPU 時間とハードウェアカウンタサイクル時間は異なります。なぜなら、ハードウェアカウンタは、CPU モードがシステムモードへ切り替えられたときにオフにされるからです。詳細は、197 ページの「トラップ」を参照してください。

ハードウェアカウンタオーバーフローのプロファイリング

ハードウェアカウンタオーバーフロープロファイリングデータには、カウンタ ID とオーバーフロー値が含まれます。この値は、カウンタがオーバーフローするように設定されている値よりも大きくなることがあります。これは、オーバーフローが発生して、そのイベントが記録されるまでの間に命令が実行されるためです。この値は特に、浮動小数点演算やキャッシュミスなどのカウンタよりも、ずっと頻繁に増分されるサイクルカウンタや命令カウンタの場合に大きくなる可能性があります。イベント記録時の遅延はまた、呼び出しスタックとともに記録されたプログラムカウンタのアドレスが正確にオーバーフローイベントに対応しないことを意味します。詳細は、232 ページの「ハードウェアカウンタオーバーフローの関連付け」を参照してください。また、197 ページの「トラップ」も参照してください。トラップおよびトラップハンドラは、ユーザーの CPU 時間とサイクルカウンタによって報告される時間の間の、大きな相違の原因になることがあります。

動作クロック周波数が動的に変わるマシンで記録される実験では、サイクルベースのカウントから時間への変換で不正確さが生じることがあります。

収集されるデータ量は、オーバーフロー値に依存します。選択した値が小さすぎると、次のような影響が出ることがあります。

- データの収集に費やされる時間が、プログラムの実行時間のかなりの部分を占めることがあります。収集実行では、プログラムの実行ではなく、オーバーフローの処理とデータの書き込みに時間のかなりが費やされる場合があります。
- カウントのかなりの部分の原因がデータ収集であることがあります。こうしたカウントは、コレクタ関数 collector_record_counters によるものです。この関数のカウントが大きい場合は、オーバーフロー値が小さすぎます。

- データ収集によってプログラムの動作が変わることがあります。たとえば、キャッシュミスのデータの収集では、キャッシュミスの大半が、コレクタの命令のフラッシュとキャッシュからのデータのプロファイリング、プログラム命令とデータとの置き換えによるものです。プログラムで多くのキャッシュミスが発生するように見えますが、データ収集を行わなければ、キャッシュミスが実際にはほとんど発生しなかった可能性があります。

データ領域プロファイリングとメモリー領域プロファイリング

メモリー領域プロファイルは、キャッシュミスなどのメモリー関連イベントが、キャッシュライン、メモリーバンク、ページなどの、マシンの物理的な構造に対して報告されるプロファイルです。

データ領域プロファイルは、これらのメモリー関連イベントが、メモリー関連イベントが発生した命令だけではなく、そのイベントの原因となる参照を行なったデータ構造に対して報告されるプロファイルです。データ領域プロファイリングは、Oracle Solaris を実行している SPARC システム上でのみ使用できます。Oracle Solaris または Linux を実行している x86 システム上ではまだ使用できません。

メモリー領域またはデータ領域のいずれかのプロファイリングの場合、収集されるデータは、メモリーベースのカウンタを使用するハードウェアカウンタである必要があります。正確なカウンタの場合は、SPARC または x86 Oracle Solaris プラットフォームのどちらでも、デフォルトではメモリー領域およびデータ領域データが収集されます。

データ領域プロファイリングをサポートするには、実行可能ファイルを `-xhwcprof` フラグでコンパイルするようにしてください。このフラグは C、C++、および Fortran コンパイラでのコンパイルに適用できますが、SPARC プラットフォームでのみ意味があります。このフラグはほかのプラットフォームでは無視されます。実行可能ファイルが `-xhwcprof` でコンパイルされない場合、`er_print` の `data_layout`、`data_single`、および `data_objects` コマンドではデータが表示されません。メモリー領域プロファイリングでは、正確なカウンタのために `-xhwcprof` は必要ありません。

実験にデータ領域またはメモリー領域のプロファイルが含まれている場合、`er_print` ユーティリティーでは、メモリーオブジェクト関連の各種コマンドのほかに、3 つの追加コマンド `data_objects`、`data_single`、および `data_layout` を使用できます。詳細は、158 ページの「データ領域リストを制御するコマンド」を参照してください。

さらに、パフォーマンスアナライザにはデータ領域プロファイリングに関連した 2 つのビューや、メモリーオブジェクト用の各種タブが用意されています。116 ページの「「データオブジェクト」ビュー」、115 ページの「「データレイアウト」ビュー」、および 115 ページの「「メモリーオブジェクト」ビュー」を参照してください。

ほかの引数なしで `collect -h` を実行すると、ハードウェアカウンタが一覧表示され、それらが、ロード関係なのか、ストア関係なのか、ロード-ストア関係なのか、および高精度かどうかが示されます。26 ページの「ハードウェアカウンタプロファイリングデータ」を参照してください。

同期待ちトレース

コレクタは、スレッドライブラリ `libthread.so` 内の関数の呼び出し、またはリアルタイム拡張ライブラリ `librt.so` の呼び出しをトレースすることによって、同期遅延イベントを収集します。イベント固有のデータは、要求と許可 (トレース対象の呼び出しの開始と終了) の高分解能のタイムスタンプと、同期オブジェクト (要求されている相互排他ロックなど) のアドレスで構成されます。スレッド ID と LWP ID は、データが記録された時点での ID です。待ち時間は、要求時間と許可時間の時間差です。記録されるイベントは、指定したしきい値を要求と許可の時間差が超えたものだけです。同期待ちトレースデータは、許可時に実験ファイルに記録されます。

遅延の原因となったイベントが完了しないかぎり、待ちスレッドがほかの作業を行うことはできません。この待ち時間は、「同期待ち時間」と「ユーザーロック時間」の両方に反映されます。同期遅延しきい値は短時間の遅延を排除するので、「ユーザーロック時間」が「同期待ち時間」よりも大きくなる可能性があります。

待ち時間は、データ収集のオーバーヘッドによってひずみます。そして、このオーバーヘッドは、収集されたイベントの個数に比例します。オーバーヘッドに費やされる待ち時間の一部は、イベント記録のしきい値を大きくすることによって最小化できます。

ヒープトレース

コレクタは、メモリーの割り当てと割り当て解除の関数である `malloc`、`realloc`、`memalign`、および `free` に割り込むことによって、これらの関数の呼び出しに関するトレースデータを記録します。メモリーを割り当てるときにこれらの関数を迂回するプログラムの場合、トレースデータは記録されません。別のメカニズムが使用されている Java メモリー管理では、トレースデータは記録されません。

トレース対象の関数は、さまざまなライブラリから読み込まれる可能性があります。パフォーマンスアナライザで表示されるデータは、特定の関数がロードされた元のライブラリに依存することがあります。

短時間で大量のトレース対象関数を呼び出すプログラムの場合、プログラムの実行に要する時間が大幅に長くなることがあります。延びた時間は、トレースデータの記録に使用されます。

I/O トレース

コレクタは、標準の I/O ルーチンの呼び出しおよびすべての I/O システムコールに関するトレースデータを記録します。

MPI トレース

MPI トレースは、VampirTrace データコレクタの修正版をベースにしています。詳細は、Technische Universitat Dresden の Web サイトにある『VampirTrace User Manual』を検索してください。

呼び出しスタックとプログラムの実行

*呼び出しスタック*は、プログラム内の命令を示す一連のプログラムカウンタ (PC) アドレスです。*リーフ PC* と呼ばれる最初の PC はスタックのいちばん下に位置し、次に実行される命令のアドレスを表します。次の PC は、リーフ PC を含む関数の呼び出しのアドレスです。次の PC はその関数の呼び出しのアドレスであり、このような関係がスタックの先頭に達するまで続きます。こうしたアドレスはそれぞれ、復帰アドレスと呼びます。呼び出しスタックの記録プロセスでは、プログラムスタックから復帰アドレスが取得されます。これは、*スタックの展開*と呼ばれています。展開の失敗については、207 ページの「不完全なスタック展開」を参照してください。

呼び出しスタック内のリーフ PC は、この PC が存在する関数にパフォーマンスデータの排他的メトリックを割り当てるときに使用されます。スタック上の各 PC は、リーフ PC も含めて、その PC が存在する関数に包括的メトリックを割り当てるために使用されます。

ほとんどの場合、記録された呼び出しスタック内の PC は、プログラムのソースコードに現れる関数に自然な形で対応しており、パフォーマンスアナライザが報告するメトリックもそれらの関数に直接対応しています。しかし、プログラムの実際の実行は、単純で直観的なプログラム実行モデルと対応しないことがあり、その場合は、パフォーマンスアナライザの報告するメトリックが混乱を招くことがあります。こうした事例については、208 ページの「プログラム構造へのアドレスのマッピング」を参照してください。

シングルスレッド実行と関数呼び出し

プログラムの実行でもっとも単純なものは、シングルスレッドプログラムが専用のロードオブジェクト内の関数を呼び出す場合です。

プログラムがメモリーに読み込まれて実行が開始されると、初期実行アドレス、初期レジスタセット、スタック (スクラッチデータの格納および関数の相互の呼び出し方法の記録に使用されるメモリー領域) からなるコンテキストが作成されます。初期アドレスは常に、あらゆる実行可能ファイルに組み込まれる `_start()` 関数の先頭位置になります。

プログラムを実行すると、たとえば関数呼び出しや条件文を表すことがある分岐命令があるまで、命令が順実行されます。分岐点では、分岐先が示すアドレスに制御が渡されて、そこから実行が続行されます。(SPARC では通常、分岐の次の命令は実行するようすでにコミットされており、この命令は分岐遅延スロット命令と呼ばれます。ただし、分岐命令には、この分岐遅延スロット命令の実行を無効にするものもあります。

呼び出しを表す命令シーケンスが実行されると、復帰アドレスがレジスタに書き込まれ、呼び出された関数の最初の命令から実行が続行されます。

ほとんどの場合は、この呼び出し先の関数に含まれる最初の数個の命令のどこかで、新しいフレーム (関数に関する情報を格納するためのメモリー領域) がスタックにプッシュされ、そのフレームに復帰アドレスが格納されます。復帰アドレスに使用されるレジスタは、呼び出された関数がほかの関数を呼び出すときに使用できます。関数から制御が戻されようとすると、スタックからフレームがポップされ、関数の呼び出し元のアドレスに制御が戻されます。

共有オブジェクト間の関数の呼び出し

共有オブジェクト内の関数が別の共有オブジェクトの関数を呼び出す場合は、同じプログラム内の単純な関数の呼び出しよりも実行が複雑になります。各共有オブジェクトには、プログラムリンケージテーブル (PLT) が 1 つあり、ここにはそのオブジェクトが参照し、そのオブジェクトの外部にあるすべての関数 (外部関数) のエントリが含まれます。最初は、PLT 内の各外部関数のアドレスは、実際には動的リンカーである `ld.so` 内のアドレスです。外部関数が初めて呼び出されると、制御が動的リンカーに移り、動的リンカーは、その外部関数への呼び出しを解決し、以降の呼び出しのために、PLT のアドレスにパッチを適用します。

3 つの PLT 命令のいずれか 1 つを実行しているときにプロファイリングイベントが発生した場合、PLT PC は削除され、排他的時間はその呼び出し命令に対応することになります。PLT エントリによる最初の呼び出し時にプロファイリングイベントが発生し、かつリーフ PC が PLT 命令のいずれかではない場合、PLT および `ld.so` のコードから発生する PC はすべて、包括的時間を集計する擬似的な関数 `@plt` に帰属します。各共有オブジェクトには、こういった擬似的な関数が 1 つ用意されています。`LD_AUDIT` インタフェースを使用しているプログラムの場合、PLT エントリが絶対にパッチされない可能性があるとともに、`@plt` の非リーフ PC の発生頻度が高くなることが考えられます。

シグナル

シグナルがプロセスに送信されると、各種のレジスタ操作とスタック操作が発生し、シグナル送信時にリーフ PC がシステム関数 `sigacthandler()` の呼び出しの復帰アドレスを示していたかのように見えます。`sigacthandler()` は、関数が別の関数を呼び出すのと同じようにして、ユーザー指定のシグナルハンドラを呼び出します。

パフォーマンスアナライザは、シグナル送信で発生したフレームを通常のフレームとして処理します。シグナル送信時のユーザーコードがシステム関数 `sigacthandler()` の呼び出し元として表示され、`sigacthandler()` はユーザーのシグナルハンドラの呼び出し元として表示されます。`sigacthandler()` とあらゆるユーザーシグナルハンドラ、さらにはそれらが呼び出すほかの関数の包括的メトリックは、割り込まれた関数の包括的メトリックとして表示されます。

コレクタは `sigaction()` に割り込むことによって、時間データ収集時にはそのハンドラが `SIGPROF` シグナルのプライマリハンドラになり、ハードウェアカウンタオーバーフローのデータ収集時には `SIGEMT` シグナルのプライマリハンドラになるようにします。

トラップ

トラップは命令またはハードウェアによって発行され、トラップハンドラによって捕捉されます。システムトラップは、命令から発行され、カーネルにトラップされるトラップです。すべてのシステムコールは、トラップ命令を使用して実装されます。ハードウェアトラップの例としては、命令を完了できないとき、あるいは命令がハードウェアに実装されていないときに、浮動小数点ユニットから発行されるものがあります。

トラップが発行されると、カーネルはシステムモードになります。Oracle Solaris 上では、マイクロステートは通常、ユーザー CPU 状態からトラップ状態、そしてシステム状態に切り替わります。マイクロステートの切り替わりポイントによっては、トラップの処理に費やされた時間が、システム CPU 時間とユーザー CPU 時間を合計したものとして現れることがあります。この時間は、ユーザーコード内でそのトラップを発行した命令、またはシステムコールに割り当てられます。

一部のシステムコールでは、呼び出しをできるだけ効率的に処理することが重要であるとみなされます。こうした呼び出しによって生成されたトラップを*高速トラップ*と呼びます。高速トラップを生成するシステム関数には、`gethrtime` および `gethrvtime` があります。これらの関数ではオーバーヘッドを伴うため、マイクロステートは切り替わりません。

このほかにも、トラップをできるだけ効率的に処理することが重要であるとみなされる場合があります。たとえば、TLB (トランスレーションルックアサイドバッファー) ミスやレジスタウィンドウのスピル/フィルの場合も、マイクロステートは切り替わりません。

TLB (translation lookaside buffer) ミス いずれの場合も、費やされた時間はユーザー CPU 時間として記録されます。ただし、システムモードに CPU モードが切り替えられたため、ハードウェアカウンタは動作していません。このため、これらのトラップの処理に費やされた時間は、なるべく同じ実験で記録された、ユーザー CPU 時間とサイクル時間の差を取ることで求めることができます。

トラップハンドラがユーザーモードに戻るケースもあります。Fortran で 4 バイトの境界に整列された整数に対し、8 バイト整数のメモリー参照を行うようなトラップです。そのトラップハンドラのフレームがスタックに現れるため、整数のロードまたはストア命令に起因するそのハンドラの呼び出しがパフォーマンスアナライザに表示される場合があります。

命令がカーネルにトラップされると、そのトラップ命令のあとの命令の実行に長い時間がかかっているようにみえます。これは、カーネルがトラップ命令の実行を完了するまで、その命令の実行を開始できないためです。

末尾呼び出しの最適化

特定の関数で最後に実行することが別の関数の呼び出しである場合、コンパイラは常に、ある特定の最適化を実行できます。新しいフレームを生成するのではなく、呼び出し先が呼び出し元のフレームを再利用し、呼び出し先のための復帰アドレスが呼び出し元からコピーされます。この最適化の目的は、スタックのサイズ削減、および SPARC プラットフォーム上でのレジスタウィンドウの使用削減にあります。

プログラムのソースの呼び出しシーケンスが、次のようになっていると仮定します。

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

B と C の末尾呼び出しが最適化されている場合、呼び出しスタックは、関数 A が関数 B、C、および D を直接呼び出しているかのように見えます。

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

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した場合、末尾呼び出しの最適化は、4 以上のレベルでのみ行われます。-g オプションなしでコードをコンパイルした場合、2 以上のレベルで末尾呼び出しの最適化が行われます。

明示的なマルチスレッド化

単純なプログラムは、シングルスレッド内で実行されます。マルチスレッド化された実行可能ファイルはスレッド作成関数を呼び出し、その関数に実行されるターゲット関数が渡されます。ターゲットの終了時にスレッドが破棄されます。

Oracle Solaris では、Solaris スレッドと POSIX スレッド (Pthread) の 2 種類のスレッド実装がサポートされています。Oracle Solaris 10 以降、両方のスレッド実装が libc.so に含まれます。

Solaris スレッドでは、新しく作成されたスレッドは、スレッド作成呼び出しで渡された関数を呼び出す _thread_start() と呼ばれる関数で実行を開始します。このスレッドによって実行されるターゲットが関係するどの呼び出しスタックでも、スタックの先頭は _thread_start() であり、スレッド作成関数の呼び出し元に接続することはありません。そのため、作成されたスレッドに関連付けられた包括的メトリックは、_thread_start() および <Total> 関数までしか伝播されません。スレッドの作成に加えて、Solaris のスレッド実装では、スレッドを実行するために LWP が Oracle Solaris 上に作成されます。LWPSolaris スレッドによる作成スレッドはそれぞれ特定の LWP に結合されます。

Oracle Solaris と Linux では、明示的なマルチスレッド化に Pthread を使用できます。

どちらの環境でも、新しいスレッドを作成するには、アプリケーションが Pthread API 関数 pthread_create() を呼び出して、関数引数の 1 つとして、ポインタをアプリケーション定義の起動ルーチンに渡します。

Oracle Solaris 10 より前の Oracle Solaris バージョンでは、新しい pthread の実行の開始時に _lwp_start() 関数が呼び出されます。Oracle Solaris 10 以降では、_lwp_start() から中間関数 _thrp_setup() が呼び出され、そこから pthread_create() で指定されたアプリケーション定義の起動ルーチンが呼び出されます。

Linux オペレーティングシステムでは、新しい pthread の実行の開始時に Linux 固有のシステム関数 clone() が呼び出され、そこから別の内部初期化関数 pthread_start_thread() が呼び出され、そこから、pthread_create() で指定されたアプリケーション定義の起動

ルーチンが呼び出されます。コレクタで使用できる Linux メトリック収集関数はスレッドに固有です。そのため、`collect` ユーティリティーが実行されると、このユーティリティーは、`pthread_start_thread()` とアプリケーション定義のスレッド起動ルーチンの間に `collector_root()` という名前のメトリック収集関数を割り込ませます。

Java テクノロジーベースのソフトウェア実行の概要

典型的な開発者にとっては、Java テクロノジーベースのアプリケーションはほかのプログラムと同じように動作します。このアプリケーションは、一般に `class.main` というメインエントリポイントから始まり、C または C++ アプリケーションの場合と同様に、ほかのメソッドを呼び出すことがあります。

オペレーティングシステムにとっては、Java プログラミング言語で書かれたアプリケーション (純粋なものか、C/C++ が混合しているもの) は JVM ソフトウェアをインスタンス化するプロセスとして動作します。JVM ソフトウェアは C++ ソースからコンパイルされ、`_start` から実行を開始し、それが `main` を呼び出すというように処理が進行します。このソフトウェアは `..class` ファイルまたは `.jar` ファイル、あるいはその両方からバイトコードを読み取り、そのプログラムで指定された操作を実行します。指定できる操作の中には、ネイティブ共有オブジェクトの動的な読み込みや、そのオブジェクト内に含まれている各種関数やメソッドへの呼び出しがあります。

JVM ソフトウェアは、従来の言語で記述されたアプリケーションでは一般に行われない多数の動作を行います。このソフトウェアは、起動時に、動的に生成されたコードの多数の領域をそのデータ領域内に作成します。これらの領域の 1 つは、アプリケーションのバイトコードメソッドを処理するために使用される実際のインタプリタコードです。

Java テクノロジベースのアプリケーションの実行中、ほとんどのメソッドは JVM ソフトウェアによって解釈されます。これらのメソッドは、*解釈されたメソッド*と呼ばれます。Java HotSpot 仮想マシンによって、バイトコードの解析時にパフォーマンスがモニターされ、頻繁に実行されているメソッドが検出されます。繰り返し実行されているメソッドは、Java HotSpot 仮想マシンによってコンパイルされ、マシンコードが生成される場合があります。結果となるメソッドは、*コンパイルされたメソッド*と呼ばれます。仮想マシンはそれ以降、そのメソッドの元のバイトコードを解釈するのではなく、より効率的なコンパイルされたメソッドを実行します。コンパイルされたメソッドはアプリケーションのデータ領域に読み込まれ、その後のある時点で読み込み解除することができます。さらに、インタプリタされたコードとコンパイルされたコードの間の変換を行うために、ほかのコードがデータ領域で生成されます。

Java プログラミング言語で記述されたコードはまた、ネイティブでコンパイルされたコード (C、C++、Fortran のいずれか) を直接呼び出すこともあります。このような呼び出しのターゲットは、ネイティブメソッドと呼ばれます。

Java プログラミング言語で記述されたアプリケーションは本質的にマルチスレッド型で、ユーザーのプログラム内でスレッドごとに 1 つの JVM ソフトウェアスレッドがあります。Java アプリケーションはまた、シグナル処理、メモリー管理、Java HotSpot 仮想マシンのコンパイルに使用されるハウスキーピングスレッドもいくつかあります。

データの収集は、J2SE の JVMTI にあるさまざまなメソッドを使用して実装されます。

Java 呼び出しスタックとマシン呼び出しスタック

パフォーマンスツールは、各スレッドの存続期間中にイベントを記録するほか、イベント時に呼び出しスタックを記録することによってデータを収集します。任意のアプリケーションの実行の任意の時点で、呼び出しスタックは、プログラムが実行のどの段階まで、またどのように達したかを表します。混合モデルの Java アプリケーションが従来の C、C++、および Fortran アプリケーションと異なる 1 つの重要な点は、ターゲットの実行中のどの時点でも、Java 呼び出しスタックとマシン呼び出しスタックという 2 つの呼び出しスタックが意味を持つことです。どちらの呼び出しスタックもプロファイリング中に記録され、解析中に調整されます。

クロックプロファイリングとハードウェアカウンタオーバーフロープロファイリング

Java プログラムに対するクロックプロファイリングとハードウェアカウンタオーバーフロープロファイリングは、Java 呼び出しスタックとマシン呼び出しスタックの両方が収集される点を除き、C、C++、および Fortran プログラムの場合と同様に機能します。

Java プロファイリングの表示モード

パフォーマンスアナライザには、Java プログラミング言語で記述されたアプリケーションのパフォーマンスデータを表示するための表示モードとして、ユーザーモード、上級モード、マシンモードの 3 つが用意されています。データがユーザーモードをサポートしている場合、デフォルトではユーザーモードが示されます。以降では、これらの 3 つの表示モードの主な違いをまとめます。

Java プロファイリングデータのユーザー表示モード

ユーザーモードでは、コンパイルされた Java メソッドや解釈された Java メソッドが名前で表示され、ネイティブメソッドがその自然な形式で表示されます。実行中は、解釈されたバージョンと 1 つまたは複数のコンパイルされたバージョンなど、特定の Java メソッドに対して多数のインスタンスが実行される可能性があります。ユーザーモードでは、すべてのメソッドが 1 つのメソッドとして集約されて表示されます。この表示モードは、パフォーマンスアナライザではデフォルトで選択されます。

ユーザー表示モードにある Java メソッドの PC は、メソッド ID とそのメソッドへのバイトコードのインデックスに対応し、ネイティブ関数の PC はマシン PC に対応します。Java スレッドの呼び出しスタックには、Java PC とマシン PC が混在している場合があります。この呼び出しスタックには、Java ユーザー表現を持たない Java ハウスキーピングコードに対応するフレームはありません。状況によっては、JVM ソフトウェアは Java スタックを展開することができず、`<no Java callstack recorded>` という特別な関数を持つシングルフレームが返されます。これは通常、合計時間の 5 ～ 10% にしかなりません。

ユーザーモードでの「関数」ビューには、Java メソッドおよび呼び出されたすべてのネイティブメソッドに対するメトリックが表示されます。「呼び出し元-呼び出し先」ビューには、呼び出しの関係がユーザーモードで示されます。

Java メソッドのソースは、コンパイル元の `.java` ファイル内のソースコードに対応し、各ソース行にメトリックがあります。Java メソッドの逆アセンブリは生成されたバイトコードのほか、各バイトコードに対するメトリックとインタリーブされた Java ソース (入手可能な場合) を示します。

Java ユーザー表現のタイムラインは、Java スレッドのみを示します。各スレッドの呼び出しスタックが、その Java メソッドとともに示されます。

Java ユーザー表現のデータ領域プロファイリングは、現在サポートされていません。

Java プロファイリングデータの上級表示モード

上級モードはユーザーモードに似ていますが、ユーザーモードでは表示されない JVM 内部要素の詳細のいくつかが、上級モードでは表示されます。上級モードでは、「タイムライン」にはすべてのスレッドが表示されます。ハウスキーピングスレッドの呼び出しスタックは、ネイティブ呼び出しスタックです。

Java プロファイリングデータのマシン表示モード

マシンモードでは、JVM ソフトウェアでインタプリタされるアプリケーションからの関数ではなく、JVM ソフトウェア自体からの関数が表示されます。また、コンパイルされたメソッドとネイティブメソッドがすべて表示されます。マシンモードは、従来の言語で書かれたアプリケーションのものと同じように見えます。呼び出しスタックは、JVM フレーム、ネイティブフレーム、およびコンパイル済みメソッドフレームを表示します。JVM フレームの中には、インタプリタされた Java、コンパイルされた Java、およびネイティブコードの間の変移コードを表すものがあります。

コンパイルされたメソッドのソースは、Java ソースに対して示されます。このデータは、選択されているコンパイルされたメソッドの特定の例を表します。コンパイルされたメソッドの逆アセンブリは、Java バイトコードではなく、生成されたマシンアセンブラコードを示します。呼び出し元 - 呼び出し先の関係はすべてのオーバーヘッドフレームと、インタプリタされたメソッド、コンパイルされたメソッド、ネイティブメソッドの間の遷移を表すすべてのフレームを示します。

マシン表示モードのタイムラインはすべてのスレッド、LWP または CPU のバーを示し、それぞれの呼び出しスタックはマシンモード呼び出しスタックになります。

OpenMP実行の概要

OpenMP アプリケーションの実際の実行モデルについては、OpenMP の仕様 (たとえば、『OpenMP Application Program Interface, Version 3.0』の 1.3 セクションを参照) で

説明されています。ただし、この仕様では、ユーザーにとって重要ないくつかの実装の詳細が規定されていません。Oracle の実際の実装は、直接記録されたプロファイリング情報を使用しても、スレッドがどのように相互作用するかをユーザーが容易に理解できないようになっています。

シングルスレッドプログラムが実行される場合と同様に、その呼び出しスタックが現在の位置と、どのようにしてそこまで到達したかを示すトレースを示します。このトレースは `_start` と呼ばれるルーチン内の開始命令から始まり、そのルーチンが `main` を呼び出し、それがさらに処理を進めてプログラム内のさまざまなサブルーチンを呼び出します。サブルーチンにループが含まれている場合、プログラムは、ループ終了条件が満たされるまでループ内のコードを繰り返し実行します。その後、実行は次のコードシーケンスへ進み、以後同様に処理が続きます。

プログラムが OpenMP で (自動並列化により) 並列化されている場合、動作は異なります。並列化されたプログラムの直感的なモデルでは、メインスレッド (マスタースレッド) が、シングルスレッドプログラムとまったく同じように実行されます。そのスレッドが並列ループまたは並列領域に到達すると、追加のスレーブスレッドが現れます。各スレッドはマスタースレッドのクローンであり、それらのすべてのスレッドが、異なる作業チャンクごとにループまたは並列領域の内容を並列に実行します。すべての作業チャンクが完了すると、すべてのスレッドの同期がとられ、スレーブスレッドが消失し、マスタースレッドが処理を続行します。

並列化されたプログラムの実際の動作は、それほど直接的なものではありません。コンパイラが並列領域またはループ用のコード (または、その他の任意の OpenMP 構造) を生成するとき、それらの内部のコードが抽出され、Oracle 実装で *mfunction* と呼ばれる独立した関数が作成されます。この関数は、アウトライン関数、またはループ本体関数とも呼ばれます。mfunction の名前は、OpenMP 構造タイプ、抽出元となった関数の名前、その構造が置かれているソース行の行番号を符号化したものです。これらの関数の名前は、パフォーマンスアナライザの上級モードとマシンモードで次の形式で表示されます。ここで、括弧内の名前はその関数の実際のシンボルテーブル名です。

```
bardo_ -- OMP parallel region from line 9 [_$p1C9.bardo_]
atomsum_ -- MP doall from line 7 [_$d1A7.atomsum_]
```

bardo_ -- 行 9 からの OMP 並列領域 [_$p1C9.bardo_] atomsum_ -- 行 7 からの MP doall [_$d1A7.atomsum_]これらの関数には、ほかのソース構造から生成される別の形式もあり、その場合、名前の中の **OMP 並列領域**は、**MP コンストラクト**、**MP doall**、**OMP 領域**のいずれかに置き換えられます。以降の説明では、これらすべてを総称して「*並列領域*」と言います。

並列ループ内のコードを実行する各スレッドは、mfunction を複数回呼び出すことができ、1 回呼び出すたびにループ内の 1 つの作業チャンクが実行されます。すべての作業チャンクが完了すると、それぞれのスレッドはライブラリ内の同期ルーチンまたは縮小ルーチンを呼び出します。その後、マスタースレッドが続行される一方、各スレーブスレッドはアイドル状態になり、マスタースレッドが次の並列領域に入るまで待機します。すべてのスケジューリングと同期は、OpenMP ランタイムの呼び出しによって処理されます。

並列領域内のコードは、その実行中、作業チャンクを実行しているか、ほかのスレッドとの同期をとっているか、行うべき追加の作業チャンクを取り出している場合があります。また、ほかの関数を呼び出す場合もあり、それによってさらに別の関数が呼び出される可能性もあります。並列領域内で実行されるスレーブスレッド (またはマスタースレッド) は、それ自体が、またはそれが

呼び出す関数から、マスタースレッドとして動作し、独自の並列領域に入って入れ子並列を生成する場合があります。

パフォーマンスアナライザは、呼び出しスタックの統計的な標本収集に基づいてデータを収集し、すべてのスレッドにわたってデータを集計したあと、収集したデータの種類に基づいてパフォーマンスのメトリックを関数、呼び出し元と呼び出し先、ソース行、および命令に対して表示します。パフォーマンスアナライザは、OpenMP プログラムのパフォーマンスに関する情報を、ユーザーモード、上級モード、マシンモードの 3 つの表示モードのいずれかで表示します。

OpenMP プログラムのデータ収集についての詳細は、OpenMP のユーザーコミュニティー Web サイトの An OpenMP Runtime API for Profiling (`http://www.compunity.org/futures/omp-api.html`) を参照してください。

OpenMP プロファイルデータのユーザー表示モード

プロファイルデータのユーザーモードの表示では、プログラムが実際に201 ページの「OpenMP実行の概要 」で説明されている直観的モデルに従って実行されたかのように情報が提示されます。マシンモードで表示される実際のデータは、ランタイムライブラリ `libmtsk.so` の実装の詳細を取り込んだもので、これは、モデルに対応していません。上級モードでは、モデルに合うように変更されたデータと、実際のデータが表示されます。

ユーザーモードでは、プロファイルデータの表示はモデルにさらに近くなるよう変更されるため、記録されたデータやマシンモードの表示とは次の点で異なっています。

- 擬似関数は、OpenMP ランタイムライブラリの視点から各スレッドの状態を表すように構築されます。
- 呼び出しスタックは、前述のように、コードの実行方法のモデルに対応するデータを報告するよう操作されます。
- クロックプロファイリング実験の場合、有効な作業の実行に費やされた時間と、OpenMP ランタイムでの待機に費やされた時間に対応する 2 つの追加のパフォーマンスメトリックが構築されます。メトリックは、OpenMP ワークメトリックとOpenMP 待ちメトリックです。
- OpenMP 3.0 および 4.0 プログラムでは、3 番目のメトリックである「OpenMP オーバーヘッド」が構築されます。

擬似関数

擬似関数は、スレッドが OpenMP ランタイムライブラリ内でいずれかの状態にあったイベントを反映するために構築され、ユーザーモードおよび上級モード呼び出しスタック上に置かれます。

次の疑似関数が定義されています。

`<OMP-overhead>`	OpenMP ライブラリ内で実行中

<OMP-idle>	作業を待っているスレーブスレッド
<OMP-reduction>	縮約操作を実行中のスレッド
<OMP-implicit_barrier>	暗黙のバリアで待機中のスレッド
<OMP-explicit_barrier>	明示的なバリアで待機中のスレッド
<OMP-lock_wait>	ロックを待っているスレッド
<OMP-critical_section_wait>	critical セクションに入るのを待っているスレッド
<OMP-ordered_section_wait>	ordered セクションに入る順番を待っているスレッド
<OMP-atomic_wait>	OpenMP 原子構造で待機中のスレッド

スレッドが、擬似関数のいずれかに対応する OpenMP ランタイム状態にある場合、擬似関数がスタック上のリーフ関数として追加されます。スレッドの実際のリーフ関数が OpenMP ランタイム内のどこかに存在する場合、その関数は、リーフ関数として <OMP-overhead> に置き換えられます。そうでない場合、OpenMP ランタイムに入っているすべての PC は、ユーザーモードスタックから除外されます。

OpenMP 3.0 および 4.0 プログラムでは、<OMP-overhead> 擬似関数は使用されません。この擬似関数は、OpenMP オーバーヘッドメトリックに置き換えられています。

ユーザーモード呼び出しスタック

OpenMP 実験の場合は、OpenMP を使用せずにプログラムがコンパイルされたときに取得されたものと同様の、再構築された呼び出しスタックが表示されます。目標は、実際の処理の詳細すべてを示すことではなく、プログラムを直観的に理解できるようにプロファイルデータを表示することです。マスタースレッドとスレーブスレッドの呼び出しスタックが調整され、OpenMP ランタイムライブラリが特定の操作を実行しているときは擬似関数 <OMP-*> が呼び出しスタックに追加されます。

OpenMP メトリック

OpenMP プログラムの時間プロファイルイベントを処理するときは、OpenMP システム内の 2 つの状態でそれぞれ費やされた時間に対応する 2 つのメトリックが示されます。「OpenMP ワーク」と「OpenMP 待ち」です。

スレッドがユーザーコードから実行されたときは、逐次か並列かを問わず、「OpenMP ワーク」に時間が累積されます。スレッドが何かを待って先に進めないときは、その待機が busy-wait (spin-wait) であるかスリープ状態であるかを問わず、「OpenMP 待ち」に時間が累積されます。これら 2 つのメトリックの合計は、時間プロファイル内の総スレッド数メトリックに一致します。

OpenMP 待ちメトリックと OpenMP ワークメトリックは、ユーザーモード、上級モード、およびマシンモードに表示されます。

OpenMP プロファイリングデータの上級表示モード

上級表示モードで OpenMP 実験を確認する場合、OpenMP ランタイムが特定の操作を実行しているときは、ユーザー表示モードと同様に <OMP-*> という形式の擬似関数が表示されます。ただし、上級表示モードでは、並列化されたループ、タスクなどを表す、コンパイラにより生成された mfunction が別個に表示されます。ユーザーモードでは、コンパイラにより生成されたこれらの mfunction はユーザー関数に含められます。

OpenMP プロファイリングデータのマシン表示モード

「マシン」モードでは、すべてのスレッドのネイティブな呼び出しスタックと、コンパイラによって生成されたアウトライン関数が表示されます。

実行のさまざまなフェーズでのプログラムの実際の呼び出しスタックは、ユーザーモデルで説明したものとはまったく異なります。マシンモードでは、呼び出しスタックが測定どおりに表示され、変換は行われず、擬似関数も構築されません。ただし、時間プロファイルのメトリックは依然として示されます。

次に示す各呼び出しスタックでは、libmtsk は OpenMP ランタイムライブラリ内の呼び出しスタックに入っている 1 つ以上のフレームを表しています。どの関数がどの順序で表示されるかの詳細は、バリア用のコードまたは縮小を行うコードの内部的な実装と同様に、OpenMP のリリースによって異なります。

1. 最初の並列領域の前

   最初の並列領域の前最初の並列領域に入る前の時点で存在するスレッドは、マスタースレッドただ 1 つだけです。呼び出しスタックはユーザーモードおよび上級モードの場合と同じです。

マスター
foo
main
_start

2. 並列領域内で実行中

マスター	スレーブ 1	スレーブ 2	スレーブ 3
foo-OMP...			
libmtsk			
foo	foo-OMP...	foo-OMP...	foo-OMP...
main	libmtsk	libmtsk	libmtsk
_start	_lwp_start	_lwp_start	_lwp_start

マシンモードでは、スレーブスレッドはマスターが開始された _start 内ではなく、_lwp_start 内で開始されたものとして示されます。一部のバージョンのスレッドライブラリでは、この関数は _thread_start として表示されます。foo-OMP... の呼び出しは、並列領域に対して生成された mfunction を表します。

3. すべてのスレッドがバリアの位置にある

マスター	スレーブ 1	スレーブ 2	スレーブ 3
libmtsk			
foo-OMP...			
foo	libmtsk	libmtsk	libmtsk
main	foo-OMP...	foo-OMP...	foo-OMP...
_start	_lwp_start	_lwp_start	_lwp_start

スレッドが並列領域内で実行されるときと異なり、スレッドがバリアの位置で待機しているときは、foo と並列領域コード foo-OMP.....の間に OpenMP ランタイムからのフレームは存在しません。その理由は、実際の実行には OMP 並列領域関数が含まれていませんが、OpenMP ランタイムがレジスタを操作し、スタック展開で直前に実行された並列領域関数からランタイムバリアコードへの呼び出しが示されるようにするからです。そうしないと、どの並列領域がバリア呼び出しに関連しているかをマシンモードで判定する方法がなくなってしまいます。

4. 並列領域から出たあと

マスター	スレーブ 1	スレーブ 2	スレーブ 3
foo			
main	libmtsk	libmtsk	libmtsk
_start	_lwp_start	_lwp_start	_lwp_start

スレーブスレッド内では、呼び出しスタック上にユーザーフレームが存在しません。

5. 入れ子の並列領域内にいるとき

マスター	スレーブ 1	スレーブ 2	スレーブ 3	スレーブ 4
	bar-OMP...			
foo-OMP...	libmtsk			
libmtsk	bar			
foo	foo-OMP...	foo-OMP...	foo-OMP...	bar-OMP...
main	libmtsk	libmtsk	libmtsk	libmtsk

マスター	スレーブ 1	スレーブ 2	スレーブ 3	スレーブ 4
_start	_lwp_start	_lwp_start	_lwp_start	_lwp_start

不完全なスタック展開

スタック展開の定義については、195 ページの「呼び出しスタックとプログラムの実行」を参照してください。

スタック展開は、次のようないくつかの場合に失敗します。

- スタックがユーザーコードによって破壊された場合。スタックが破壊された原因によっては、プログラムでコアダンプするか、またはデータ収集コードでコアダンプする場合があります。
- ユーザーコードが、関数呼び出しの標準の ABI 規則に従っていない場合。特に、SPARC プラットフォームで、保存命令が実行される前に復帰レジスタ %o7 が変えられる場合。

  どんなプラットフォームでも、手書きのアセンブラコードには規則違反がある場合があります。
- リーフ PC の関数内の位置が、呼び出し先のフレームがスタックからポップされたあとで、かつ関数が復帰する前である場合。
- 呼び出しスタックに約 250 を超えるフレームが含まれている場合、この呼び出しスタックを完全に展開するだけの領域がコレクタにはありません。この場合、呼び出しスタックの _start から特定の時点までの関数の PC は実験ファイルに記録されません。擬似関数 <Truncated-stack> は、記録されたいちばん上のフレームを集計するために <Total> から呼び出されたものとして示されます。
- x86 プラットフォームで、最適化された関数のフレームをコレクタが展開できなかった場合。

中間ファイル

-E または -P コンパイラオプションを使用して中間ファイルを生成すると、パフォーマンスアナライザは元のソースファイルではなく、この中間ファイルを注釈付きソースコードとして使用します。-E を使用して生成された #line ディレクティブは、ソース行へのメトリックの割り当てで問題が発生する原因となることがあります。

関数が生成されるようにコンパイルされたソースファイルへの参照用の行番号を持たない関数からの命令が存在する場合、次の行が注釈付きのソースに現れます。

function_name -- <instructions without line numbers>

行番号は、次の条件下では欠落することがあります。

- -g オプションを指定しないでコンパイルした場合。
- コンパイルのあとにデバッグ情報がストリップされた場合、またはその情報を含む実行可能ファイルかオブジェクトファイルが移動、削除、変更された場合。
- 関数に、元のソースファイルからではなく、#include ファイルから生成されたコードが含まれている場合。

- 高レベルの最適化で、コードが異なるファイルの関数からインライン化された場合。
- ソースファイルに、ほかの何らかのファイルを参照する `#line` ディレクティブが含まれている場合。この状況は、`-E` オプションでコンパイルしたあと、結果の `.i` ファイルをコンパイルした場合に発生することがあります。また、`-P` フラグでコンパイルした場合にも発生する可能性があります。
- 行番号情報を読み取るオブジェクトファイルが見つからない場合。
- 使用したコンパイラが、不完全な行番号テーブルを生成する場合。

プログラム構造へのアドレスのマッピング

呼び出しスタックが処理され、PC 値に変換されたあと、アナライザは、これらの PC をプログラム内の共有オブジェクト、関数、ソース行、および逆アセンブリ行 (命令) にマップします。ここでは、これらのマッピングについて説明します。

プロセスイメージ

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

プロセスの先頭テキストセクションは、実行可能ファイルそのものから生成されます。そのほかのテキストセクションは、プロセスの開始時に実行可能ファイルとともに読み込まれたか、プロセスによって動的にロードされた共有オブジェクトに対応しています。呼び出しスタック内の PC は、呼び出しスタックの記録時に読み込まれた実行可能ファイルと共有オブジェクトに基づいて解決されます。実行可能ファイルと共有オブジェクトは非常に似ているため、まとめて*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の実行の過程でロードされたり、アンロードされたりする場合があるため、実行のタイミングによって、PC が対応する関数が異なることがあります。また、共有オブジェクトが読み込み解除されたあとに別のアドレスに再度読み込まれた場合は、異なる時点で異なる複数の PC が同じ関数に対応することもあります。

ロードオブジェクトと関数

各ロードオブジェクトには、それが実行可能ファイルか共有オブジェクトかにかかわらず、コンパイラによって生成された命令を含むテキストセクション、データ用のデータセクション、および各種のシンボルテーブルが含まれています。すべてのロードオブジェクトには、そのオブジェクト内に、グローバルに認識されているすべての関数の名前とアドレスを指定する ELF シンボル

テーブルが含まれている必要があります。-g オプションでコンパイルされたロードオブジェクトには、追加のシンボリック情報が含まれています。この情報は ELF シンボルテーブルを補足するものであり、グローバルではない関数に関する情報、関数が属するオブジェクトモジュールに関する追加情報、およびアドレスをソース行に関連付ける行番号情報を提供します。

「*関数*」という用語は、ソースコードで記述された高度な操作を表す一連の命令を表します。この用語は、Fortran で使用されるサブルーチンや、C++ および Java プログラミング言語で使用されるメソッドなどもカバーしています。関数はソースコード内で明確に記述され、その名前は通常、アドレスセットを表すシンボルテーブルに表示されます。プログラムカウンタがそのセットに含まれている場合、プログラムはその関数内で実行されています。

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すべて、関数にマップできます。関数ロードオブジェクト内のアドレス ロードオブジェクト関数のアドレス呼び出しスタック上のリーフ PC およびほかのすべての PC で、まったく同じマッピング情報が使用されます。関数の多くは、プログラムのソースモデルに直接対応します。以降のセクションでは、これに該当しない関数について説明します。

別名を持つ関数

一般に、関数は大域関数として定義されます。このことは、プログラム内のあらゆる部分で関数名が既知であることを意味します。大域関数の名前は、実行可能ファイル内で一意である必要があります。アドレス空間内に同一名の大域関数が複数存在する場合、実行時リンカーはすべての参照をそのうちの 1 つに決定します。その他の関数は実行されず、関数リストにそれらの関数が含まれることはありません。「選択の詳細」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調べることができます。

さまざまな状況で、同じ関数が異なる名前で認識されることがあります。この一般的な例としては、コードの同一部分に対して、いわゆる弱いシンボルと強いシンボルが使用されている場合があります。一般に、強い名前は対応する弱い名前と同じですが、前にアンダースコアが付きます。スレッドライブラリ内の多くの関数には、強い名前、弱い名前、代替内部シンボルに加えて、pthread および Solaris スレッド用に別の名前があります。このようなすべてのケースで、パフォーマンスアナライザの「関数」ビューでは 1 つの名前だけが使用されます。選択される名前は、アドレスをアルファベット順に並べた最後のシンボルです。ほとんどの場合は、この名前がユーザーの使用する名前に対応しています。「選択の詳細」ウィンドウでは、選択した関数のすべての別名が表示されます。

一意でない関数名

別名を持つ関数ではコードの同じ部分に対して複数の名前が反映されますが、状況によっては、コードの複数の部分が同じ名前を持つことがあります。

- モジュール性を実現するために、関数が静的関数として定義されることがあります。これは、その関数名がプログラムの一部 (一般には、コンパイル済みの 1 つのオブジェクトモ

ジュール) でだけ認識されることを意味します。このような場合、プログラムのまったく異なる部分を参照している同じ名前の複数の関数がパフォーマンスアナライザに表示されます。「選択の詳細」ウィンドウでは、こうした関数を区別するために、これら各関数のそれぞれにオブジェクトモジュール名が表示されます。また、こうした関数のどの名前が選択されたとしても、その関数のソース、逆アセンブリ、呼び出し元と呼び出し先を表示することができます。

- ライブラリ関数の弱い名前を持つラッパー関数または割り込み関数がプログラムで使用され、そのライブラリ関数の呼び出しに優先されることがあります。一部のラッパー関数はライブラリ内の元の関数を呼び出しますが、その場合は、その名前の両方のインスタンスがパフォーマンスアナライザの関数リストに表示されます。こうした関数は、元の共有オブジェクトやオブジェクトモジュールが異なるため、それらの情報を基に区別することができます。コレクタは一部のライブラリ関数をラップしますが、ラッパー関数と実際の関数の両方がパフォーマンスアナライザに表示される場合があります。

ストリップ済み共有ライブラリの静的関数

ライブラリ内では、ライブラリ内部で使用される名前がユーザーが使用する名前と競合しないように、静的関数がよく使用されます。ライブラリをストリップすると、静的関数の名前はシンボルテーブルから削除されます。このような場合、パフォーマンスアナライザは、ストリップ済み静的関数を含むライブラリ内のテキスト領域ごとに擬似的な名前を生成します。この名前は `<static>@0x12345` という形式で、`@` 記号に続く文字列は、ライブラリ内のそのテキスト領域のオフセットを表します。パフォーマンスアナライザは、連続する複数のストリップ済み静的関数と単一のストリップ済み静的関数を区別できないため、2 つ以上のストリップ済み静的関数が、各メトリックがまとめられた状態で表示される場合があります。

ストリップ済み静的関数は、その PC が静的関数の保存命令のあとに表示されるリーフ PC である場合を除いて、正しい呼び出し元から呼び出されたように表示されます。シンボリック情報がない場合、パフォーマンスアナライザは保存アドレスを認識しないため、復帰レジスタを呼び出し元として使用するかどうかを判断できません。復帰レジスタは常に無視されます。複数の関数が 1 つの `<static>@0x12345` 関数にまとめられることがあるため、実際の呼び出し元または呼び出し先が隣接する関数と区別されないことがあります。

Fortran での代替エントリポイント

Fortran には、コードの 1 つの部分を指す複数のエントリを保持し、呼び出し元が関数の途中を呼び出せるようにする方法が用意されています。このようなコードをコンパイルしたときに生成されるコードは、メインのエントリポイントの導入部、代替エントリポイントの導入部、関数のコード本体で構成されます。各導入部では、関数があとで復帰するためのスタックが作成され、そのあとで、コード本体に分岐または接続します。

各エントリポイントの導入部のコードは、そのエントリポイント名を持つテキスト領域に常に対応しますが、サブルーチン本体のコードは、エントリポイント名の 1 つだけを受け取ります。受け取る名前は、コンパイラによって異なります。

導入部にそれほど長い時間がかかることはなく、サブルーチンの本体に関連付けられているエントリポイント以外のエントリポイントに対応する関数がパフォーマンスアナライザに表示されることもほとんどありません。通常、代替エントリポイントを含む Fortran サブルーチン内の時間を表す呼び出しスタックは、導入部ではなく、サブルーチンの本体に PC があるため、本体に関連付けられた名前だけが呼び出し先として表示されます。同様に、そうしたサブルーチンからのあらゆる呼び出しは、サブルーチン本体に関連付けられている名前から行われたものとみなされます。

クローン関数

コンパイラは、通常以上の最適化が可能な関数への呼び出しを見分けることができます。こういった呼び出しの一例としては、引数の一部が定数である関数への呼び出しが挙げられます。コンパイラは、最適化できる特定の呼び出しを識別すると、クローンと呼ばれるこの関数のコピーを作成し、最適化されたコードを生成します。クローン関数名は、特定の呼び出しを識別する、符号化された名前です。パフォーマンスアナライザはこの名前を復号化し、クローン関数の各インスタンスを関数リスト内に個別に表示します。各クローン関数は異なる命令セットを持っているため、注釈付き逆アセンブリリストにはクローン関数が個別に表示されます。各クローン関数のソースコードは同じであるため、注釈付きソースリストでは関数のすべてのコピーについてデータが集計されます。

インライン関数

インライン関数とは、実際の呼び出しの代わりに、コンパイラが生成した命令がその関数の呼び出し位置に挿入される関数のことです。インライン化には、C++ のインライン関数定義と明示的または自動的なインライン化の 2 種類があり、どちらもパフォーマンスを向上させるために実行され、またどちらもパフォーマンスアナライザに影響を与えます。

- C++ のインライン関数定義。このようにインライン化する理由は、関数呼び出しのコストがインライン関数によって行われる作業よりはるかに大きいためです。関数呼び出しを設定するより、関数のコードを呼び出し位置に挿入する方が優れています。一般に、アクセス関数は、必要な命令が 1 つだけであることが多いため、インライン化対象として定義されます。`-g` オプションでコンパイルすると、関数のインライン化は無効になります。`-g0` でコンパイルすると、関数のインライン化が許可されます。この方法が推奨されます。
- 高レベルの最適化 (4 および 5) で行われる明示的または自動的なインライン化。明示的および自動的なインライン化は、`-g` が有効なときにも行われます。この種のインライン化を行うのは、関数呼び出しの時間を節約するための場合もあります。しかし、多くの場合は、レジスタの利用や命令の実行スケジューリングを最適化できる命令数を増やすためです。

いずれのインライン化も、メトリックの表示に同じ影響を及ぼします。ソースコードに記述されていて、インライン化された関数は、関数リストにも、また、そうした関数のインライン化先の関数の呼び出し先としても表示されません。通常ならば、インライン化された関数の呼び出し位置で包括的メトリックとみなされるメトリック (呼び出された関数で費やされた時間を表す) が、実

際には呼び出し位置 (インライン化された関数の命令を表す) が原因の排他的メトリックと報告されます。

注記 - インライン化によってデータの解釈が難しくなることがあります。このため、パフォーマンス解析のためにプログラムをコンパイルするときには、インライン化を無効にすることを推奨します。C++ アクセス関数のインライン化は、パフォーマンスコストの上昇を避けるため、無効にすべきではありません。

場合によっては、関数がインライン化されている場合も、いわゆるライン外関数が残されます。一部の呼び出し側ではライン外関数が呼び出され、それ以外では命令がインライン化されます。このような場合、関数は関数リストに含まれますが、関連するメトリックはライン外呼び出しだけを表します。

コンパイラ生成の本体関数

関数内のループまたは並列化ディレクティブが指定された領域を並列化する場合、コンパイラは、元のソースコードには含まれていない新しい本体関数を作成します。こうした関数については、201 ページの「OpenMP実行の概要 」で説明しています。

ユーザーモードでは、パフォーマンスアナライザはこれらの関数を表示しません。上級モードとマシンモードでは、パフォーマンスアナライザはこれらの関数を通常の関数として表示し、コンパイラで生成される名前に加えて、抽出元の関数に基づいて名前を割り当てます。これらの関数の排他的メトリックおよび包括的メトリックは、本体関数で費やされる時間を表します。また、構造の抽出元の関数は各本体関数の包括的メトリックを示します。このプロセスは、201 ページの「OpenMP実行の概要 」で説明しています。

並列ループを含む関数をインライン化した場合、そのコンパイラ生成の本体関数名には、元の関数ではなく、インライン化先の関数の名前が反映されます。

注記 - コンパイラ生成の本体関数の名前は、-g でコンパイルされたモジュールでのみ復号化できます。

アウトライン関数

アウトライン関数は、フィードバック最適化コンパイル中に作成される場合があります。それらは、通常では実行されないコード、特に、最終的な最適化コンパイル用のフィードバックの生成に使用される「試験実行」の際に実行されないコードを表しています。一般的な例は、ライブラリ関数の戻り値でエラーチェックを実行するコードです。通常、エラー処理コードは実行されません。ページングや命令キャッシュの動作を向上させるため、このようなコードはアドレス空間内の別の場所に移動され、別の関数となります。アウトライン関数の名前は、コードの取り出し元関数の名前や特定のソースコードセクションの先頭の行番号を含む、アウトライン化したコードのセクションに関する情報をエンコードします。これらの符号化された名前は、リリースごとに異なります。パフォーマンスアナライザは、読みやすいバージョンの関数名を表示します。

アウトライン関数は、実際には呼び出されず、ジャンプ先になります。同様に、復帰はせず、ジャンプで戻ります。動作をユーザーのソースコードモデルにより近づけるため、パフォーマンスアナライザは、main 関数からそのアウトライン部分への擬似的な呼び出しを生成します。

アウトライン関数は、適切な包括的および排他的メトリックを持つ通常の関数として表示されます。また、アウトライン関数のメトリックは、アウトライン化元の関数の包括的メトリックとして追加されます。

フィードバック最適化コンパイルの詳細は、次のいずれかのマニュアルにある `-xprofile` コンパイラオプションの説明を参照してください。

- 『Oracle Developer Studio 12.5: C ユーザーズガイド』の 付録 B,「C コンパイラオプションリファレンス,」
- 『Oracle Developer Studio 12.5: C++ ユーザーズガイド』の 付録 A,「C++ コンパイラオプション,」
- 『Oracle Developer Studio 12.5: Fortran ユーザーズガイド』の 第 3 章,「Fortran コンパイラオプション」

動的にコンパイルされる関数

動的にコンパイルされる関数は、プログラムの実行中にコンパイルおよびリンクされる関数です。コレクタ API 関数を使用して必要な情報をユーザーが提供しないかぎり、コレクタは C や C++ で記述された動的にコンパイルされる関数に関する情報を把握できません。API 関数については、53 ページの「動的な関数とモジュール」を参照してください。情報を提供しなかった場合、関数は `<Unknown>` としてパフォーマンス解析ツールに表示されます。

Java プログラムの場合は、コレクタが、Java HotSpot 仮想マシンによってコンパイルされるメソッドに関する情報を取得します。API 関数を使用して情報を指定する必要はありません。ほかのメソッドの場合、パフォーマンスツールはメソッドを実行する JVM ソフトウェアの情報を表示します。Java ユーザー表現では、すべてのメソッドがインタプリタされたバージョンとマージされます。マシン表現では、HotSpot でコンパイルされたバージョンが個別に表示され、JVM 関数はインタプリタされたメソッドごとに表示されます。

<Unknown> 関数

状況によっては、PC が既知の関数にマップされないことがあります。このような場合、その PC は `<Unknown>` という名前の特殊関数にマップされます。

次の状況は、PC の `<Unknown>` へのマッピングを示しています。

- C または C++ で記述された関数が動的に生成され、この関数に関する情報がコレクタ API 関数によってコレクタに提供されない場合。コレクタ API 関数の詳細については、53 ページの「動的な関数とモジュール」参照してください。

- Java メソッドが動的にコンパイルされるが、Java プロファイリングが無効である場合。
- PC が実行可能ファイルまたは共有オブジェクトのデータセクション内のアドレスに対応している場合。たとえば、そのデータセクション (`.mul` や `.div` など) 内に複数の関数が存在する SPARC V7 バージョンの `libc.so` があります。コードがデータセクションにあるため、SPARC V8 または SPARC V9 プラットフォームで動作していることをライブラリが検出したときに、動的に書き換えてマシン命令を利用できるようになります。
- 実験ファイルに記録されない実行可能ファイルのアドレス空間内の共有オブジェクトに PC が対応する場合。
- PC が既知のロードオブジェクト内に存在しない場合。もっとも考えられる原因は、展開に失敗して、PC 値として記録された値が PC ではなく、別のワードである場合です。PC が復帰レジスタのとき、既知のロードオブジェクト内に存在しないように見える場合は、`<Unknown>` 関数に割り当てられずに無視されます。
- コレクタにシンボリック情報がない JVM ソフトウェアの内部部分に PC がマップしている場合。

`<Unknown>` 関数の呼び出し元と呼び出し先は呼び出しスタック内の前の PC と次の PC を表し、通常どおり処理されます。

OpenMP の特殊な関数

擬似関数は、スレッドが OpenMP ランタイムライブラリ内の何らかの状態にあったイベントを反映するために構築され、ユーザーモード呼び出しスタック上に置かれます。次の疑似関数が定義されています。

`<OMP-overhead>`	OpenMP ライブラリ内で実行中
`<OMP-idle>`	作業を待っているスレーブスレッド
`<OMP-reduction>`	縮約操作を実行中のスレッド
`<OMP-implicit_barrier>`	暗黙のバリアで待機中のスレッド
`<OMP-explicit_barrier>`	明示的なバリアで待機中のスレッド
`<OMP-lock_wait>`	ロックを待っているスレッド
`<OMP-critical_section_wait>`	critical セクションに入るのを待っているスレッド
`<OMP-ordered_section_wait>`	ordered セクションに入る順番を待っているスレッド

<JVM-System> 関数

ユーザー表現では、`<JVM-System>` 関数は、Java プログラムの実行以外のアクションを実行している JVM ソフトウェアによって使用された時間を表します。JVM ソフトウェアは、ガベージコレクションや HotSpot コンパイルなどのタスクを、この時間間隔内で実行します。ユーザーモードでは、`<JVM-System>` は関数リストに表示されます。

<no Java callstack recorded> 関数

<no Java callstack recorded> 関数は、Java スレッドが Java ユーザー表現のみである点を除き、<Unknown> 関数と同じです。コレクタが Java スレッドからイベントを受信すると、ネイティブスタックを展開し、JVM ソフトウェアを呼び出して対応する Java スタックを取得します。その呼び出しが何らかの理由で失敗した場合は、そのイベントが擬似関数 <no Java callstack recorded> とともにパフォーマンスアナライザに表示されます。JVM ソフトウェアは、デッドロックを回避するためか、Java スタックを展開すると過剰な同期化が発生するときに呼び出しスタックの報告を拒否することがあります。

<Truncated-stack> 関数

呼び出しスタック内の個々の関数のメトリックを記録するため、パフォーマンスアナライザによって使用されるバッファーのサイズは制限されています。呼び出しスタックのサイズが大きくなってバッファーがいっぱいになった場合、呼び出しスタックのサイズがそれ以上増えると、パフォーマンスアナライザが関数のプロファイル情報を強制的にドロップします。ほとんどのプログラムでは排他的 CPU 時間の大部分がリーフ関数で費やされるため、パフォーマンスアナライザは、エントリ関数 _start() と main() から始めて、スタックの下部にあるもっとも重要性の低い関数のメトリックをドロップします。ドロップされた関数のメトリックは、1 つの擬似関数 <Truncated-stack> に統合されます。また、<Truncated-stack> 関数は Java プログラムにも表示される場合があります。

より深いスタックをサポートするには、SP_COLLECTOR_STACKBUFSZ 環境変数の数値を増やして設定します。

<Total> 関数

<Total> 関数は、プログラム全体を表すために使用される擬似的な構成概念です。すべてのパフォーマンスメトリックは、呼び出しスタック上の関数に帰属するのに加えて、特殊関数 <Total> に帰属します。この関数は関数リストの先頭に表示され、そのデータを使用してほかの関数のデータの概略を見ることができます。「呼び出し元-呼び出し先」リストでは、この関数は、任意のプログラムの実行のメインスレッドにおける _start() の名目上の呼び出し元として、また作成されたスレッドの _thread_start() の名目上の呼び出し元としても表示されます。スタックの展開が不完全であった場合、<Total> 関数は、<Truncated-stack> の呼び出し元として表示される可能性があります。

ハードウェアカウンタオーバーフロープロファイルに関連する関数

次の関数は、ハードウェアカウンタオーバーフロープロファイルに関連します。

- collector_not_program_related – カウンタはプログラムに関連しません。
- collector_hwcs_out_of_range – カウンタは、オーバーフローシグナルを生成せずにオーバーフロー値を超えたように見えます。値が記録され、カウンタがリセットされます。
- collector_hwcs_frozen – カウンタは、オーバーフロー値を超えて停止されたように見えますが、オーバーフローシグナルが消失したように見えます。値が記録され、カウンタがリセットされます。
- collector_hwc_ABORT – 通常は特権付きプロセスがカウンタの制御を取得したときに、ハードウェアカウンタの読み取りに失敗したため、ハードウェアカウンタの収集が終了しました。
- collector_record_counter – ハードウェアカウンタイベントの処理および記録中に累積されたカウントであり、ハードウェアカウンタオーバーフロープロファイリングのオーバーヘッドの一部を占めます。この値が <Total> カウントの大きな部分に対応している場合は、オーバーフローの間隔を長くする (つまり、分解能の構成を小さくする) ことが推奨されます。

インデックスオブジェクトへのパフォーマンスデータのマッピング

インデックスオブジェクトは、各パケットに記録されたデータからインデックスを計算できる要素のセットを表します。事前定義されているインデックスオブジェクトセットは、スレッド、CPU、標本、秒です。その他のインデックスオブジェクトは、直接発行されるか、または .er.rc ファイル内にある er_print indxobj_define コマンドで定義できます。パフォーマンスアナライザでは、「ツール」メニューから「設定」を選択し、「ビュー」タブを選択して「「カスタムインデックスオブジェクト」ビューを追加」ボタンをクリックしてインデックスオブジェクトを定義できます。

パケットごとにインデックスが計算され、パケットに関連付けられているメトリックが、そのインデックスのインデックスオブジェクトに追加されます。-1 のインデックスは、<Unknown> インデックスオブジェクトにマップされます。インデックスオブジェクトの階層表示は意味がないので、インデックスオブジェクトのメトリックはすべて排他的メトリックです。

メモリーオブジェクトへのパフォーマンスデータのマッピング

メモリーオブジェクトは、キャッシュ行、ページ、およびメモリーバンクなど、メモリーサブシステム内のコンポーネントです。このオブジェクトは、記録された仮想アドレスまたは物理アドレスから計算されたインデックスから決定されます。メモリーオブジェクトは、仮想ページおよび物理ページについて 8K バイト、64K バイト、512K バイト、および 4M バイトのサイズで事前定義されています。それ以外は、er_print ユーティリティーで mobj_define コマンドによって定義できま

す。また、パフォーマンスアナライザの「設定」ダイアログの「メモリーオブジェクトを追加」ボタンを使用してカスタムメモリーオブジェクトを定義することもできます。詳細は、131 ページの「構成設定」を参照してください。

特定の SPARC システムアーキテクチャーのメモリーオブジェクトを定義するファイルをロードできます。「マシンモデルのロード」ボタンをクリックして、対象のシステムアーキテクチャーを選択します。「適用」または「了解」をクリックすると、オブジェクトの新しいリストが「メモリーオブジェクトビュー」列に表示されます。これらのビューから選択して、関連するデータを表示できます。詳細は、ヘルプで「マシンモデル」を検索してください。

デフォルトでは、パフォーマンスアナライザは、実験が記録されたマシンに適したマシンモデルのファイルをロードします。マシンモデルのファイルでは、メモリーオブジェクトとインデックスオブジェクトの両方を定義できます。

プログラムデータオブジェクトへのデータアドレスのマッピング

正確なメモリー操作に対応するハードウェアカウンタイベントからの PC が見つかると、パフォーマンスアナライザは、コンパイラからハードウェアプロファイルサポート情報内に提供された命令識別子と記述子を使用して、関連するプログラムデータオブジェクトを生成します。

*データオブジェクト*という用語は、ソースコードに記述されているプログラム定数、変数、配列、および構造体や共用体などの集合体のほか、個別の集合体要素を示すために使用します。データオブジェクトのタイプとそのサイズはソース言語によって異なります。多くのデータオブジェクトの名前は明示的にソースプログラム内で付けられますが、名前が付けられないものもあります。データオブジェクトの中には、ほかの単純なデータオブジェクトから生成または集計され、より複雑なデータオブジェクトの集合になるものもあります。

各データオブジェクトには、定義元であり、参照が可能なソースプログラムの領域であるスコープが関連付けられています。このスコープは、大域 (ロードオブジェクトなど)、特定のコンパイル単位 (オブジェクトファイル)、または関数である可能性があります。同一のデータオブジェクトを異なるスコープで定義したり、特定のデータオブジェクトを異なるスコープで異なる方法で参照したりできます。

バックトラックを有効にして収集されたメモリー操作に関するハードウェアカウンタイベントのデータ派生メトリックは、関連付けられたプログラムデータオブジェクトタイプに帰属します。これらのメトリックは、そのデータオブジェクトを含むすべての集合体と、`<Unknown>` や `<Scalars>` などのすべてのデータオブジェクトを含むと見なされる擬似的な `<Total>` に伝搬されます。`<Unknown>` の各種のサブタイプは、`<Unknown>` 集合体まで伝搬されます。次のセクションでは、`<Total>`、`<Scalars>`、および `<Unknown>` データオブジェクトについて説明します。

データオブジェクト記述子

データオブジェクトは、宣言された型と名前の組み合わせで記述します。単純なスカラーデータオブジェクト `{int i}` は、`int` 型の変数 `i` を記述しているのに対して、`{const+pointer`

+int p} は、int 型 p への定数ポインタを記述しています。型名のスペースは「_」(アンダースコア) に置き換えられ、名前の付いていないデータオブジェクトは「-」(ハイフン)、たとえば {double_precision_complex -} という名前で表されます。

集合体全体も同様に、foo_t 型の構造体の場合は {structure:foo_t} と表します。集合体の要素では、その要素のコンテナを追加指定する必要があります。たとえば、前述の foo_t 型の構造体のメンバーである int 型の i の場合は {structure:foo_t}.{int i} となります。集合体はそれ自体、(さらに大きい) 集合体の要素になることも可能で、対応する記述子は集合体記述子を連結したもの、最終的にはスカラー記述子になります。

完全修飾された記述子は、データオブジェクトを明確にするために必ずしも必要ではありませんが、データオブジェクトの識別を支援するために一般的な完全指定を示します。

<Total> データオブジェクト

<Total> データオブジェクトは、プログラム全体のデータオブジェクトを表すために使用される擬似的な構造です。あらゆるパフォーマンスメトリックは、異なるデータオブジェクト (およびそのオブジェクトが属する集合体) のメトリックとして帰属するほか、<Total> という特別なデータオブジェクトに帰属します。これは、データオブジェクトリストの上部に表示されます。そのデータを使用すると、ほかのデータオブジェクトのデータの概略を得ることができます。

<Scalars> データオブジェクト

集合体要素のパフォーマンスメトリックが、それに関連付けられた集合体のメトリック値にさらに帰属するのに対して、すべてのスカラー定数および変数のパフォーマンスメトリックは、擬似的な <Scalars> データオブジェクトのメトリック値にさらに帰属します。

<Unknown> データオブジェクトとその要素

さまざまな状況下で、特定のデータオブジェクトにイベントデータをマップできない場合があります。このような場合、そのデータは、<Unknown> という名前の特殊なデータオブジェクトとその要素の 1 つに次のようにマップされます。

- トリガー PC を持つモジュールが -xhwcprof を使用してコンパイルされていない

  トリガー PC を持つモジュールが -xhwcprof を使用してコンパイルされていない イベントの原因となっている命令またはデータオブジェクトは、オブジェクトコードがハードウェアカウンタプロファイルサポートを指定してコンパイルされていなかったので、識別されませんでした。
- バックトラッキングで有効な分岐先が検出できなかった

  バックトラッキングで有効な分岐先を検出できなかったイベントの原因となっている命令は、コンパイルオブジェクト内で提供されたハードウェアプロファイルサポート情報が、バックトラッキングの妥当性を検証するには不十分だったため、識別されませんでした。

- バックトラッキングで分岐先をトラバースした

  バックトラッキングで分岐先をトラバースしたイベントの原因となっている命令またはデータオブジェクトは、バックトラッキングで命令ストリーム内から制御転送ターゲットが検出されたため、識別されませんでした。
- 識別する記述子がコンパイラから提供されなかった

  コンパイラは、メモリー参照命令のデータオブジェクト情報を提供しませんでした。
- タイプ情報がない

  コンパイラは、その命令をメモリー参照命令として識別しませんでした。
- コンパイラが提供したシンボリック情報から判別不能

  コンパイラには、その命令のシンボリック情報がありません。コンパイラのテンポラリは一般に識別されません。
- ジャンプ命令または呼び出し命令によってバックトラッキングが阻止された

  バックトラッキングが、ジャンプ命令または呼び出し命令によって阻止された イベントの原因となっている命令は、バックトラッキングで命令ストリームの中から分岐命令または呼び出し命令が検出されたため、識別されませんでした。
- バックトラッキングでトリガー PC が検出されなかった

  バックトラッキングでトリガー PC が検出されなかったイベントの原因である命令を、最大のバックトラッキング範囲内から検出できませんでした。
- トリガー命令のあとでレジスタが変更されたため、仮想アドレスを特定できなかった

  トリガー命令のあとでレジスタが変更されたため、仮想アドレスを判別できなかったレジスタがハードウェアカウンタのスキッド中に上書きされたため、データオブジェクトの仮想アドレスを判別できませんでした。
- メモリー参照命令で有効な仮想アドレスが指定されなかった

  メモリー参照命令で有効な仮想アドレスが指定されなかった データオブジェクトの仮想アドレスが有効であるように見えませんでした。

♦ ♦ ♦ 第 7 章

7 注釈付きソースと逆アセンブリデータについて

注釈付きソースコードと注釈付き逆アセンブリコードは、関数内のパフォーマンス低下の原因になっているソース行または命令を特定するのに役立ち、コンパイラがコードに対して行なった変換処理に関するコメントを表示します。ここでは、注釈の処理と、注釈付きコードを解釈する上での問題点をいくつか説明します。

この章では、次の内容について説明します。

- 221 ページの「ツールがソースコードを見つけるしくみ」
- 222 ページの「注釈付きソースコード」
- 229 ページの「注釈付き逆アセンブリコード」
- 233 ページの「「ソース」タブ、「逆アセンブリ」タブ、「PC」タブの特別な行」
- 239 ページの「実験なしのソース/逆アセンブリの表示」

ツールがソースコードを見つけるしくみ

注釈付きソースコードと注釈付き逆アセンブリコードを表示するには、パフォーマンスアナライザと `er_print` ユーティリティーが、実験が実行されたプログラムで使用されているソースコードおよびロードオブジェクトファイルにアクセスできる必要があります。パフォーマンスアナライザがソースを見つけることができない場合は、「解決」ボタンをクリックしてソースへのパスを指定できます。

ロードオブジェクトファイルは、最初に実験の `archives` ディレクトリで検索されます。そこに見つからない場合は、後述のとおり、ソースファイルおよびオブジェクトファイルと同じアルゴリズムを使用して検索されます。

ほとんどの実験で、ソースファイルとオブジェクトファイルはフルパス形式で記録されます。Java ソースファイルにも、ファイルへのディレクトリ構造を示すパッケージ名があります。実験を、それが記録された同じシステムで表示する場合、ソースファイルとロードオブジェクトはフルパスを使用して見つけることができます。実験を別のマシンに移動した場合や別のマシンで表示する場合は、それらのフルパスにアクセスできないことがあります。

ソースファイルとオブジェクトファイルを見つけるために、パスマッピングとパス検索という 2 つの補完的な方法が使用されます。ロードオブジェクトファイルが `archives` サブディレクトリに見つからない場合も、それらを見つけるために同じ方法が使用されます。

パスマップと検索パスを設定すると、ツールが実験で参照されているファイルを見つけやすくなります。パフォーマンスアナライザでは、131 ページの「構成設定」で説明されているように、「設定」ダイアログボックスを使用して「パスマップ」タブでパスマップを設定し、「検索パス」タブを使用して検索パスを設定します。er_print ユーティリティーの場合は、Commands That Control Searching For Source Filesで説明されている pathmap および 157 ページの「ソースファイルの検索を制御するコマンド」 ディレクティブを使用します。

最初にパスマッピングが適用され、ファイルのフルパスの先頭を別のパスで置き換える方法を指定します。たとえば、ファイルが /a/b/c/sourcefile と指定され、pathmap ディレクティブで /a/ から /x/y/ へのマッピングが指定されている場合は、ファイルを /x/y/b/c/sourcefile で見つけることができます。/a/b/c/ を /x/ にマップする pathmap ディレクティブでは、/x/sourcefile でこのファイルを見つけることができます。

パスマッピングによってファイルが見つからない場合は、検索パスが使用されます。検索パスは、指定のベース名を持つファイルを検索するディレクトリのリストを指定します。上記の例では、ベース名は sourcefile です。setpath コマンドで検索パスを設定し、addpath コマンドで検索パスにディレクトリを追加することができます。Java ファイルの場合は、パッケージ名が試されたあとでベース名が試されます。

検索パス内の各ディレクトリを使用して、試行するフルパスが構築されます。Java ソースファイルの場合は、2 つのフルパスが構築されます。1 つはベース名、もう 1 つはパッケージ名のものです。ツールは各フルパスにパスマッピングを適用し、マップされたパスのいずれもファイルを指していない場合は、次の検索パスディレクトリを試します。

検索パスにファイルが見つからず、どのパスマッピング接頭辞も元のフルパスと一致していなかった場合は、元のフルパスが試されます。いずれかのパスマップ接頭辞が元のフルパスと一致していたが、ファイルが見つからなかった場合、元のフルパスは試されません。

デフォルトの検索パスには現在のディレクトリと実験ディレクトリが含まれるため、ソースファイルにアクセスできるようにする 1 つの方法は、これらの場所のいずれかにソースファイルをコピーするか、ソースファイルの現在の場所を指すシンボリックリンクをこれらの場所に置くことです。

注釈付きソースコード

実験の注釈付きソースコードは、「パフォーマンスアナライザ」ウィンドウの左のペインで「ソース」ビューを選択することによって、パフォーマンスアナライザで表示できます。または、実験を実行しなくても、er_src ユーティリティーを使用して注釈付きソースコードを表示できます。ここでは、パフォーマンスアナライザでソースコードを表示する方法について説明します。er_src ユーティリティーを使用した注釈付きソースコードの表示の詳細は、239 ページの「実験なしのソース/逆アセンブリの表示」を参照してください。

パフォーマンスアナライザの注釈付きソースには、次の情報が含まれています。

- 元のソースファイルの内容

- 実行可能ソースコードの各行のパフォーマンスメトリック
- 特定のしきい値を超えたメトリックを含むコード行の強調表示
- インデックス行
- コンパイラ解説

パフォーマンスアナライザの「ソース」ビューのレイアウト

「ソース」ビューは、いくつかの列に分かれています。左側には個々のメトリックの固定幅の列が表示され、右側には注釈付きソースが残りの幅で表示されます。

元のソース行の識別

注釈付きソースで黒字で表示されるすべての行は、元のソースファイルから取得されたものです。注釈付きソース列の各行の先頭の番号は、元のソースファイルの行番号に対応します。別の色で文字が表示されている行は、インデックス行かコンパイラのコメント行です。

「ソース」ビュー内のインデックス行

ソースファイルとは、オブジェクトファイルを生成するためにコンパイルされるか、またはバイトコードに解釈されるファイルのことです。オブジェクトファイルには通常、ソースコード内の関数、サブルーチン、またはメソッドに対応する実行可能コードの領域が 1 つ以上含まれています。パフォーマンスアナライザはオブジェクトファイルを解析して、それぞれの実行可能領域を関数として識別します。そして、オブジェクトコード内で見つけた関数を、オブジェクトコードに関連付けられたソースファイル内の関数、ルーチン、サブルーチン、またはメソッドにマップしようとします。パフォーマンスアナライザは、解析が成功すると、オブジェクトコードで検出された関数の最初の命令に対応する注釈付きソースファイル内の場所に、インデックス行を追加します。

注釈付きソースは、「関数」ビューに表示されるリストにインライン関数が表示されていない場合でも、インライン関数を含むすべての関数のインデックス行を表示します。「ソース」ビューには、インデックス行が赤いイタリック体で、テキストが山括弧内に示されます。もっとも単純な種類のインデックス行は、関数のデフォルトコンテキストに対応しています。関数のデフォルトソースコンテキストは、その関数の最初の命令が帰するソースファイルとして定義されます。次の例は、C 関数 `icputime` のインデックス行を示しています。

```
                  578. int
                  579. icputime(int k)
0.       0.       580. {
                       <Function: icputime>
```

この例が示すように、インデックス行は、最初の命令に続く行に表示されます。C ソースの場合は、最初の命令は、関数本体の先頭が開く括弧に対応します。Fortran ソースの場合は、各サブルーチンのインデックス行が、`subroutine` キーワードを含む行に続きます。また、次の例に示す

ように、main 関数のインデックス行が、アプリケーションの起動時に実行される最初の Fortran ソース命令に続きます。

```
                      1. ! Copyright (c) 2006, 2010, Oracle and/or its affiliates. All Rights
 Reserved.
                      2. ! @(#)omptest.f 1.11 10/03/24 SMI
                      3. ! Synthetic f90 program, used for testing openmp directives and the
                      4. !       analyzer
                      5.
0.    0.    0.    0.  6.        program omptest
                                     <Function: MAIN>
                                  7.
                                  8. !$PRAGMA C (gethrtime, gethrvtime)
```

場合によっては、パフォーマンスアナライザがオブジェクトコード内で見つけた関数を、そのオブジェクトコードに関連付けられたソースファイル内のどのプログラミング命令でもマップできないことがあります。たとえば、そのコードが #include されているか、またはヘッダーファイルなどの別のファイルからインライン化されている場合があります。

また、特別なインデックス行やコンパイラのコメントではない特殊な行は、赤で示されます。たとえば、コンパイラの最適化の結果、ソースファイルに記述されているコードに対応しないオブジェクトコード内の関数について、特別なインデックス行が作成される場合があります。詳細は、233 ページの「「ソース」タブ、「逆アセンブリ」タブ、「PC」タブの特別な行」を参照してください。

コンパイラ解説

コンパイラのコメントは、コンパイラによって最適化されたコードがどのように生成されたかを示します。コンパイラ解説の行は、インデックス行や元のソース行と区別するために、青色で表示されます。コンパイラのさまざまな部分で、実行可能ファイルに解説が組み込まれる可能性があります。各コメントは、ソースの特定の行に関連付けられます。注釈付きソースの書き込み時には、ソース行に対してコンパイラが生成するコメントが、ソース行の直前に挿入されます。

コンパイラのコメントは、最適化するためにソースコードに対して行われた変換の大部分に関する情報を提供します。こうした変換には、ループの最適化、並列化、インライン化、パイプライン化などがあります。次の例は、コンパイラ解説を示しています。

```
0.    0.    0.    0.     28.       SUBROUTINE dgemv_g2 (transa, m, n, alpha, b, ldb,   &
                         29.      &                  c, incc, beta, a, inca)
                         30.       CHARACTER (KIND=1) :: transa
                                  31.       INTEGER   (KIND=4) :: m, n, incc, inca, ldb
                                  32.       REAL      (KIND=8) :: alpha, beta
                                  33.       REAL      (KIND=8) :: a(1:m), b(1:ldb,1:n), c(1:n)
                                  34.       INTEGER            :: i, j
                                  35.       REAL      (KIND=8) :: tmr, wtime, tmrend
                                  36.       COMMON/timer/ tmr
                                  37.
                              Function wtime_  not inlined because the compiler has not seen
                              the body of the routine
```

```
0.    0.    0.    0.     38.       tmrend = tmr + wtime()


                           Function wtime_ not inlined because the compiler has not seen
                           the body of the routine
                           Discovered loop below has tag L16
0.    0.    0.    0.     39.       DO WHILE(wtime() < tmrend)

                          Array statement below generated loop L4
0.    0.    0.    0.     40.       a(1:m) = 0.0
                                  41.

                          Source loop below has tag L6
0.    0.    0.    0.     42.       DO j = 1, n       ! <=-----\ swapped loop indices

                          Source loop below has tag L5
                           L5 cloned for unrolling-epilog.  Clone is L19
                           All 8 copies of L19 are fused together as part of unroll and jam
                           L19 scheduled with steady-state cycle count = 9
                           L19 unrolled 4 times
                           L19 has 9 loads, 1 stores, 8 prefetches, 8 FPadds,
                           8 FPmuls, and 0 FPdivs per iteration
                           L19 has 0 int-loads, 0 int-stores, 11 alu-ops, 0 muls,
                           0 int-divs and 0 shifts per iteration
                           L5 scheduled with steady-state cycle count = 2
                           L5 unrolled 4 times
                           L5 has 2 loads, 1 stores, 1 prefetches, 1 FPadds, 1 FPmuls,
                           and 0 FPdivs per iteration
                           L5 has 0 int-loads, 0 int-stores, 4 alu-ops, 0 muls,
                           0 int-divs and 0 shifts per iteration
0.210 0.210 0.210 0.     43.          DO i = 1, m
4.003 4.003 4.003 0.050  44.             a(i) = a(i) + b(i,j) * c(j)
0.240 0.240 0.240 0.     45.          END DO
0.    0.    0.    0.     46.       END DO
                                  47.       END DO
                                  48.
0.    0.    0.    0.     49.       RETURN
0.    0.    0.    0.     50.       END
```

「ソース」ビューに表示されるコンパイラ解説の種類は、「設定」ダイアログボックスの「ソース/逆アセンブリ」タブを使用して設定できます。詳細は、131 ページの「構成設定」を参照してください。

共通部分式の削除

非常に一般的な最適化の 1 つとして、同じ式が複数の場所に存在するときに、この式のコードを 1 つの場所にまとめることによってパフォーマンスを向上させることができます。たとえば、コードブロックの `if` と `else` の分岐の両方で同じ演算が記述されている場合、コンパイラはその演算を `if` 文の直前に移動することができます。実際にそのようにした場合、コンパイラは以前出現した一方の式に基づいて、命令に行番号を割り当てます。共通のコードに割り当てられた行番号が `if` 構造の 1 つの分岐に対応しており、かつそのコードが実際には常にほかの分

岐を実行する場合、注釈付きソースは、実行されない分岐内の行に関するメトリックを表示します。

ループの最適化

コンパイラは、数種類のループの最適化を実行できます。一般的なものを次に示します。

- ループの展開
- ループのピーリング
- ループの交換
- ループの分散
- ループの融合

ループの展開では、ループ本体内でループを数回反復し、それに応じてループインデックスを調整します。ループの本体が大きくなるほど、コンパイラはより効率的に命令をスケジュールできます。また、ループインデックスの増分や条件検査操作によるオーバーヘッドが減少します。残りのループは、ループのピーリングを使って処理されます。

ループのピーリングでは、ループから多数のループの反復を取り除き、これらをループの前か後ろに適宜移動します。

ループの交換では、キャッシュラインヒット率を最大化するために、入れ子のループの順序を変更してメモリーのストライドを最小限に抑えます。

ループの融合は、隣り合ったループや近接したループを 1 つのループにまとめます。ループの融合からは、ループの展開と同じような利点がもたらされます。さらに、最適化済みの 2 つのループで共通のデータにアクセスする場合は、ループの融合によってループのキャッシュの局所性が改善されて、コンパイラは命令レベルの並列化機能をさらに活用することが可能になります。

ループの分散はループの融合の反対で、ループは複数のループに分割されます。この最適化は、ループ内の計算回数が過度に多くなり、パフォーマンス低下の原因となるレジスタのスピルが発生する場合に適しています。また、ループの分裂は、ループに条件文が含まれている場合にも有効です。場合によっては、条件文を含むものと含まないものの 2 つにループを分割できます。この方法により、条件文を含まないループでのソフトウェアのパイプライン化の機会を増やすことができます。

場合によって、入れ子のループでは、コンパイラはループの分裂を適用してループを分割し、次にループの融合を実行して異なる方法でループをまとめ直すことでパフォーマンスを改善します。この場合は、次の例のようなコンパイラ解説が表示されます。

```
Loop below fissioned into 2 loops
Loop below fused with loop on line 116
[116]    for (i=0;i<nvtxs;i++) {
```

関数のインライン化

インライン関数では、コンパイラは実際の関数呼び出しを行う代わりに、その関数が呼び出されている場所に関数の命令を直接挿入します。つまり、C/C++ マクロと同様に、それぞれの呼び出し場所でインライン関数の命令の複製が作成されます。コンパイラは、高レベルの最適化 (4 および 5) で明示的または自動的なインライン化を実行します。

インライン化によって関数呼び出しの負荷が減り、レジスタの使用や命令のスケジュールを最適化するための命令がさらに提供されますが、その代わりに、コードのメモリー使用量が多くなります。次の例は、コンパイラ解説のインライン化を示しています。

```
                Function initgraph inlined from source file ptralias.c
                    into the code for the following line
0.       0.         44.       initgraph(rows);
```

注記 - コンパイラ解説は、パフォーマンスアナライザの「ソース」ビューで 2 行にまたがって折り返されることはありません。

並列化

Sun、Cray、または OpenMP の並列化ディレクティブを含むコードは、複数のプロセッサ上での並列実行用にコンパイルできます。コンパイラのコメントは、並列化が実行されている場所と実行されていない場所、およびその理由を示します。次の例は、並列化コンピュータの解説を示します。

```
0.       6.324       9. c$omp  parallel do shared(a,b,c,n) private(i,j,k)
                    Loop below parallelized by explicit user directive
                    Loop below interchanged with loop on line 12
0.010    0.010      [10]            do i = 2, n-1

                    Loop below not parallelized because it was nested in a parallel loop
                    Loop below interchanged with loop on line 12
0.170    0.170       11.               do j = 2, i
```

並列実行とコンパイラ生成の本体関数の詳細は、201 ページの「OpenMP実行の概要 」を参照してください。

注釈付きソースの特別な行

「ソース」ビューでは、特殊なケースのためのその他の注釈を表示できます。これらの注釈は、コンパイラ解説の形式で、またはインデックス行と同じ色で特別な行に表示できます。詳細は、233 ページの「「ソース」タブ、「逆アセンブリ」タブ、「PC」タブの特別な行」を参照してください。

ソース行メトリック

実行可能コードの各行のソースコードメトリックは、固定幅の列に表示されます。メトリックは、関数リストのものと同じです。実験のデフォルト値は、.er.rc ファイルを使用して変更できます。詳細は、174 ページの「.er.rc ファイル内のデフォルトの設定」を参照してください。また、パフォーマンスアナライザで「設定」ダイアログボックスを使用して、表示されるメトリックや強調表示するしきい値を変更することもできます。詳細は、131 ページの「構成設定」を参照してください。

注釈付きソースコードは、ソース行レベルでのアプリケーションのメトリックスを示します。注釈付きソースは、アプリケーションの呼び出しスタックに記録された PC (プログラムカウント) を読み取り、各 PC をソース行にマッピングすることによって作成されます。注釈付きソースファイルを生成するために、パフォーマンスアナライザは、最初に特定のオブジェクトモジュール (.o ファイル) またはロードオブジェクト内に生成されたすべての関数を特定し、次に各関数のすべての PC のデータをスキャンします。

注釈付きソースを生成するには、パフォーマンスアナライザが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索して読み取り、PC からソース行へのマッピングを特定できる必要があります。また、ソースファイルを読み取って、表示される注釈付きコピーを生成できる必要もあります。実験のソースコードを検索するために使用されるプロセスについては、221 ページの「ツールがソースコードを見つけるしくみ」を参照してください。

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

ソース行メトリックの解釈

命令のメトリックは、命令の実行待ちの間に得られたメトリックとして解釈する必要があります。イベントが記録されるときに実行中である命令がリーフ PC と同じソース行に存在している場合、メトリックはこのソース行を実行した結果であると解釈できます。ただし、実行中の命令とリーフ PC が存在しているソース行がそれぞれ異なる場合、リーフ PC が存在しているソース行のメトリックの少なくとも一部は、実行中命令のソース行が実行待ちしていた間に集計されたメトリックであると解釈する必要があります。この一例としては、1 つのソース行で計算された値が次のソース行で使用される場合が挙げられます。

正確なハードウェアカウンタを使用したハードウェアカウンタオーバーフロープロファイリング (collect -h からの出力で示されます) の場合、リーフ PC は実行される次の命令ではなく、カウンタオーバーフローの原因となっている命令の PC です。正確でないハードウェアカウンタの場合は、報告されるリーフ PC が、オーバーフローの原因となっている命令の数個あとになる可能性があります。これは、オーバーフローがいつ発生したかを認識するためのカーネルメカニズムのスキッドが可変量であるためです。

メトリックを解釈する方法の問題がもっとも重要になるのは、キャッシュミスやリソースキューストールなどのように実行に大幅な遅延が発生する場合や、命令が前の命令の結果を待ってい

る場合です。このような場合は、ソース行のメトリックが異常に高く見えることがあります。コード内で近くにあるほかの行を調べて、メトリック値が高くなる原因になる行を見つけてください。

メトリックの形式

表14「注釈付きソースコードのメトリック」に、注釈付きソースコードの行に表示可能な 4 つのメトリックの形式を示します。

表 14 注釈付きソースコードのメトリック

メトリック	重要度
(空白)	プログラムに、このコード行に対応する PC が存在しません。コメント行は常にこの空白になります。また、次の場合の見かけ上のコード行も空白になります。 ■ 最適化中に、見かけ上のコード部分のすべての命令が削除されている。 ■ コードが別の場所で繰り返されていて、コンパイラによって共通する部分式が認識され、その行のすべての命令に繰り返し部分の行番号が付けられている。 ■ コンパイラによって、その行の命令に不正な行番号が付けられている。
`0.`	プログラム内のいくつかの PC がこの行から派生したものとしてタグ付けされていますが、それらの PC を参照しているデータがありません。統計的にサンプリングされたかトレースされた呼び出しスタックに、そのような PC は存在しません。`0.` メトリックは、その行が実行されなかったことを意味するのではなく、プロファイリングデータパケットや記録されたトレースデータパケットに統計として表示されなかったことだけを意味します。
`0.000`	この行の少なくとも 1 つの PC がデータに表れていますが、メトリック値の計算でゼロに丸められました。
`1.234`	この行に属するすべての PC のメトリックの合計が、表示されているゼロ以外の数値になりました。

注釈付き逆アセンブリコード

注釈付き逆アセンブリは、関数またはオブジェクトモジュールの命令のアセンブリコードリスト、および各命令に関連付けられたパフォーマンスメトリックを提供します。注釈付き逆アセンブリは複数の方法で表示することができ、どの方法で表示されるかは、行番号のマッピングおよびソースファイルが存在するかどうか、また注釈付き逆アセンブリが要求されている関数のオブジェクトモジュールが既知かどうかによって決まります。

- オブジェクトモジュールが既知でない場合、パフォーマンスアナライザでは指定された関数の命令だけが逆アセンブルされ、どのソース行も逆アセンブリ内に表示されません。
- オブジェクトモジュールが既知の場合、オブジェクトモジュール内のすべての関数が逆アセンブルされます。
- ソースファイルが使用可能であり、かつ行番号データが記録されている場合、パフォーマンスアナライザでは表示の設定に応じて、ソースと逆アセンブリコードがインターリーブされます。
- コンパイラによってオブジェクトコードに何からの解説が挿入されている場合は、対応する設定が指定されていれば、その解説も逆アセンブリでインターリーブされます。

逆アセンブリコード内の各命令には、次の情報の注釈が付けられます。

- コンパイラによって報告されたソース行番号
- 相対アドレス
- 命令の 16 進表現 (要求があった場合)
- 命令のアセンブラの ASCII 表現

呼び出しアドレスの解決が可能な場合、それらのアドレスは関数名などのシンボルに変換されます。メトリックは、命令の行に表示されます。対応する設定が指定されている場合は、インターリーブされた任意のソースコードに表示できます。表示可能なメトリック値は、表14「注釈付きソースコードのメトリック」に示すソースコードの注釈で説明しているとおりです。

複数の場所に `#include` されるコードの逆アセンブリリストには、コードが `#include` されるたびに逆アセンブリ命令が 1 回現れます。ソースコードは初回のみインタリーブされ、逆アセンブリコードの繰り返されるブロックはファイルに表示されます。たとえば、`inc_body.h` と呼ばれるヘッダーで定義されているコードのブロックが `inc_body`、`inc_entry`、`inc_middle`、および `inc_exit` という名前の 4 つの関数によって `#include` されている場合、逆アセンブリ命令のブロックは `inc_body.h` の逆アセンブリリストに 4 回現れますが、ソースコードは逆アセンブリ命令の 4 つのブロックの先頭でのみインターリーブされます。「ソース」ビューに切り替えると、逆アセンブリコードの繰り返しにそれぞれ対応するインデックス行が表示されます。

インデックス行が「逆アセンブリ」ビューに表示される場合があります。「ソース」ビューとは異なり、これらのインデックス行を直接ナビゲーションの目的に使用することはできません。インデックス行の直下にある命令のいずれかにカーソルを置いて「ソース」ビューを選択すると、そのインデックス行で参照されているファイルに移動します。

ほかのファイルのコードを `#include` するファイルでは、ソースコードのインタリーブなしで、インクルードされたコードが逆アセンブリ命令として表示されます。これらの命令のいずれかにカーソルを置いて「ソース」ビューを選択すると、`#include` されたコードを含むファイルが開かれます。このファイルが表示された状態で「逆アセンブリ」ビューを選択すると、インターリーブされたソースコードとともに逆アセンブリコードが表示されます。

インライン関数の場合はソースコードと逆アセンブリコードをインタリーブできますが、マクロの場合はできません。

コードが最適化されていない場合、各命令の行番号は逐次順であり、ソース行と逆アセンブリされた命令は予想どおりにインタリーブされます。最適化されている場合は、あとの行の命令が前の行の命令よりも前に表示されることがあります。アナライザのインタリーブアルゴリズムでは、命令が行 *N* にあると表示される場合は、常に、行 *N* とその行までのすべてのソース行がその命令の前に挿入されます。最適化を行なった結果、制御転送命令とその遅延スロット命令の間にソースコードが現れます。ソース行の *N* に関連するコンパイラのコメントは、その行の直前に挿入されます。

注釈付き逆アセンブリの解釈

注釈付き逆アセンブリを解釈するのは簡単ではありません。リーフ PC とは、次に実行する命令のアドレスです。このため、命令の属性メトリックは、命令の実行待ちに費やされた時間とみ

なされます。ただし、命令が必ずしも順番に実行されるとは限らないため、呼び出しスタックの記録で遅延が発生することもあります。注釈付き逆アセンブリコードを利用するには、実験の記録先であるハードウェアと、そのハードウェアが命令を読み込み、実行する方法を理解しておいてください。

次では、注釈付き逆アセンブリコードを解釈する上での問題点をいくつか説明します。

命令発行時のグループ化

命令は、命令発行グループと呼ばれるグループ単位でロードおよび発行されます。グループに含まれる命令は、ハードウェア、命令の種類、すでに実行された命令、およびほかの命令またはレジスタの依存関係によって異なります。その結果、ある命令が常に前の命令と同じクロックサイクルで実行され、次に実行される命令として現れない場合、その命令の出現回数は実際よりも少なくなることを意味します。呼び出しスタックが記録されるときに、次に実行するとみなされる命令が複数存在する可能性もあります。

命令発行規則はプロセッサの種類ごとに異なり、キャッシュ行内の命令位置合わせに依存します。リンカーはキャッシュ行より高い精度で命令位置合わせを強制的に行うため、関連がないように見える関数を変更すると、異なる命令位置合わせが生じることがあります。位置合わせが異なると、パフォーマンスの向上や劣化が発生することがあります。

次の例では、同じ関数をわずかに異なる状況でコンパイルしてリンクしています。2 つの出力例は、`er_print` ユーティリティーからの注釈付き逆アセンブリリストを示しています。2 つの例の命令は同じですが、位置合わせが異なっています。

次の出力例では、命令位置合わせによって、2 つの命令 `cmp` と `bl,a` が異なるキャッシュ行にマップされています。これらの 2 つの命令の実行待ちに長い時間がかかっています。

```
   Excl.     Incl.
User CPU  User CPU
    sec.      sec.
                            1. static int
                            2. ifunc()
                            3. {
                            4.     int i;
                            5.
                            6.     for (i=0; i<10000; i++)
                               <function: ifunc>
   0.010     0.010              [ 6]    1066c:  clr         %o0
   0.        0.                 [ 6]    10670:  sethi       %hi(0x2400), %o5
   0.        0.                 [ 6]    10674:  inc         784, %o5
                            7.         i++;
   0.        0.                 [ 7]    10678:  inc         2, %o0
## 1.360     1.360              [ 7]    1067c:  cmp         %o0, %o5
## 1.510     1.510              [ 7]    10680:  bl,a        0x1067c
   0.        0.                 [ 7]    10684:  inc         2, %o0
   0.        0.                 [ 7]    10688:  retl
   0.        0.                 [ 7]    1068c:  nop
                            8.     return i;
```

```
                          9. }
```

次の出力例では、命令位置合わせによって、2 つの命令 `cmp` と `bl,a` が同じキャッシュ行にマップされています。これらの命令のどちらかのみ、実行待ちに長い時間がかかっています。

```
   Excl.     Incl.
User CPU  User CPU
    sec.      sec.
                          1. static int
                          2. ifunc()
                          3. {
                          4.     int i;
                          5.
                          6.     for (i=0; i<10000; i++)
                             <function: ifunc>
   0.        0.             [ 6]    10684:  clr         %o0
   0.        0.             [ 6]    10688:  sethi       %hi(0x2400), %o5
   0.        0.             [ 6]    1068c:  inc         784, %o5
                          7.         i++;
   0.        0.             [ 7]    10690:  inc         2, %o0
## 1.440     1.440          [ 7]    10694:  cmp         %o0, %o5
   0.        0.             [ 7]    10698:  bl,a        0x10694
   0.        0.             [ 7]    1069c:  inc         2, %o0
   0.        0.             [ 7]    106a0:  retl
   0.        0.             [ 7]    106a4:  nop
                          8.     return i;
                          9. }
```

命令発行の遅延

特定のリーフ PC の示す命令の発行前に遅延があると、そのリーフ PC の出現回数が多くなることがあります。これは、次のケースを初めとして、さまざまな多くの理由で発生する可能性があります。

- 命令がカーネルにトラップされたときのように、前の命令の実行に時間がかかり、割り込みが不可能な場合。
- 算術演算命令が必要とするレジスタの内容が前の命令によって設定されていて、その命令がまだ完了していない場合。このような遅延の例としては、たとえば、データキャッシュミスが発生したロード命令があります。
- 浮動小数点演算命令が、別の浮動小数点演算命令の終了待ちになっている場合。このような状況は、平方根や浮動小数点除算などのパイプライン化が不可能な命令で発生します。
- 命令を含むメモリーワードが命令キャッシュに含まれていない場合 (I キャッシュミス)。

ハードウェアカウンタオーバーフローの関連付け

一部のプラットフォームおよび `precise` カウンタでの TLB ミスを除き、ハードウェアカウンタオーバーフローイベントの呼び出しスタックは、オーバーフローが発生した時点よりあとの命令シーケ

ンス内のある時点で記録されます。この遅延は、オーバーフローによって生成された割り込みの処理にかかった時間など、さまざまな原因によって発生します。サイクルおよび発行された命令などの一部のカウンタの場合、この遅延は問題になりません。しかし、キャッシュミスや浮動小数点演算をカウントするようなカウンタの場合は、そのオーバーフローの原因となっているものとは別の命令がメトリックの原因とされます。

多くの場合、イベントを発生させた PC は記録された PC の数命令前でしかなく、その命令は逆アセンブリリスト内で正確に特定できます。ただし、この命令範囲内に分岐先がある場合、イベントを引き起こした PC に対応する命令を特定することは困難であるか、または不可能です。

`precise` キーワードでラベル付けされたカウンタを持つプロセッサを備えたシステムでは、バイナリの特殊コンパイルなしでメモリー領域プロファイリングを行うことができます。たとえば、SPARC T4 および M7 プロセッサには、いくつかの `precise` カウンタがあります。システムでメモリー領域プロファイリングが可能かどうかを調べるには、`collect -h` コマンドを実行し、`precise` キーワードを探します。

たとえば、SPARC T7 プロセッサを備えたシステムで次のコマンドを実行すると、使用可能な高精度 raw カウンタが表示されます。

```
$ collect -h | grep -i precise | grep -v alias
          Counters labeled as precise in the list below will collect memory-space data by
 default.
    loads      Instr_ld       precise load-store     events 0123 Load Instructions
    stores     Instr_st       precise load-store     events 0123 Store Instructions
    dcm        DC_miss_commit precise load-store     events 0123 L1 D-cache Misses
        use the 'precise' counter:
    Instr_ld                  precise load-store     events 0123
    Instr_st                  precise load-store     events 0123
    Instr_SPR_ring_ops        precise load-store     events 0123
    Instr_atomic              precise load-store     events 0123
    Instr_SW_prefetch         precise load-store     events 0123
    Instr_block_ld_st         precise load-store     events 0123
    DC_miss_L2_L3_hit_commit  precise load-store     events 0123
                              precise load-store     events 0123
                              precise load-store     events 0123
    DC_miss_commit            precise load-store     events 0123
```

「ソース」タブ、「逆アセンブリ」タブ、「PC」タブの特別な行

パフォーマンスアナライザでは、コード行、命令、またはプログラムカウンタに直接には対応していないいくつかの行が「ソース」、「逆アセンブリ」、および「PC」ビューに表示されます。以降のセクションでは、これらの特別な行について説明します。

アウトライン関数

フィードバック最適化コンパイルで、アウトライン関数が作成されることがあります。これらは、「ソース」ビューと「逆アセンブリ」ビューに特殊なインデックス行として表示されます。「ソース」ビューでは、アウトライン関数に変換されたコードのブロックに注釈が表示されます。

```
                  Function binsearchmod inlined from source file ptralias2.c into the
0.       0 .        58.         if( binsearchmod( asize, &element ) ) {
0.240    0.240      59.             if( key != (element << 1) ) {
0.       0.         60.                 error |= BINSEARCHMODPOSTESTFAILED;
                       <Function: main -- outline code from line 60 [_$o1B60.main]>
0.040    0.040    [ 61]                 break;
0.       0.         62.             }
0.       0.         63.         }
```

「逆アセンブリ」ビューでは、アウトライン関数は通常、ファイルの末尾に表示されます。

```
                       <Function: main -- outline code from line 85 [_$o1D85.main]>
0.       0.            [ 85] 100001034:  sethi       %hi(0x100000), %i5
0.       0.            [ 86] 100001038:  bset        4, %i3
0.       0.            [ 85] 10000103c:  or          %i5, 1, %l7
0.       0.            [ 85] 100001040:  sllx        %l7, 12, %l5
0.       0.            [ 85] 100001044:  call        printf ! 0x100101300
0.       0.            [ 85] 100001048:  add         %l5, 336, %o0
0.       0.            [ 90] 10000104c:  cmp         %i3, 0
0.       0.            [ 20] 100001050:  ba,a        0x1000010b4
                       <Function: main -- outline code from line 46 [_$o1A46.main]>
0.       0.            [ 46] 100001054:  mov         1, %i3
0.       0.            [ 47] 100001058:  ba          0x100001090
0.       0.            [ 56] 10000105c:  clr         [%i2]
                       <Function: main -- outline code from line 60 [_$o1B60.main]>
0.       0.            [ 60] 100001060:  bset        2, %i3
0.       0.            [ 61] 100001064:  ba          0x10000109c
0.       0.            [ 74] 100001068:  mov         1, %o3
```

アウトライン関数の名前は角括弧内に表示され、コードの取り出し元関数の名前や、ソースコード内のセクションの先頭の行番号を含む、アウトライン化したコードのセクションに関する情報をエンコードします。これらの符号化された名前は、リリースごとに異なります。パフォーマンスアナライザは、読みやすいバージョンの関数名を表示します。詳細は、212 ページの「アウトライン関数」を参照してください。

アプリケーションのパフォーマンスデータの収集時にアウトライン関数が呼び出されると、パフォーマンスアナライザは、注釈付き逆アセンブリに特殊な行を表示して、その関数の包括的メトリックを示します。詳細は、239 ページの「包括的メトリック」を参照してください。

コンパイラ生成の本体関数

関数内のループまたは並列化ディレクティブが指定された領域を並列化する場合、コンパイラは、元のソースコードには含まれていない新しい本体関数を作成します。こうした関数については、201 ページの「OpenMP実行の概要 」で説明しています。

コンパイラは、並列構造の種類、構造が抽出された元の関数の名前、構造の元のソース内の先頭の行番号、および並列構造のシーケンス番号をエンコードする本体関数に符号化された名前を割り当てます。これらの符号化された名前は、マイクロタスクライブラリのリリースごとに異なりますが、より分かりやすい名前に復号化されて表示されます。

次の例は、マシンモードで関数リストに表示される一般的なコンパイラ生成の本体関数を示しています。

```
7.415      14.860      psec_ -- OMP sections from line 9 [_$s1A9.psec_]
3.873       3.903      craydo_ -- MP doall from line 10 [_$d1A10.craydo_]
```

これらの例では、構造が抽出された元の関数の名前が最初に示され、次に並列構造の種類、並列構造の行番号、コンパイラ生成の本体関数の符号化された名前が角括弧で示されます。同様に、逆アセンブリコード内にも特殊なインデックス行が生成されます。

```
0.       0.            <Function: psec_ -- OMP sections from line 9 [_$s1A9.psec_]>
0.       7.445         [24]    1d8cc:  save        %sp, -168, %sp
0.       0.            [24]    1d8d0:  ld          [%i0], %g1
0.       0.            [24]    1d8d4:  tst         %i1

0.       0.            <Function: craydo_ -- MP doall from line 10 [_$d1A10.craydo_]>
0.       0.030         [ ?]    197e8:  save        %sp, -128, %sp
0.       0.            [ ?]    197ec:  ld          [%i0 + 20], %i5
0.       0.            [ ?]    197f0:  st          %i1, [%sp + 112]
0.       0.            [ ?]    197f4:  ld          [%i5], %i3
```

Cray のディレクティブでは、関数はソースコード行番号に関連付けされていない可能性があります。このような場合、行番号の代わりに `[ ?]` が表示されます。注釈付きソースコードにインデックス行が表示される場合は、次の例に示すように、インデックス行は行番号なしで命令を示します。

```
                    9. c$mic  doall shared(a,b,c,n) private(i,j,k)

                   Loop below fused with loop on line 23
                   Loop below not parallelized because autoparallelization
                     is not enabled
                   Loop below autoparallelized
                   Loop below interchanged with loop on line 12
                   Loop below interchanged with loop on line 12
3.873     3.903         <Function: craydo_ -- MP doall from line 10 [_$d1A10.craydo_],
                       instructions without line numbers>
0.        3.903     10.            do i = 2, n-1
```

注記 - インデックス行やコンパイラのコメント行は、実際の表示では折り返されません。

動的にコンパイルされる関数

動的にコンパイルされる関数は、プログラムの実行中にコンパイルおよびリンクされる関数です。コレクタ API 関数`collector_func_load()` を使用して必要な情報をユーザーが提供しないかぎり、コレクタは C や C++ で記述された動的にコンパイルされる関数に関する情報を持っ

ていません。「関数」ビュー、「ソース」ビュー、および「逆アセンブリ」ビューに表示される情報は、次のように、collector_func_load() に渡される情報によって異なります。

- 情報が指定されない場合、collector_func_load() は呼び出されません。動的にコンパイルおよびロードされた関数は、<Unknown> として関数リストに表示されます。パフォーマンスアナライザには、関数ソースも逆アセンブリコードも表示されません。
- ソースファイル名や行番号テーブルは指定されていないが、関数の名前、そのサイズとアドレスが指定されている場合は、動的にコンパイルおよびロードされた関数の名前とそのメトリックが関数リストに表示されます。注釈付きソースが使用可能であり、逆アセンブリ命令が表示されます。ただし、行番号は、不明であることを示すために [?] で指定されます。
- ソースファイル名は指定されているが、行番号テーブルが指定されていない場合、ソースファイル名を指定しない場合と同様の情報がパフォーマンスアナライザによって表示されます。ただし、注釈付きソースの先頭には、関数が行番号のない命令で構成されていることを示す特殊なインデックス行が表示されます。次に例を示します。

```
1.121     1.121          <Function func0, instructions without line numbers>
                      1. #include        <stdio.h>
```

- ソースファイル名と行番号テーブルが指定されている場合、関数とそのメトリックは、従来の方法でコンパイルされた関数と同様に「関数」ビュー、「ソース」ビュー、および「逆アセンブリ」ビューに表示されます。

コレクタ API 関数の詳細については、53 ページの「動的な関数とモジュール」を参照してください。

Java プログラムでは、ほとんどのメソッドが JVM ソフトウェアによってインタプリタされます。別個のスレッドで動作する Java HotSpot 仮想マシンは、インタプリタの実行中にパフォーマンスをモニターします。モニタリングプロセス中、仮想マシンは、1 つ以上の解釈されたメソッドを取得し、それらのメソッドのためのマシンコードを生成し、元のマシンコードを解釈するのではなくさらに効率的なマシンコードバージョンを実行すると決定する場合があります。

Java プログラムの場合は、コレクタ API 関数を使用する必要はありません。パフォーマンスアナライザは、次の例に示すように、メソッドのインデックス行の下にある特殊な行を使用して、注釈付き逆アセンブリリスト内に Java HotSpot コンパイル済みコードが存在することを示します。

```
                  11.    public int add_int () {
                  12.       int       x = 0;
2.832     2.832      <Function: Routine.add_int: HotSpot-compiled leaf instructions>
0.        0.         [ 12] 00000000: iconst_0
0.        0.         [ 12] 00000001: istore_1
```

逆アセンブリリストには、コンパイルされた命令ではなく、解釈されたバイトコードのみが示されます。デフォルトでは、コンパイルされたコードのメトリックは、特別な行の隣りに表示されます。排他的および包括的 CPU 時間は、解釈されたバイトコードの行ごとに示されているすべての包括的および排他的 CPU 時間の合計とは異なります。一般に、メソッドが何回か呼び出された場合、コンパイルされた命令の CPU 時間は、解釈されたバイトコードの CPU 時間の合計より長くなります。この不一致は、解釈されたコードがメソッドの最初の呼び出し時に 1 回だけ実行されるのに対して、コンパイルされたコードはそれ以降に実行されるために発生します。

注釈付きソースには、Java HotSpot でコンパイルされた関数は表示されません。その代わりに、行番号なしで命令を示す特別なインデックス行が表示されます。たとえば、前述の例で逆アセンブリの抽出に対応する注釈付きソースは次のようになります。

```
                    11.    public int add_int () {
2.832     2.832       <Function: Routine.add_int(), instructions without line numbers>
0.        0.        12.       int       x = 0;
                      <Function: Routine.add_int()>
```

Java ネイティブ関数

ネイティブコードは、元は C、C++、または Fortran で記述されたコンパイル済みのコードで、Java コードから Java Native Interface (JNI) を介して呼び出されます。次の例は、デモプログラム `jsynprog` に関連付けられたファイル `jsynprog.java` の注釈付き逆アセンブリからの抜粋です。

```
                    5. class jsynprog
                       <Function: jsynprog.<init>()>
0.       5.504         jsynprog.JavaCC() <Java native method>
0.       1.431         jsynprog.JavaCJava(int) <Java native method>
0.       5.684         jsynprog.JavaJavaC(int) <Java native method>
0.       0.            [  5] 00000000: aload_0
0.       0.            [  5] 00000001: invokespecial <init>()
0.       0.            [  5] 00000004: return
```

ネイティブメソッドは Java ソースに含まれていないため、`jsynprog.java` の注釈付きソースの先頭には、行番号なしで命令を示す特別なインデックス行を使って各 Java ネイティブメソッドが表示されます。

```
0.       5.504         <Function: jsynprog.JavaCC(), instructions without line
                          numbers>
0.       1.431         <Function: jsynprog.JavaCJava(int), instructions without line
                          numbers>
0.       5.684         <Function: jsynprog.JavaJavaC(int), instructions without line
                          numbers>
```

注記 - 実際の注釈付きソースの表示では、インデックス行は折り返されません。

クローン関数

コンパイラには、渡される引数の一部が定数である関数の呼び出しなど、追加の最適化を実行できる関数の呼び出しを認識する機能があります。コンパイラは、最適化できる特定の呼び出しを識別すると、クローンと呼ばれるこの関数のコピーを作成し、最適化されたコードを生成します。

注釈付きソースでは、コンパイラ解説は、クローン関数が作成されているかどうかを示します。

```
0.       0.       Function foo from source file clone.c cloned,
                   creating cloned function _$c1A.foo;
                   constant parameters propagated to clone
0.       0.570     27.    foo(100, 50, a, a+50, b);
```

注記 - 実際の注釈付きソースの表示では、コンパイラのコメント行は折り返されません。

クローン関数名は、特定の呼び出しを識別する符号化された名前です。前述の例では、コンパイラ解説は、クローン関数の名前が `_$c1A.foo` であることを示します。次に示すように、この関数は関数リストに表示されます。

```
0.350     0.550     foo
0.340     0.570     _$c1A.foo
```

クローン関数はそれぞれ異なる命令セットを持っているため、注釈付き逆アセンブリリストにはクローン関数が個別に表示されます。これらはソースファイルに関連付けられていないため、命令はいずれのソース行番号とも関連付けられていません。次の例は、クローン関数の注釈付き逆アセンブリの最初の数行を示しています。

```
0.       0.           <Function: _$c1A.foo>
0.       0.           [?]    10e98:  save        %sp, -120, %sp
0.       0.           [?]    10e9c:  sethi       %hi(0x10c00), %i4
0.       0.           [?]    10ea0:  mov         100, %i3
0.       0.           [?]    10ea4:  st          %i3, [%i0]
0.       0.           [?]    10ea8:  ldd         [%i4 + 640], %f8
```

静的関数

ライブラリで内部的に使用される名前が、ユーザーが使用する可能性のある名前と競合しないように、ライブラリ内ではほとんど静的関数が使用されます。ライブラリをストリップすると、静的関数の名前はシンボルテーブルから削除されます。このような場合、パフォーマンスアナライザは、ストリップ済み静的関数を含むライブラリ内のテキスト領域ごとに擬似的な名前を生成します。この名前は `<static>@0x12345` という形式で、`@` 記号に続く文字列は、ライブラリ内のそのテキスト領域のオフセットを表します。パフォーマンスアナライザは、連続する複数のストリップ済み静的関数と単一のストリップ済み静的関数を区別できないため、2 つ以上のストリップ済み静的関数が、各メトリックがまとめられた状態で表示される場合があります。次の例は、サンプルの `jsynprog` デモの関数リスト内の静的関数を示しています。

```
0.       0.       <static>@0x18780
0.       0.       <static>@0x20cc
0.       0.       <static>@0xc9f0
0.       0.       <static>@0xd1d8
0.       0.       <static>@0xe204
```

「PC」ビューでは、これらの関数は、次のようにオフセットとともに表示されます。

```
0.       0.       <static>@0x18780 + 0x00000818
0.       0.       <static>@0x20cc + 0x0000032C
0.       0.       <static>@0xc9f0 + 0x00000060
```

```
0.       0.       <static>@0xd1d8 + 0x00000040
0.       0.       <static>@0xe204 + 0x00000170
```

ストリップ済みライブラリ内で呼び出された関数は、「PC」ビューで次のように表示される場合もあります。

```
<library.so> -- no functions found + 0x0000F870
```

包括的メトリック

注釈付き逆アセンブリでは、アウトライン関数にかかった時間にタグを付けるための特殊な行が存在します。

次の例は、アウトライン関数が呼び出されたときに表示される注釈付き逆アセンブリを示しています。

```
0.       0.        43.         else
0.       0.        44.         {
0.       0.        45.                 printf("else reached\n");
0.       2.522         <inclusive metrics for outlined functions>
```

ストア命令とロード命令の注釈

-xhwcprof オプションでコンパイルすると、コンパイラは、ストア (st) 命令とロード (ld) 命令についての追加情報を生成します。逆アセンブリリストに、注釈付きの st 命令と ld 命令が表示されます。

分岐先

-xhwcprof オプションでコンパイルした場合、注釈付き逆アセンブリリストに表示される擬似行 <branch target> は、複数のコードパスから到達できる命令の PC に対応します。

実験なしのソース/逆アセンブリの表示

注釈付きソースコードと注釈付き逆アセンブリコードは、実験を実行しなくても、er_src ユーティリティーを使用して表示できます。メトリックスが表示されないことを除けば、この表示は、パフォーマンスアナライザで生成されるものと同じです。er_src コマンドの構文は次のとおりです。

```
er_src [ -func | -{source,src} item tag | -{disasm,dis} item tag |
```

`-{cc,scc,dcc}` *com-spec* | `-outfile` *filename* | `-V`] *object*

object は、実行可能ファイル、共有オブジェクト、またはオブジェクトファイル (`.o` ファイル) の名前です。

item は、実行可能ファイルまたは共有オブジェクトを構築するために使用される関数、ソースファイル、またはオブジェクトファイルの名前です。*item* は、*function*'*file*'の形式でも指定できます。その場合、`er_src` は、指定されたファイルのソースコンテキストに、指定された関数のソースまたは逆アセンブリを表示します。

tag は、同じ名前の関数が複数存在する場合に、参照する *item* を決定するために使用されるインデックスです。これは必須ですが、関数の解決に不要な場合は無視されます。

特別な item および tag の `all -1` は、オブジェクトのすべての関数に対して、注釈付きソースまたは逆アセンブリを生成するように `er_src` に指示します。

注記 - 実行可能ファイルや共有オブジェクトに `all -1` を使用した結果生成される出力は、非常に大きくなることもあります。

以降のセクションでは、`er_src` ユーティリティーで受け入れられるオプションについて説明します。

-func

指定オブジェクトのすべての関数を一覧表示します。

-{source,src} *item tag*

リストされた item の注釈付きソースを示します。

-{disasm,dis} *item tag*

リストに逆アセンブリを含めます。デフォルトでは、リストに逆アセンブリは含まれません。使用可能なソースがない場合は、コンパイラ解説のない逆アセンブリのリストが生成されます。

-{cc,scc,dcc} *com-spec*

どのコンパイラの解説クラスを表示するかを指定します。*com-spec* は、コロンで区切られたクラスのリストです。*com-spec* は、`-scc` オプションが指定されている場合はソースのコンパイラのコメントに、`-dcc` オプションが指定されている場合は逆アセンブリのコメントに、`-cc` が指定され

ている場合は両方のコメントに適用されます。これらのクラスについては、153 ページの「ソースリストと逆アセンブリリストを管理するコマンド」を参照してください。

コメントクラスは、デフォルト値ファイルで指定することができます。最初にシステム全体のデフォルトファイル `er.rc` が読み取られ、次にユーザーのホームディレクトリの `.er.rc` ファイル (存在する場合)、次に現在のディレクトリの `.er.rc` ファイルが読み取られます。ホームディレクトリの `.er.rc` ファイル内のデフォルト値はシステムのデフォルト値よりも優先され、現在のディレクトリの `.er.rc` ファイル内のデフォルト値は、ユーザーのホームおよびシステムのデフォルト値よりも優先されます。これらのファイルはパフォーマンスアナライザや `er_print` ユーティリティーによっても使用されますが、`er_src` ユーティリティーが使用するのは、ソースおよび逆アセンブリのコンパイラ解説に関する設定だけです。デフォルトファイルについては、174 ページの「`.er.rc` ファイル内のデフォルトの設定」を参照してください。`er_src` ユーティリティーは、デフォルト値ファイル内の `scc` および `dcc` 以外のコマンドを無視します。

`-outfile` *filename*

リストの出力用ファイル *filename* を開きます。デフォルトの場合、またはファイル名がダッシュ (-) の場合は、出力は `stdout` に書き込まれます。

`-V`

現在のリリースバージョンを出力します。

♦ ♦ ♦ 第 8 章

8 実験の操作

この章では、コレクタおよびパフォーマンスアナライザとともに利用できるユーティリティーについて説明します。

この章では、次の内容について説明します。

- 243 ページの「実験の操作」
- 244 ページの「実験へのラベル付け」
- 248 ページの「その他のユーティリティー」

実験の操作

実験は、コレクタによって作成されたディレクトリ内に格納されます。実験の操作に、cp、mv、rm など通常の UNIX® コマンドを使用し、ディレクトリに適用することができます。Forte Developer 7 (Sun™ ONE Studio 7, Enterprise Edition for Solaris) より前のリリースの実験には当てはまりません。このため、これらの UNIX コマンドのような働きを持つ、実験のコピー、移動、および削除用の 3 つのユーティリティーが用意されています。次に、これらのユーティリティー er_cp(1)、er_mv(1)、er_rm(1) を説明します。

実験のデータには、プログラムによって使用された各ロードオブジェクトのアーカイブファイルが含まれます。これらのアーカイブファイルには、ロードオブジェクトの絶対パスとその最終修正日付が含まれています。実験を移動またはコピーしたときにこの情報が変更されることはありません。

er_cp ユーティリティーを使った実験のコピー

次の 2 つの形式の er_cp コマンドを使用できます。

```
er_cp [-V] experiment1 experiment2
er_cp [-V] experiment-list directory
```

最初の形式の er_cp コマンドは、*experiment1* を *experiment2* にコピーします。*experiment2* が存在する場合、er_cp はエラーメッセージを出力して終了します。2 つ目の形式は、リスト中の空白で区切られた一群の実験をディレクトリにコピーします。ディレクトリにコピーする実験と同

じ名前の実験がすでに含まれている場合は、er_cp ユーティリティーはエラーメッセージで終了します。-V オプションは、er_cp ユーティリティーのバージョンを出力します。このコマンドでは、以前のバージョンのツールで作成された実験はコピーされません。

er_mv ユーティリティーを使った実験の移動

次の 2 つの形式の er_mv コマンドを使用できます。

```
er_mv [-V] experiment1 experiment2
er_mv [-V] experiment-list directory
```

最初の形式の er_mv コマンドは、*experiment1* を *experiment2* に移動します。*experiment2* が存在する場合、er_mv ユーティリティーはエラーメッセージを出力して終了します。2 つ目の形式のコマンドは、リスト中の空白で区切られた一群の実験をディレクトリに移動します。移動先のディレクトリに移動対象の実験と同じ名前の実験がすでに存在する場合、er_mv ユーティリティーはエラーメッセージを出力して終了します。-V オプションは、er_mv ユーティリティーのバージョンを出力します。このコマンドでは、以前のバージョンのツールで作成された実験は移動されません。

er_rm ユーティリティーを使った実験の削除

er_rm ユーティリティーは、実験または実験グループのリストを削除します。実験グループが削除されるときは、グループ内の実験が削除されてから、グループファイルが削除されます。

er_rm コマンドの構文は次のとおりです。

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

-f オプションを使用するとエラーメッセージが表示されず、実験が存在するかどうかにかかわらずコマンドは正常に終了します。-V オプションは、er_rm ユーティリティーのバージョンを出力します。このコマンドでは、以前のリリースのツールで作成された実験が削除されます。

実験へのラベル付け

er_label コマンドを使用すると、実験の部分を定義し、それに名前つまりラベルを割り当てることができます。ラベルは、ユーザーが開始時間と停止時間のマーカーを使って定義した実験内の 1 つ以上の期間中に発生したプロファイリングイベントを取得します。

時間マーカーは、現在時間、時間オフセットだけ増減した現在時間、または実験の開始時間に対するオフセットとして指定できます。ラベルには時間間隔をいくつでも指定でき、作成後のラベルに間隔を追加することもできます。

er_label ユーティリティーでは、間隔にはマーカーのペア (開始時間のあとに停止時間) が指定されていると想定しています。停止マーカーが開始マーカーの前に指定されていたり、開始マーカーが前の開始マーカーのあとに続いている (間に停止マーカーがない)、停止マーカーが前の停止マーカーのあとに続いている (間に開始マーカーがない) など、順序がおかしいマーカーはユーティリティーによって無視されます。

実験にラベルを割り当てるには、er_label コマンドをコマンド行で実行するか、またはそれをスクリプト内で実行します。実験へのラベルの追加が完了すると、それらのラベルを使用してフィルタリングを行えます。たとえば、127 ページの「フィルタリングのためのラベルの使用」で説明されているように、ラベルに定義されている期間のプロファイリングイベントを含めたり除外したりするように実験をフィルタリングしてもかまいません。

注記 - ほかのキーワードと同じラベル名 (フィルタリングに使用される可能性のあるもの) は、競合や予期しない結果を招く可能性があるため、作成しないようにしてください。er_print -describe コマンドを使用して、実験のキーワードを確認できます。

---

er_label コマンドの構文

er_label コマンドの構文です。

```
er_label -o experiment-name -n label-name -t {start|stop}[=time-specification] [-C comment
```

オプションは次のように定義されています。

-o *experiment-name* は必須オプションで、ラベルを付ける実験の名前を指定します。実験名は 1 つだけ指定でき、実験グループはサポートされません。-o オプションは、コマンド行の任意の場所に指定できます。

-n *label-name* は必須オプションで、ラベル名を指定します。

label-name は任意の長さで指定できますが、文字で始まる英数字である必要があり、文字列の前後に引用符が使用されている場合でもスペースを含めることはできません。*label-name* が存在する場合は新しい条件が追加され、存在しない場合はそのラベルが作成されます。1 つの -n 引数が必要ですが、これをコマンド行の任意の場所に指定できます。ラベル名は大文字と小文字が区別されません。ラベル名がフィルタ内に存在する可能性のあるほかの名前と競合してはいけません。これには、実験内のプロパティーや、メモリーオブジェクトまたはインデックスオブジェクトの名前が含まれます。ロードされた実験のプロパティーは、er_print describe コマンドで一覧表示されます。メモリーオブジェクトは、er_print mobj_list コマンドで一覧表示されます。インデックスオブジェクトは、er_print indxobj_list コマンドで一覧表示されます。

-C *comment* は、ラベルに関する省略可能なコメントです。1 つのラベルに複数の -C オプションを使用でき、ラベルが表示されるときにはコメントがセミコロンとスペースをはさんで連結されます。たとえば、ラベル内の各時間間隔に関する情報を提供するために、複数のコメントを使用できます。

-t start|stop =*time-specification* は、実験内の時間範囲を定義するために使用される、開始点または停止点の指定です。=*time-specification* を省略した場合は、現在時間のマーカーが作成されます。

time-specification は、次のいずれかの形式で指定できます。

hh:mm:ss.uuu
: 開始マーカーまたは停止マーカーを置くべき位置を、実験の開始からの相対時間で指定します。少なくとも秒を指定する必要があり、時、分、およびサブ秒は任意で指定できます。

  指定する時間の値は、次のように解釈されます。

  nn
  : 整数 (コロンなし) を指定した場合は、秒と解釈されます。値が 60 より大きい場合、ラベルでは秒が mm:ss に変換されます。たとえば、-t start=120 は、実験の開始後 02:00 に開始マーカーを置きます。

  nn.nn
  : 任意の精度の小数を含めた場合は、秒の小数部と解釈され、ナノ秒精度で保存されます。たとえば、-t start=120.3 は、実験の開始後 02:00.300 (2 分と 300 ナノ秒) に開始マーカーを置きます。

  nn:nn
  : nn:nn 形式を使用して時間を指定した場合は、mm:ss と解釈され、mm の値が 60 より大きい場合、時間は hh:mm:ss に変換されます。ss に指定する数値は 0 - 59 である必要があり、それ以外の場合はエラーが発生します。たとえば、-t start=90:30 は、実験の開始後 01:30:30 (1 時間 30 分 30 秒) に開始マーカーを置きます。

  nn:nn:nn
  : nn:nn:nn 形式を使用して時間を指定した場合は、hh:mm:ss と解釈されます。分と秒に指定する数値は 0 - 59 である必要があり、それ以外の場合はエラーが発生します。たとえば、-t stop=01:45:10 は、実験の開始後 1 時間 45 分 10 秒に停止マーカーを置きます。

@
: 実験のマーカーを er_label コマンドが実行された時点に置くために現在時間を指定します。現在時間はコマンドの 1 回の呼び出しで一度設定され、@ を使用する追加マーカーは元のタイムスタンプ値を基準として設定されます。

@+*offset*
: 現在のタイムスタンプよりあとの時間を指定します。ここで、*offset* は、前述と同じ *hh:mm:ss.uuu* 規則を使用する時間です。この時間形式では、元のタイムスタンプより指定の時間だけあとにマーカーが置かれます。た

とえば、-t stop=@+180 は、現在時間より 3 分後に停止マーカーを置きます。

@-*offset* 現在のタイムスタンプより前の時間を指定します。ここで、*offset* は、前述と同じ *hh:mm:ss.uuu* 規則を使用する時間です。この時間形式では、元のタイムスタンプより指定の時間だけ前にマーカーが置かれます。たとえば、-t start=@-20:00 は、現在時間より 20 分前に開始マーカーを置きます。実験が少なくとも 20 分実行されていない場合、このマーカーは無視されます。

1 つの er_label コマンドで複数の -t 指定を使用したり、同じラベル名の別々のコマンドで複数の -t 指定を使用したりできますが、これは -t start マーカーと -t stop マーカーのペアで指定するようにしてください。

-t start または -t stop オプションのあとに時間指定がない場合、指定は =@ と見なされます。マーカーの 1 つには時間指定を含める必要があります。

er_label の例

例 19 実験の開始を基準とする時間マーカーを使用してラベルを定義する

実験の開始 15 秒後から 10 分間の実行部分をカバーする snap という名前のラベルを実験 test.1.er に定義するには、次のコマンドを使用します。

```
% er_label -o test.1.er -n snap -t start=15 -t stop=10:15
```

または、間隔のマーカーを個別のコマンドで指定することもできます。

```
% er_label -o test.1.er -n snap -t start=15
% er_label -o test.1.er -n snap -t stop=10:15
```

例 20 現在時間を基準とする時間マーカーを使用してラベルを定義する

5 分前から現在時間までの実行部分をカバーする last5mins という名前のラベルを実験 test.1.er に定義するには、次のようにします。

```
% er_label -o test.1.er -n last5mins -t start=@-05:00 -t stop
```

スクリプトでの er_label の使用

er_label の使用方法の 1 つは、クライアントによって独立した 1 つ以上のプロセスとして駆動されるサーバープログラムのプロファイリングをサポートすることです。この使用モデルの場

合、サーバーでの実験の作成を開始するため、collect コマンドを使ってサーバーを起動します。サーバーが起動され、クライアントの要求を受け入れる準備が整ったら、要求を発行してサーバーを駆動するとともに、er_label を実行してクライアント要求が発生する実験の部分にラベルを付けるようなクライアントスクリプトを実行できます。

次のサンプルクライアントスクリプトは、サーバーに対して実行された要求ごとに、test.1.er 実験に時間ラベルを生成します。作成される 5 つのラベルはそれぞれ、指定された要求の処理に消費された時間をマークします。

```
for REQ in req1 req2 req3 req4 req5
          do

          echo "========================================================="
          echo " $REQ started at `date`"

          er_label -o test.1.er -n $REQ -t start=@
          run_request $REQ
          er_label -o test.1.er -n $REQ -t stop=@
          done
```

次のサンプルスクリプトは、すべての要求を含む all という単一のラベルを生成する別の使用方法を示しています。

```
for REQ in req1 req2 req3 req4 req5
          do

          echo "========================================================="
          echo " $REQ started at `date`"

          er_label -o test.1.er -n all -t start=@
          run_request $REQ
          er_label -o test.1.er -n all -t stop
          done
```

er_label の 2 番目の呼び出しでは、-t stop のあとに時間指定がないため、デフォルトで stop=@ となります。

より複雑なスクリプトを作成し、複数のスクリプトを同時に同じノードまたは異なるノードで実行できます。すべてのノードからアクセスできる共有ディレクトリに実験が置かれている場合、スクリプトは同じ実験に間隔をマークできます。各スクリプトのラベルは、同じ場合も異なる場合もあります。

その他のユーティリティー

その他のいくつかのユーティリティーは一般に、通常の状況では必要ありません。ここでは、完全な説明を行うためこれらのユーティリティーについて解説し、それらを使用することが必要となる可能性がある状況についても説明します。

er_archive ユーティリティー

er_archive コマンドの構文は次のとおりです。

```
er_archive [ -nqF ] [-s option [-m regexp ] ] experiment er_archive -V
```

er_archive は、実験が正常に完了したときに (アーカイブが無効になった状態で実行されていないかぎり) 自動的に実行されます。実験で参照された共有オブジェクトのリストを読み取り、実験で参照されたすべての共有オブジェクトを初期の実験内のディレクトリ *archives* にコピーします。これらのコピーは、初期の実験とそのサブ実験に同じ名前を持つ異なるバージョンのオブジェクトを含めることができるように、符号化された名前とともに格納されます。Java 実験の場合は、参照されたすべての .jar ファイルも実験にコピーされます (er_archive は、*.archive* ファイルを生成しなくなりました)。ターゲットプログラムが異常終了した場合は、er_archive が実行されない可能性があります。その場合は、手動で実行するようにしてください。

ユーザーが実験を、その実験が記録されたマシンとは異なるマシン上で検査する場合は、すべての共有オブジェクトの正しいバージョンを取得するために、er_archive を実験の完了時に実行するか、またはデータが記録されたマシン上の実験に対して手動で実行する必要があります。

共有オブジェクトが見つからない場合、そのタイムスタンプが実験で記録されているタイムスタンプとは異なる場合、または er_archive が実験が記録されたマシンとは異なるマシン上で実行されている場合、それらのオブジェクトはコピーされません。er_archive が手動で (-q フラグなしで) 実行された場合は、警告が stderr に書き込まれます。このような共有オブジェクトを参照している実験が er_print またはアナライザによって読み取られた場合も、警告が生成されます。

er_archive ではまた、手動で実行するときに -s 引数を指定することによって、ソース (必要なすべてのオブジェクトファイルと必要なシンボルテーブルを含む) をアーカイブすることもできます。ソースのアーカイブは、環境変数 SP_ARCHIVE_ARGS を設定することによって、実験が完了したときに自動的に実行されるように指定できます。その環境変数には、-s 引数と -m 引数を引数のペアで、かつ 1 つ以上のスペースで区切られたオプションとして含めることができます。コマンド行で -s 引数が複数指定されている場合は、最後の引数が優先されます。-s がコマンド行で渡されると同時に、環境変数によっても設定されている場合は、環境変数のオプションが優先されます。

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

–n

名前付きの実験のみをアーカイブし、子孫はアーカイブしません。

–q

stderr に警告を出力しません。警告はアーカイブファイルに組み込まれ、パフォーマンスアナライザ、または er_print ユーティリティーからの出力に表示されます。

–F

アーカイブファイルを強制的に書き込み、または再書き込みします。この引数を使用すると、er_archive を手動で実行し、警告を含むファイルを再書き込みできます。

–s *option*

ソースファイルのアーカイブを指定します。*option* に使用できる値は次のとおりです。

no	どのソースファイルもアーカイブしません。
all	見つかったすべてのソース、オブジェクト、および .anc ファイルをアーカイブします。
used[src]	実験で記録されたデータの対象となる関数に関連しており、かつ見つかったソース、オブジェクト、および .anc ファイルをアーカイブします。 コマンド行または環境変数で -s 引数が複数指定されている場合は、指定されたすべてのオプションが同じである必要があります。そうでない場合、er_archive はエラーで終了します。

–m *regex*

-s フラグで指定されており、かつ実行可能ファイルまたは共有オブジェクト内に記録されているそのフルパス名が指定されている *regex* に一致するソース、オブジェクト、および .anc ファイルのみをアーカイブします。regex の詳細は、regex(5) のマニュアルページを参照してください。

コマンド行または環境変数で複数の -m 引数を指定できます。ソースファイルがいずれかの式に一致している場合は、そのファイルがアーカイブされます。

–V

er_archive ユーティリティーのバージョン番号の情報を書き込み、終了します。

er_export ユーティリティー

er_export コマンドの構文は次のとおりです。

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

er_export ユーティリティーは、実験に含まれている raw データを ASCII テキストに変換します。ファイルの書式と内容は変更される可能性があり、どのような用途でも特定の書式や内容に依存するべきではありません。このユーティリティーは、パフォーマンスアナライザが実験を読み取ることができない場合にのみ使用することを目的としています。この出力を使用すると、ツール開発者は raw データを把握したり、障害を解析したりできます。-V オプションは、バージョン番号の情報を表示します。

er_html ユーティリティー

er_html コマンドの構文は次のとおりです。

```
er_html [-AO] [-c path] [-cc compiler_commentary_class] [-ct] [-h]
      [-im metric_list] [-l limit] [-n column_names] [-o|-O directory]
      [-q] [-tp percent] [-v] [-V] experiment-list
```

er_html ユーティリティーは、1 つ以上の実験または実験グループを、ブラウザで表示できる HTML 形式にエクスポートします。表示するメトリックとその報告方法を選択することも、er_html で自動的にメトリックを選択することもできます。

このユーティリティーでは、アプリケーションをリモートで解析したり、実験結果を確認のためにほかのユーザーと共有したり、何らかの変更を行う前にアプリケーションのパフォーマンスをアーカイブしたりできます。er_html の使用方法およびすべてのオプションの意味については、er_html(1) のマニュアルページを参照してください。

♦ ♦ ♦ 第 9 章

9 カーネルプロファイリング

この章では、Oracle Solaris のロード実行中に、Oracle Developer Studio のパフォーマンスツールを使用してカーネルのプロファイリングを行う方法について説明します。カーネルプロファイリングは、Oracle Developer Studio ソフトウェアを Oracle Solaris 10 または Oracle Solaris 11 上で実行している場合に使用できます。カーネルプロファイリングは Linux システムでは*利用できません*。

この章では、次の内容について説明します。

- 253 ページの「カーネル実験」
- 253 ページの「カーネルプロファイリング用のシステムの設定」
- 254 ページの「 er_kernel ユーティリティーの実行」
- 259 ページの「カーネルプロファイルの分析」

カーネル実験

パフォーマンスアナライザを使用するか、er_kernel ユーティリティーを使用して、カーネルプロファイルを記録できます。パフォーマンスアナライザでカーネルをプロファイルするときには、er_kernel ユーティリティーもバックグラウンドで実行しています。

er_kernel ユーティリティーは、Oracle Solaris オペレーティングシステムに組み込まれている包括的な動的トレース機能である DTrace を使用します。

er_kernel ユーティリティーは、カーネルプロファイルデータを取り込み、collect で記録されたユーザー実験に似た形式で、パフォーマンスアナライザ実験としてデータを記録します。この実験は、er_print ユーティリティーまたはパフォーマンスアナライザによって処理できます。カーネル実験は、関数データ、呼び出し元-呼び出し先のデータ、命令レベルのデータ、およびタイムラインを示すことができますが、ほとんどの Oracle Solaris モジュールが行番号テーブルを保持していないため、ソース行データを示すことはできません。

カーネルプロファイリング用のシステムの設定

er_kernel ユーティリティーを使用してカーネルプロファイリングを行う前に、DTrace へのアクセスを設定しておく必要があります。

通常、DTrace は root ユーザーだけに制限されています。root 以外のユーザーとして er_kernel ユーティリティーを実行するには、特別な権限の割り当てを受けて、sys グループのメンバーになる必要があります。必要な権限を割り当てるには、ファイル /etc/user_attr で自分のユーザー名の行を次のように編集します。

username::::defaultpriv=basic,dtrace_kernel,dtrace_proc

自分自身を sys グループに追加するには、自分のユーザー名を /etc/group ファイル内の sys行に追加します。

er_kernel ユーティリティーの実行

er_kernel ユーティリティーを実行すると、カーネルのみ、またはカーネルと実行中の負荷の両方をプロファイリングできます。詳細な説明については、er_kernel(1) のマニュアルページを参照してください。

使用方法のメッセージを表示するには、er_kernel コマンドを引数なしで実行します。

er_kernel オプション -p on はデフォルトで使用されるため、明示的に指定する必要はありません。

er_kernel ユーティリティーへの -p on 引数を、高分解能プロファイルの場合は -p high に、低分解能プロファイルの場合は -p low に置き換えることができます。負荷の実行に 2 ～ 20 分を要すると思われる場合は、デフォルトの時間プロファイルが適切です。実行に要する時間が 2 分未満と思われる場合は -p high を使用し、20 分を超えると思われる場合は -p low を使用します。

-t *所要時間* 引数を追加でき、これを追加すると er_kernel ユーティリティーは *所要時間* で指定された時間に従って自動的に終了します。

-t *所要時間*は、m (分) または s (秒) のサフィックスを付けた単数で指定できます。これは、実験を強制終了するまでの時間 (分または秒) を示します。デフォルトでは、所要時間は秒です。*所要時間* はハイフンで区切られた 2 つの数で指定することもできます。これは、1 つ目の時間が経過するまでデータ収集を停止し、そして、データ収集を始める時間を示しています。2 つ目の時間が経過すると、データ収集が終了されます。2 つ目の時間がゼロの場合、初めてプログラムが停止したあと、そのプログラムの実行の終わりまで、データの収集が実行されます。実験が終了しても、ターゲットプロセスは最後まで実行できます。

期間または間隔が指定されていない場合、er_kernel は停止するまで実行されます。終了させるには、Ctrl-C (SIGINT) を押します。あるいは、kill コマンドを使用し、SIGINT、SIGQUIT、または SIGTERM を er_kernel プロセスに送信しても終了させることもできます。er_kernel プロセスは、これらの信号のいずれかを受け取ると、実験を終了し (-A off が指定されていないかぎり)、er_archive を実行します。er_archive ユーティリティーは、実験で参照されている共有オブジェクトのリストを読み取り、それぞれについてアーカイブファイルを構築します。

-x オプションは、プロファイルイベントをアイドル状態の CPU から除外するために使用でき、デフォルトで on に設定されているためイベントが記録されません。-x off を設定してアイドル状態の CPU からのプロファイルイベントを記録すると、すべての CPU 時間の理由が明らかになります。

-v引数を追加すると、実行に関するより多くの情報を画面に出力できます。-n 引数を使用すると、実際には何も記録せずに、記録される実験のプレビューを表示できます。

デフォルトでは、er_kernel ユーティリティーによって生成される実験の名前は ktest.1.er で、続けて実験が生成されると番号が順に増えていきます。-o *experiment-name* オプションを使用して、実験の名前を変更できます。あるいは、-O *file* オプションを使用して、指定したファイルのあとに er_kernel 自体の出力を付加することもできます。ただし、生成された負荷からの出力はリダイレクトされません。出力オプションの詳細については、er_kernel(1) のマニュアルページを参照してください。

▼ er_kernel を使用してカーネルをプロファイルする方法

1. **次のように入力し、実験を収集します。**

   ```
   % er_kernel -p on
   ```

2. **別のシェルでロードを実行します。**

3. **負荷が完了したら、>Ctrl-C キーを押して er_kernel ユーティリティーを終了します。**

4. **デフォルトでは ktest.1.er という名前の結果の実験を、パフォーマンスアナライザまたは er_print ユーティリティーにロードします。**

   カーネルのクロックプロファイリングによって 2 つのメトリックが生成されます。カーネルの親実験に記録されるクロックプロファイルイベントに対する「KCPU サイクル」(メトリック名 kcycles) と、CPU がユーザーモードのときにユーザープロセスのサブ実験に記録されるクロックプロファイルイベントに対する「KUCPU サイクル」(メトリック名 kucycles) です。

   パフォーマンスアナライザでは、これらのメトリックがカーネル関数については「関数」ビュー、呼び出し先-呼び出し元については「呼び出し元-呼び出し先」ビュー、命令については「逆アセンブリ」ビューに表示されます。「ソース」ビューにはデータは表示されません。カーネルモジュールは、通常、出荷時点ではファイルおよび行シンボルテーブル情報 (スタブ) を含んでいないからです。

▼ er_kernel を使用してロード状態でプロファイルする方法

プログラムでもスクリプトでも、ロードとして使用する単一のコマンドがある場合、次のようにします。

1. **次のように入力し、実験を収集します。**

   ```
   % er_kernel -p on load
   ```

   load がスクリプトの場合は、生成されたコマンドがあればそれらの終了を待ってから終了するようにしてください。そうしないと、実験が途中で終了する可能性があります。

   er_kernel ユーティリティーは子プロセスに対してフォークし、休眠期間中に一時停止します。子プロセスは続いて、指定されたロードを実行します。負荷が終了すると、er_kernel ユーティリティーは再び休眠期間だけ一時停止し、そのあと終了します。

   休眠期間の長さは、er_kernel コマンドへの -q 引数によって秒単位で指定できます。

2. **次のように入力し、実験を解析します。**

   ```
   % analyzer ktest.1.er
   ```

   実験は、負荷の実行中、およびその前後の休眠期間での Oracle Solaris カーネルの動作を示します。カーネルプロファイルに表示される内容の詳細は、259 ページの「カーネルプロファイルの分析」を参照してください。

カーネルのハードウェアカウンタオーバーフローのプロファイリング

er_kernel ユーティリティーは、DTrace の cpc プロバイダを使ってカーネルのハードウェアカウンタオーバーフロープロファイルを収集できますが、このプロバイダは、Oracle Solaris 11 を実行しているシステム上でのみ使用できます。

カーネルのハードウェアカウンタオーバーフロープロファイリングを実行するには、collect コマンドの場合と同様に、-h オプションを er_kernel コマンドで使用します。重要な違いとして、collect コマンドは、対応するクロックプロファイル速度とほぼ一致するようにカウンタオーバーフロー値を調整します。対照的に、er_kernel は生の静的オーバーフロー値を選択しますが、特に raw カウンタ名の場合、これは適切でないことがあります。その結果、データ収集レートが低すぎたり (データのノイズが増加)、高すぎたり (測定されるパフォーマンス動作にひずみが発生) する場合があります。別名を使用すること、カウンタオーバーフロー値を 10 倍に変更しても観測されるパフォーマンス動作が著しく変化しないことを確認すること、またはその両方をお勧めします。

collect コマンドと同様に、明示的な -p off 引数が指定されていない場合は、クロックベースのプロファイリングがデフォルトで有効になります。そのチップについてのデフォルトカウンタセットを高頻度または低頻度で要求する -h high または -h low が指定された場合、デフォルトのクロックプロファイリングも high または low に設定され、明示的な -p 引数が推奨されます。

er_kernel -h コマンドは、DTrace cpc プロバイダを使用して、ハードウェアカウンタオーバーフロープロファイルを収集します。ハードウェアカウンタプロファイリングは、Oracle Solaris 11 よ

り前のシステム上では使用できません。チップ上のオーバーフローのメカニズムにより、どのカウンタがオーバーフローしたかをカーネルが認識できる場合は、チップによって提供される数のカウンタを使用できます。それ以外の場合、指定できるカウンタは 1 つだけです。

データ領域プロファイリングは、DTrace バージョン 1.8 以降を実行している SPARC システム上で、精密カウンタについてのみサポートされます。サポートされていないシステム上でリクエストされた場合は、データ領域のフラグが無視されますが、実験は引き続き実行されます。

システムのハードウェアカウンタメカニズムは、ユーザープロファイリングのために複数のプロセスで使用できますが、いずれかのユーザープロセス、cputrack、または別の er_kernel がこのメカニズムを使用している場合、カーネルプロファイリングには使用できません。その場合、er_kernel は「HW カウンタは一時的に使用できません。システムプロファイリングに使用されている可能性があります。」と報告します。

ハードウェアカウンタオーバーフローのプロファイリングをサポートするプロセッサを備えたマシンでハードウェアカウンタを表示するには、er_kernel –h コマンドをほかの引数なしで実行します。

チップ上のオーバーフローのメカニズムにより、どのカウンタがオーバーフローしたかをカーネルが知ることができる場合には、チップが提供するすべてのカウンタのプロファイリングを行えますが、それ以外の場合、指定できるカウンタは 1 つだけです。er_kernel -h の出力には、複数のカウンタを使用できるかどうかを示すために、「最大 4 つの HW カウンタを使用する HW カウンタプロファイリングを指定できます」といったメッセージが表示されます。

ハードウェアカウンタプロファイリングの詳細は、26 ページの「ハードウェアカウンタプロファイリングデータ」および64 ページの「collect -h を使用したハードウェアカウンタプロファイリング」を参照してください。

ハードウェアカウンタオーバーフローのプロファイリングの詳細については、er_print のマニュアルページも参照してください。

カーネルおよびユーザープロセスのプロファイリング

er_kernel ユーティリティーでは、カーネルおよびアプリケーションのプロファイリングを実行できます。-F オプションを使用すると、アプリケーションのプロセスを追跡し、そのデータを記録するかどうかを制御できます。er_kernel を使用して負荷を生成するときに -F を指定しない場合、負荷は追跡されます。-F を指定した場合、負荷は -F 式と一致していれば追跡されます。

-F on または -F all オプションを使用すると、er_kernel は、カーネルのほかに、すべてのアプリケーションプロセスに関する実験を記録します。er_kernel 実験の収集中に検出されたユーザープロセスが追跡され、追跡されたプロセスごとにサブ実験が作成されます。

root ユーザー以外で er_kernel を実行した場合、非特権ユーザーは通常ほかのユーザーのプロセスについて何も読み取ることができないため、多くのサブ実験は記録されない可能性があります。

十分な権限がある場合、ユーザープロセスのデータはプロセスがユーザーモードのときだけ記録され、ユーザー呼び出しスタックだけが記録されます。追跡された各プロセスのサブ実験には、kucycles メトリックのデータが含まれています。サブ実験には、*_process-name*_PID_*process-pid*.1.er の形式を使用して名前が付けられます。たとえば、sshd プロセスに関して実行された実験には、_sshd_PID_1264.1.er という名前が付けられることがあります。

一部のユーザープロセスだけを追跡するには、-F =*regexp* を使用して正規表現を指定し、正規表現に一致する名前または PID を持つプロセスに関する実験を記録できます。

たとえば、er_kernel -F =synprog では、synprog というプログラムのプロセスが追跡されます。

/proc ファイルシステムから er_kernel によって読み取られるプロセス名は、OS によって最大 15 文字まで (および 0 バイト) に切り捨てられます。そのように切り捨てられたプロセス名に一致するようにパターンを指定するべきです。

正規表現については、regexp(5) のマニュアルページを参照してください。

デフォルトでは、-F off オプションが設定されるため、er_kernel はユーザープロセスのプロファイリングを実行しません。

注記 - er_kernel の -F オプションは、collect の -F オプションとは異なります。collect -F コマンドは、コマンド行で指定したターゲットによって作成されたプロセスだけを追跡する場合に使用し、er_kernel -F は、システムで現在実行されているすべてのプロセスを追跡する場合に使用します。

カーネルとロードを同時にプロファイリングするための代替方法

er_kernel -F=*regexp* の使用のほかに、*load* の代わりに collect *load* のターゲットを使用してer_kernel を実行すると、カーネルとロードを同時にプロファイルできます。collect および er_kernel のいずれかを指定するだけで、ハードウェアカウンタを含められます。er_kernel がハードウェアカウンタを使用している場合、collect コマンドは使用できません。

この方法の利点は、CPU 上でユーザープロセスが実行していないときにそのプロセスに関するデータを収集するという点にあり、これに対して、er_kernel で収集されるユーザー実験では、ユーザー CPU 時間とシステム CPU 時間しか含められません。さらに、collect を使用した場合は、OpenMP および Java プロファイリングのデータをユーザーモードで取得されます。er_kernel を使用した場合は、どちらもマシンモードでしか取得できず、Java 実験での HotSpot コンパイルに関する情報は得られません。

▼ カーネルとロードを同時にプロファイルする方法

1. **er_kernel コマンドと collectコマンドの両方を次のように入力することにより、カーネルプロファイルとユーザープロファイルの両方を収集します。**

   ```
   % er_kernel collect load
   ```

2. **2 つのプロファイルを一緒に解析するには、次のように入力します。**

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

   パフォーマンスアナライザで表示されるデータは、ktest.1.er からのカーネルプロファイルと test.1.er からのユーザープロファイルの両方を示します。「タイムライン」ビューを使用すると、2 つの実験間の相関関係がわかります。

注記 - スクリプトを負荷として使用し、スクリプトのさまざまな部分のプロファイリングを個別に行うには、スクリプト内の各種コマンドの前に collect コマンドと適切な引数を付加します。

カーネルプロファイルの分析

カーネルの親実験には、kcycles メトリックのデータが含まれています。CPU がシステムモードのときは、カーネル呼び出しスタックが記録されます。CPU がアイドル状態のときは、擬似関数 <IDLE> のシングルフレーム呼び出しスタックが記録されます。CPU がユーザーモードのときは、擬似関数 <*process-name*_PID_*process-pid*> に属するシングルフレーム呼び出しスタックが記録されます。カーネル実験には、ユーザープロセスに関する呼び出しスタック情報は記録されません。

カーネル親実験の擬似関数 <INCONSISTENT_PID> は、不明な理由によって矛盾するプロセス ID を持つ DTrace イベントが生成されたかどうかを示します。

-F を使用してユーザープロセスの追跡を指定した場合、追跡された各プロセスのサブ実験には、kucycles メトリックのデータが含まれます。ユーザーレベルの呼び出しスタックは、そのプロセスがユーザーモードで実行されていたすべての時間プロファイルイベントに関して記録されます。

「プロセス」ビューと「タイムライン」ビューのフィルタを使用して、関心のある PID まで絞り込むことができます。

索引

数字・記号

@plt 関数, 196

あ

アウトライン関数, 212, 234
アドレス空間、テキスト領域とデータ領域, 208
アナライザ *参照* パフォーマンスアナライザ
一意でない関数名, 209
イベント
　「タイムライン」ビューに表示された, 107
イベント状態, 109
イベント密度, 109
インデックスオブジェクト, 160
　定義, 160
　リスト, 160
インデックスオブジェクトのメトリック
　er_print ユーティリティーでのリストの表示, 170
インデックスオブジェクトビュー, 113
インデックス行, 223
　er_print ユーティリティー, 153, 154
　「逆アセンブリ」タブ, 230
　「逆アセンブリ」ビュー, 120
　「ソース」タブ, 223, 230
　「ソース」ビュー, 110
インデックス行、特殊
　HotSpot コンパイル済み命令, 236
　Java ネイティブメソッド, 237
　アウトライン関数, 234
　コンパイラ生成の本体関数, 235
　行番号なしの命令, 236
インライン関数, 211
エントリポイント, 代替, Fortran 関数, 210
オーバーフロー値、ハードウェアカウンタ *参照* ハードウェアカウンタのオーバーフロー値
オプション、コマンド行、er_print ユーティリティー, 142

か

カーネルプロファイリング
　クロックプロファイリング, 255
　システムの設定, 253
　収集およびユーザープロセス, 258
　データの種類, 25, 253
　ハードウェアカウンタオーバーフロー, 256
　プロファイルの解析, 259
　ユーザープロセス, 257
概要データ、er_print ユーティリティーでの出力, 173
カスタムフィルタ, 126
間隔、標本収集 *参照* 標本収集の間隔
間隔、プロファイリング *参照* プロファイリング間隔
環境変数
　JAVA_PATH, 57
　JDK_HOME, 57
　LD_BIND_NOW, 43
　LD_LIBRARY_PATH, 91
　LD_PRELOAD, 90
　PATH, 57
　SP_COLLECTOR_NO_OMP, 57
　SP_COLLECTOR_STACKBUFSZ, 54, 215
　SP_COLLECTOR_USE_JAVA_OPTIONS, 58
　VampirTrace, 32
　VT_BUFFER_SIZE, 33, 94
　VT_MAX_FLUSHES, 33
　VT_STACKS, 32, 94
　VT_UNIFY, 95
　VT_VERBOSE, 33, 94
関数
　@plt, 196

<JVM-System>, 214
<no Java callstack recorded>, 215
<Total>, 215
<Truncated-stack>, 215
<Unknown>, 213
MPI、トレースされる, 33
アウトライン, 212, 234
アドレスの変動, 208
一意でない, 名前, 209
インライン, 211
クローン, 211, 237
コレクタ API, 50, 53
システムライブラリ、コレクタによる割り込み, 46
静的、重複名を持つ, 209
静的, ストリップ済み共有ライブラリ, 238
静的、ストリップ済み共有ライブラリ, 210
大域, 209
代替エントリポイント (Fortran), 210
定義, 209
動的にコンパイルされる, 53, 213, 235
別名を持つ, 209
ラッパー, 210
ロードオブジェクト内のアドレス, 209
関数の呼び出し
共有オブジェクト間, 196
シングルスレッドプログラムでの, 195
「関数」ビュー, 107
関数名、er_print ユーティリティーでの長短いずれかの形式の選択, 171
関数呼び出し
再帰、メトリックの割り当て, 39
関数リスト
er_print ユーティリティーでの出力, 147
コンパイラ生成の本体関数, 235
ソート順序、er_print ユーティリティーでの指定, 148
関数リストのメトリック
.er.rc ファイルにおけるデフォルト値の選択, 175
.er.rc ファイルにおけるデフォルトのソート順序の設定, 176
er_print ユーティリティーでの選択, 147
er_print ユーティリティーでのリストの表示, 169
「関数を表示/非表示/API のみ」ダイアログボックス、「ライブラリとクラスの可視性」ダイアログボックスを参照, 124
関数 PC、アグリゲーション, 119, 119
キーワード、メトリック、er_print ユーティリティー, 145
擬似関数、OpenMP 呼び出しスタック内の, 203
逆アセンブリコード, 注釈付き
クローン関数, 238
逆アセンブリコード、注釈付き
er_print ユーティリティーでの強調表示しきい値の設定, 156
er_print ユーティリティーでの出力, 154
er_print ユーティリティーでの設定, 155
er_src ユーティリティーでの表示, 239
HotSpot コンパイル済み命令, 236
Java ネイティブメソッド, 237
st 命令と ld 命令, 239
解釈, 230
クローン関数, 211, 238
実行可能ファイルの場所, 60
説明, 229
ハードウェアカウンタメトリックの帰属, 233
分岐先, 239
包括的メトリック, 239
命令発行の依存関係, 231
メトリックの形式, 229
「逆アセンブリ」ビュー, 119
「競合」ビュー, 121
共通部分式の削除, 225
共有オブジェクト、関数の呼び出し, 196
「行」ビュー, 119
クローン関数, 211, 237, 237
クロックプロファイリング
collect コマンドを使用したデータの収集, 63
dbx でのデータ収集, 83
gethrtime および gethrvtime との比較, 191
オーバーヘッドによるひずみ, 191
間隔 *参照* プロファイリング間隔
定義, 22
プロファイルパケット内のデータ, 189
メトリック, 23, 190
メトリックの精度, 192
「経過時間」ビュー, 117
検索パス設定, 137
構成設定, 138
高速トラップ, 197
構文
er_archive ユーティリティー, 249

er_export ユーティリティー, 251
er_html ユーティリティー, 251
er_print ユーティリティー, 142
er_src ユーティリティー, 239
高メトリック値
注釈付き逆アセンブリコード内, 156
注釈付きソースコード内, 155
コレクタ
API、プログラムでの使用, 50, 51
collect コマンドでの実行, 62
dbx での実行, 82
dbx での無効化, 86
dbx での有効化, 87
実行中のプロセスへの接続, 89
定義, 18, 21, 21
コレクタによるシステムライブラリ関数への割り込み処理, 46
コンテキストメニューからのデータのフィルタリング
「関数」ビュー, 107
「呼び出しツリー」ビュー, 111
「呼び出し元-呼び出し先」ビュー, 112
コンパイラ解説, 110
er_print ユーティリティーでの注釈付き逆アセンブリリストの選択, 155
er_print ユーティリティーでの注釈付きソースリストの選択, 154
er_print ユーティリティーでの注釈付きのソースおよび逆アセンブリリストの選択, 156
er_src ユーティリティーでのフィルタリング, 240
インライン関数, 227
共通部分式の削除, 225
クラスの定義, 154
クローン関数, 237
説明, 224
表示されるフィルタの種類, 225
並列化, 227
ループの最適化, 226
コンパイラ生成の本体関数
名前, 235
パフォーマンスアナライザでの表示, 212, 234
コンパイラの最適化
インライン化, 227
並列化, 227
コンパイル
Java プログラミング言語, 44
「行」ビュー, 42
最適化によるプログラム解析への影響, 43
注釈付きソースと逆アセンブリのソースコード, 42
データ収集のリンク, 43
データ領域プロファイリング用, 42
デバッグシンボル情報の形式, 42

さ

サーバーのプロファイリング, 247
再帰関数呼び出し
メトリックの割り当て, 39
サイクル当たりの命令数, 145
最適化
共通部分式の削除, 225
プログラム解析への影響, 43
末尾呼び出し, 197
サブルーチン *参照* 関数
「サマリー」タブ, 119
サマリーメトリック
1 つの関数、er_print ユーティリティーでの出力, 149
すべての関数、er_print ユーティリティーでの出力, 149
サンプルコード, 20, 97
時間メトリック、精度, 107
しきい値、強調表示
注釈付き逆アセンブリコード内、er_print ユーティリティー, 156
注釈付きソースコード内、er_print ユーティリティー, 155
しきい値、同期待ちトレース
collect コマンドによる設定, 85
dbx collector での設定, 85
収集オーバーヘッドに対する影響, 194
調整, 30
定義, 29
式の文法, 177
シグナル
collect コマンドでの一時停止と再開のための使用, 77
collect コマンドによる手動標本収集での使用, 76
ハンドラへの呼び出し, 196
プロファイリング, 48

プロファイリング、dbx から collect コマンドに渡す, 77
シグナルハンドラ
コレクタによってインストール, 49
コレクタによってインストールされる, 196
ユーザープログラム, 49
システム全体のプロファイリング, 257
実験, 58
参照 実験ディレクトリ
er_print ユーティリティーでの一覧表示, 164
er_print ユーティリティーでのヘッダー情報, 172
Java および OpenMP のモードの設定, 171
移動, 60, 244
格納場所, 79, 88
グループ, 59
現在のパスの追加, 158
コピー, 243
サイズの制限, 73
削除, 244
定義, 58
データアグリゲーション, 98
デフォルト名, 59
のサイズ制限, 88
場所, 58
パスの接頭辞の再マッピング, 158
派生、ロード, 98
必要なストレージ容量、概算, 60
開く, 98
ファイル検索パスの設定, 157
複数, 98
プレビュー, 98
プログラムからの終了, 53
命名, 59
ラベル付け, 244
ロードオブジェクトのアーカイブ, 79, 88
実験グループ, 58
collect コマンドによる名前の指定, 79
dbx での名前の指定, 88
削除, 244
作成, 59, 98
定義, 59
デフォルト名, 59
名前の制限, 59
複数, 98
プレビュー, 98
実験ごとのメトリック, 114
実験サイズの制限, 73
実験ディレクトリ
collect コマンドでの指定, 79
dbx での指定, 88
デフォルト, 58
実験データのフィルタリング, 124, 125
er_print, 165
カスタムフィルタ, 126
ラベルの使用, 127
実験で使用されるソースおよびオブジェクトファイル, 137
実験の移動, 60, 244
実験のコピー, 243
実験のサイズ制限, 88
実験の比較, 129
比較形式の設定, 129
実験の命名, 59
実験のラベル付け
フィルタリング, 127
実験、派生
読み取り用のモードの設定、er_print ユーティリティー, 176
「実験」ビュー, 121
実験へのラベル付け, 244
実験へのロードオブジェクトのアーカイブ, 79, 88
実験または実験グループの削除, 244
実験名, 58
dbx での指定, 88
MPI のデフォルト, 60, 94
カーネルプロファイリング, 60, 187
制限, 59
デフォルト, 59
「実験を開く」ダイアログボックス, 138
「実験 ID」ビュー, 114
実行中のプロセスへのコレクタの接続, 89
実行統計
er_print ユーティリティーでの出力, 174
関数との時間の比較, 192
出力ファイル
閉じて新規に開く、er_print ユーティリティー, 170
閉じる、er_print ユーティリティー, 170
シングルスレッドプログラムの実行, 195
シンボルテーブル、ロードオブジェクト, 208
スタック深度, 54
スタックフレーム

定義, 196
トラップハンドラから, 197
末尾呼び出しの最適化での再利用, 197
スレッド
er_print ユーティリティーでの選択, 167
作成, 198
選択内容の一覧表示、er_print ユーティリティー, 164
メトリック, 113
ワーカー, 198
「スレッド」ビュー, 113
制限 *参照* 制限事項
制限事項
Java プロファイリング, 57
実験グループ名, 59
実験名, 59
派生プロセスのデータ収集, 57
プロファイリング間隔の値, 54
静的関数
重複名, 209
ストリップ済み共有ライブラリ, 210, 238
静的リンク
データ収集への影響, 43
設定, 138
設定の .er.rc へのエクスポート, 139
「選択の詳細」ウィンドウ, 102
相関関係、メトリックに対する影響, 191
「ソース/逆アセンブリ」タブ
「設定」, 135
「ソース/逆アセンブリ」ビュー, 120
ソース行、er_print ユーティリティーでの整列済みリスト, 157
ソースコード、コンパイラ解説, 110
ソースコード, 注釈付き
コンパイラ生成の本体関数, 235
ソースコード、注釈付き
er_print ユーティリティーでの強調表示しきい値の設定, 155
er_print ユーティリティーでのコンパイラの解説クラスの設定, 154
er_print ユーティリティーでの出力, 153
er_src ユーティリティーでの表示, 239
アウトライン関数, 234
インデックス行, 223
解釈, 228
クローン関数, 237
クローン関数の, 211
コンパイラ解説, 224
説明, 222, 228
ソースと注釈との識別, 223
ソースファイルの場所, 60
中間ファイルの使用, 207
パフォーマンスアナライザでの表示, 222
メトリックの形式, 229
行番号なしの命令, 236
ソースと逆アセンブリコード、注釈付き
er_print ユーティリティーでの設定, 156
「ソース」ビュー, 109
ソート順序
関数リスト、er_print ユーティリティーでの指定, 148
属性メトリック
再帰の影響, 40
使用, 37
図示, 38
定義, 36

た

代替ソースコンテキスト, 153
タイムライン設定, 134
タイムライン内のクロックプロファイリング, 108
タイムライン内のトレースデータ, 108
タイムライン内の CPU 使用率標本, 108
タイムラインのヒープサイズ, 108
「タイムライン」ビュー, 107
「選択の詳細」ウィンドウ, 103
中間ファイル、注釈付きソースリストとして使用, 207
注釈付き逆アセンブリコード *参照* 逆アセンブリコード, 注釈付き
注釈付きソースコード *参照* ソースコード、注釈付き
チュートリアル, 20, 97
ツールバー, 102
「ツール」メニュー, 101
ディスク容量、実験用の概算, 60
データオブジェクト
<Scalars> 記述子, 218
<Total>記述子, 218
記述子, 217
スコープ, 217
定義, 217
ハードウェアカウンタオーバーフロー実験の, 158

レイアウト, 115
「データオブジェクト」ビュー, 116
データ競合
詳細情報, 163
リスト, 163
「データサイズ」ビュー, 117
データ収集
collect コマンドの使用, 62
collect コマンド用の一時停止, 77
collect コマンド用の再開, 77
dbx での一時停止, 87
dbx での再開, 87
dbx での無効化, 86
dbx での有効化, 87
dbx の使用, 82
MPI プログラム、collect コマンドの使用法, 93
MPI プログラムから, 92
セグメント例外, 44
動的メモリー割り当ての影響, 44
の比率, 61
のプログラム制御, 50
プログラムからの一時停止, 53
プログラムからの再開, 53
プログラムからの制御, 50
プログラムからの無効化, 53
プログラムの準備, 44
データ収集時のセグメント例外, 44
データ収集の一時停止
collect コマンド用の, 77
dbx での, 87
プログラムから, 53
データ収集の再開
collect コマンド用の, 77
dbx での, 87
プログラムから, 53
データの種類, 22
MPI トレース, 32
クロックプロファイリング, 22
同期待ちトレース, 29
ハードウェアカウンタプロファイリング, 26
ヒープトレース, 30
データ派生メトリック
er_print ユーティリティーでのリストの表示, 169
データ表示方法
設定オプション, 131
データ領域プロファイリング, 193
データオブジェクト, 217
「データレイアウト」ビュー, 115
デッドロック
詳細情報, 163
リスト, 163
「デッドロック」ビュー, 121
デフォルト
デフォルト値ファイルでの設定, 175
デフォルトのメトリック, 107
デフォルトパス, 157
「デュアルソース」ビュー, 121
同期遅延イベント
定義, 29
プロファイルパケット内のデータ, 194
同期遅延イベントカウント
定義されたメトリック, 30
同期待ち時間
定義, 29, 194
メトリック、定義, 30
同期待ちトレース
collect コマンドによるデータ収集, 67
dbx でのデータの収集, 85
er_sync.so のプリロード, 90
しきい値 *参照* しきい値、同期待ちトレース
定義, 29
プロファイルパケット内のデータ, 194
待ち時間, 29, 194
メトリック, 30
「統計」ビュー, 121
動的にコンパイルされる関数
定義, 213, 235
のコレクタ API, 53
トラップ, 197

な

入力ファイル
er_print ユーティリティー, 177
er_print ユーティリティーでの終了, 177, 177
ネットワーク接続されたディスク, 58

は

バージョン情報

collect コマンドの, 81
er_cp ユーティリティー, 244
er_mv ユーティリティー, 244
er_print ユーティリティー, 177
er_rm ユーティリティー, 244
er_src ユーティリティー, 241
ハードウェアカウンタ
dbx collector コマンドによる選択, 84
一覧の取得, 63
オーバーフロー値, 26
カウンタ名, 64
定義, 26
データオブジェクトとメトリック, 158
の一覧の取得, 84
リストの説明, 28
ハードウェアカウンタオーバーフロー値
小さすぎたり大きすぎたりする場合の影響, 192
ハードウェアカウンタオーバーフローのプロファイリング
カーネル , 256
ハードウェアカウンタオーバーフロープロファイリング
プロファイルパケット内のデータ, 192
ハードウェアカウンタ属性オプション, 65
ハードウェアカウンタの一覧
dbx collector コマンドによる取得, 84
ハードウェアカウンタのオーバーフロー値
dbx での設定, 84
定義, 26
ハードウェアカウンタの別名, 27
ハードウェアカウンタプロファイリング
collect コマンドによるデータ収集, 64
dbx によるデータ収集, 84
定義, 26
デフォルトカウンタ, 64
ハードウェアカウンタメトリック、「データオブジェクト」ビューに表示される, 116
ハードウェアカウンタライブラリ、libcpc.so, 56
ハードウェアカウンタリスト
collect コマンドによる取得, 63
raw カウンタ, 29
フィールドの説明, 28
別名が設定されたカウンタ, 28
排他的メトリック
PLT 命令, 196
計算方法, 195
図示, 38
定義, 36
の使用, 37
パス接頭辞の再マッピング, 138
パス接頭辞のマッピング, 138
パスの接頭辞の再マッピング, 158
「パスマップ」タブ, 138
派生実験
読み取り用のモードの設定、er_print ユーティリティー, 176
ロード, 98
派生プロセス
コレクタの処理対象, 57
実験の場所, 58
実験名, 59
選択対象についてのデータ収集, 89
追跡対象すべてのデータの収集, 74
データ収集の制限事項, 57
パフォーマンスアナライザ
「CPU」ビュー, 113
「I/O」ビュー, 116
MPI タイムラインコントロール, 122
MPI チャートコントロール, 123
「MPI チャート」ビュー, 123
「OpenMP タスク」ビュー, 118
「OpenMP 並列領域」ビュー, 118
「PC」ビュー, 119
インデックスオブジェクトビュー, 113
「関数」ビュー, 107
起動, 98
「逆アセンブリ」ビュー, 119
「競合」ビュー, 121
「行」ビュー, 119
「経過時間」ビュー, 117
検索パス設定, 137
コマンド行オプション, 99
「サマリー」タブ, 119
「実験 ID」ビュー, 114
実験の記録, 98
「実験」ビュー, 121
「スレッド」ビュー, 113
「選択の詳細」ウィンドウ, 102
「ソース/逆アセンブリ」タブ, 135
「ソース/逆アセンブリ」ビュー, 120
「ソース」ビュー, 109
タイムライン設定, 134
「タイムライン」ビュー, 103, 107

ツールバー, 102
定義, 18, 97
「データオブジェクト」ビュー, 116
「データサイズ」ビュー, 117
「データレイアウト」ビュー, 115
「デッドロック」ビュー, 121
「デュアルソース」ビュー, 121
「統計」ビュー, 121
パスマップ設定, 138
表示されるビュー, 131
「秒」ビュー, 114
「標本」ビュー, 113
「プロセス」ビュー, 114
「命令頻度」ビュー, 122
「メトリック」タブ, 133
メニューバー, 101
「メモリーオブジェクト」ビュー, 115
「呼び出しツリー」ビュー, 111
「呼び出し元-呼び出し先」ビュー, 111
「ライブラリとクラスの可視性」ダイアログボックス, 124
パフォーマンスデータ、メトリックに変換, 21
パフォーマンスメトリック *参照* メトリック
ヒープトレース
collect コマンドでのデータ収集, 69
dbx でのデータ収集, 85
er_heap.so のプリロード, 90
メトリック, 30
「ヒープ」ビュー, 117
必要なストレージ容量、実験用の概算, 60
ビュー
表示のための選択, 132
「ビュー」設定, 132
「ビュー」メニュー, 101
秒
1 秒あたりに記録されたメトリック, 114
表示モード
説明, 136
「秒」ビュー, 114
標本
collect コマンドによる定期的記録, 70
collect による手動記録, 76
dbx がプロセスを停止したときの記録, 86
dbx での手動記録, 87
dbx での定期的記録, 86
er_print ユーティリティーでの選択, 166
間隔 *参照* 標本収集の間隔
記録の状況, 32
選択内容の一覧表示、er_print ユーティリティー, 164
「タイムライン」ビューに表示された, 107
定義, 32
パケットに含まれる情報, 31
プログラムからの記録, 52
メトリック, 113
標本収集間隔
dbx での設定, 86
標本収集の間隔
定義, 32
「標本」ビュー, 113
ファイルのパス, 157
ファイルのパスの追加, 158
「ファイル」メニュー, 101
フィルタ式の例, 179
「フィルタ」パネル, 125
フィルタリングとライブラリの可視性, 124
フォントサイズ
変更, 100
符号化された関数名, 235, 238
プリロード
er_heap.so, 90
er_sync.so, 90
libcollector.so, 90
フレーム、スタック *参照* スタックフレーム
プログラムカウンタのメトリック, 119
プログラムカウンタ (PC)、定義, 195
プログラム構造、呼び出しスタックのアドレスのマッピング, 208
プログラムの実行
共有オブジェクトと関数の呼び出し, 196
シグナル処理, 196
シングルスレッド, 195
トラップ, 197
末尾呼び出しの最適化, 197
明示的なマルチスレッド化, 198
呼び出しスタックの説明, 195
プログラムリンケージテーブル (PLT), 196
プロセス
プロセスごとのメトリック, 114
プロセスのアドレス空間のテキスト領域とデータ領域, 208

「プロセス」ビュー, 114
プロファイリング間隔
collect コマンドによる設定, 63, 83
dbx collector コマンドによる設定, 83
値に関する制限事項, 54
実験のサイズ、への影響, 61
定義, 22
プロファイリング、定義, 21
プロファイルアプリケーション
プレビューコマンド, 128
プロファイルパケット
クロックデータ, 189
同期待ちトレースデータ, 194
のサイズ, 61
ハードウェアカウンタオーバーフローデータ, 192
分岐先, 239
並列実行
ディレクティブ, 227
別名が設定されたハードウェアカウンタ, 28
別名を持つ関数, 209
「ヘルプ」メニュー, 101
包括的メトリック
PLT 命令, 196
アウトライン関数, 239
計算方法, 195
再帰の影響, 40
図示, 38
定義, 36
の使用, 37
本体関数、コンパイラ生成の
名前, 235
パフォーマンスアナライザでの表示, 212, 234

ま

マイクロステート, 103
切り替え, 197
メトリックへの関連, 190
マシンモデル, 133, 217
「マシンモデルのロード」ボタン, 133
待ち時間 *参照* 同期待ち時間
末尾呼び出しの最適化, 197, 197
マルチスレッドアプリケーション
へのコレクタの接続, 89
マルチスレッド化
明示的, 198
明示的なマルチスレッド化, 198
命令当たりのサイクル数, 145
命令発行
グループ化、注釈付き逆アセンブリへの影響, 231
遅延, 232
命令頻度
er_print ユーティリティーでのリストの出力, 173
「命令頻度」ビュー, 122
メソッド *参照* 関数
メトリック
MPI トレース, 35
関数リスト *参照* 関数リストのメトリック
クロックプロファイリング, 23, 190
時間の精度, 107
しきい値, 120
しきい値、設定, 110
相関関係の影響, 191
ソース行のための解釈, 228
属性, 112 *参照* 属性メトリック
タイミング, 23
定義, 21
デフォルト, 107
同期待ちトレース, 30
ハードウェアカウンタ、命令への帰属, 233
排他的 *参照* 排他的メトリック
ヒープトレース, 30
包括的 *参照* 包括的メトリック
包括的および排他的, 107, 112
命令の解釈, 230
メモリー割り当て, 30
「メトリック」タブ, 133
「メトリック」メニュー, 101
メモリーオブジェクト
定義
er_print を使用, 161
パフォーマンスアナライザ, 133
「メモリーオブジェクト」ビュー, 115
メモリーリーク、定義, 31
メモリー領域プロファイリング, 193
メモリー割り当て, 30
およびリーク, 117
データ収集への影響, 44

や

ユーザーモード, 136

呼び出しスタック, 117
定義, 195
展開, 195
不完全な展開, 207
プログラム構造へのアドレスのマッピング, 208
末尾呼び出しの最適化の影響, 198
呼び出しスタックの展開, 195
呼び出しスタックのナビゲーション, 103
呼び出しスタックフラグメント, 112
「呼び出しツリー」ビュー, 111
コンテキストメニューからのデータのフィルタリング, 111
呼び出し元-呼び出し先
属性、定義, 36
「呼び出し元-呼び出し先」ビュー, 111
呼び出し元-呼び出し先メトリック
er_print ユーティリティーでの 1 つの関数の出力, 150
er_print ユーティリティーでの出力, 149
er_print ユーティリティーでのリストの表示, 169
「呼び出し元/呼び出し回数」パネル, 103

ら

ライブラリ
collectorAPI.h, 52
libcollector.so, 50, 90
libcpc.so, 48, 56
MPI, 48
システム, 46
ストリップ済み共有、および静的関数, 210, 238
への割り込み, 46
「ライブラリとクラスの可視性」ダイアログボックス, 124
ラッパー関数, 210
リーク、メモリー、定義, 31
リーフ PC、定義, 195
リモートパフォーマンスアナライザ, 130
クライアントの要件, 130
サーバーの要件, 130
リモートホストへの接続, 131
リモートホストへの接続
, 131
ループの最適化, 226
ロードオブジェクト
er_print ユーティリティーでの選択, 169
er_print ユーティリティーでのリストの出力, 173
関数のアドレス, 209
シンボルテーブル, 208
選択内容の一覧表示、er_print ユーティリティー, 167
定義, 208
内容, 208
のレイアウトの書き込み, 159

A

addpath コマンド, 158
analyzer コマンド
JVM オプション (-J) オプション, 100
JVM パス (-j) オプション, 99
冗長 (-v) オプション, 100
データ収集オプション, 99
バージョン (-V) オプション, 101
フォントサイズ (-f) オプション, 100
ヘルプ (-h) オプション, 101
API、コレクタ, 50

C

collect コマンド
-I オプション, 70
-N オプション, 70
-P オプション, 80
-C オプション, 186
exec 後のターゲット停止 (-x) オプション, 77
-i オプション, 69
Java バージョン (-j) オプション, 75
MPI トレース (-m) オプション, 73
-M オプション, 72
ppgsz コマンドとの組み合わせ, 92
-o オプション, 78
アーカイブ (-A) オプション, 79
オプションの一覧表示, 63
カウントデータの記録 (-c) オプション, 69
クロックプロファイリング (-p) オプション, 63
構文, 62
実験グループ (-g) オプション, 79
実験制御オプション, 73

実験ディレクトリ (-d) オプション, 79
実験名 (-o) オプション, 80
出力オプション, 78
冗長 (-v) オプション, 81
その他のオプション, 80
定期的標本収集 (-S) オプション, 70
データ競合の検出 (-r) オプション, 71
データ記録の一時停止と再開 (-y) オプション, 77
データ収集オプション, 63
データ収集の時間範囲 (-t) オプション, 76
データ制限 (-L) オプション, 73
同期待ちトレース (-s) オプション, 67
ドライラン (-n) オプション, 81, 81
によるデータ収集, 62
バージョン (-V) オプション, 81
ハードウェアカウンタプロファイリング (-h) オプション, 64
派生プロセス追跡 (-F) オプション, 74
ヒープトレース (-H) オプション, 69
標本ポイント記録 (-l) オプション, 76
collectorAPI.h, 52
コレクタとの C/C++ インタフェースの一部として, 51
config.xml ファイル, 138
CPI メトリック, 145
CPU
CPU ごとのメトリック, 113
「CPU」ビューでの一覧表示, 113
er_print ユーティリティーでの選択, 167
選択内容の一覧表示、er_print ユーティリティー, 164
「CPU」ビュー, 113

D

data_layout コマンド, 159
data_objects コマンド, 158
data_single コマンド, 159
dbx collector サブコマンド
archive, 88
dbxsample, 86
disable, 86
enable_once (廃止) , 89
enable, 87
hwprofile, 84
limit, 88
pause, 87
profile, 83
quit (廃止), 89
resume, 87
sample record, 87
sample, 86
show, 89
status, 89
store filename (廃止), 89
store, 88
synctrace, 85, 85
tha, 85
dbx
でのコレクタの実行, 82
ddetail コマンド, 163
deadlocks コマンド, 163
DTrace
アクセスの設定, 254
説明, 253

E

.er.rc ファイル, 139, 241
コマンド, 175, 175
場所, 174
er_archive ユーティリティー, 249
er_cp ユーティリティー, 243
er_export ユーティリティー, 251
er_heap.so、プリロード, 90
er_html ユーティリティー, 251
er_kernel ユーザー
カーネルおよびユーザープロセス, 257
er_kernel ユーティリティー, 24, 253
カーネルプロファイルの解析, 259
収集およびユーザープロセスで使用, 258
ハードウェアカウンタオーバーフローのプロファイリング, 256
er_label ユーティリティー, 244
コマンド構文, 245
時間指定, 246
使用例, 247
スクリプトでの使用, 247

er_mv ユーティリティー, 244
er_print コマンド
  addpath, 158
  allocs, 152
  appendtfile, 170
  callers-callees, 149
  cc, 156
  cmetric_list, 169
  cpu_list, 164
  cpu_select, 167
  csingle, 150
  data_layout, 159
  data_metric_list, 169
  data_objects, 158
  data_single, 159
  dcc, 155
  ddetail, 163
  deadlocks, 163
  describe, 165
  disasm, 154
  dmetrics, 175
  dsort, 176
  en_desc, 176
  exit, 177
  experiment_list, 164
  filters, 165
  fsingle, 149
  fsummary, 149
  functions, 147
  header, 172
  help, 177
  ifreq, 173
  indx_metric_list, 170
  indxobj_define, 160
  indxobj_list, 160
  indxobj, 160
  leaks, 152
  limit, 170
  lines, 157
  lsummary, 157
  lwp_list, 164
  lwp_select, 166
  metric_list, 169
  name, 171
  object_api, 168
  object_hide, 168
  object_list, 167
  object_select, 169
  object_show, 168
  objects_default, 168
  objects, 173
  outfile, 170
  overview, 173
  pathmap, 158
  pcs, 156
  procstats, 176
  psummary, 156
  quit, 177
  races, 163
  rdetail, 163
  sample_list, 164
  sample_select, 166
  scc, 154
  script, 177
  setpath, 157
  sort, 148
  source, 153
  statistics, 174
  sthresh, 155, 156
  thread_list, 164
  thread_select, 167
  version, 177, 177
  viewmode , 171
  使用例, 181
  フィルタ構文, 177
  フィルタ式のキーワード, 165
  フィルタの例, 179
  フィルタリング , 165
  メトリック, 147
er_print ユーティリティー
  構文, 142
  コマンド *参照* er_print コマンド
  コマンド行オプション, 142
  メトリックキーワード, 145
  メトリックリスト, 143
  目的, 142

er_print ユーティリティーからの蓄積された統計の出力, 176
er_print ユーティリティーでの出力の制限, 170
er_print ユーティリティーでの派生実験の読み取り用モードの設定, 176
er_rm ユーティリティー, 244
er_src ユーティリティー, 239
er_sync.so、プリロード, 90

F

Fortran
  コレクタ API, 50
  サブルーチン, 209
  代替エントリポイント, 210
Fortran 関数内の代替エントリポイント, 210

I

I/O トレース, 116
  collect コマンドによるデータ収集, 69
「I/O」ビュー, 116
indxobj_define コマンド, 160
indxobj_list コマンド, 160
indxobj コマンド, 160
IPC メトリック, 145

J

--jdkhome analyzer コマンドオプション, 99
JAVA_PATH 環境変数, 57
Java
  er_print の表示出力の設定, 171
  動的にコンパイルされるメソッド, 53, 213
  プロファイリングの制限事項, 57
Java 仮想マシンパス、analyzer コマンドオプション, 99
JDK_HOME 環境変数, 57

L

LD_LIBRARY_PATH 環境変数, 91
LD_PRELOAD 環境変数, 90
libcollector.h
  コレクタとの Java プログラミング言語インタフェースの一部として, 52
libcollector.so 共有ライブラリ
  プログラムでの使用, 50
libcollector.so共有ライブラリ
  プリロード, 90
libcpc.so、使用, 56
libfcollector.h, 51
<JVM-System> 関数, 214
<no Java callstack recorded> 関数, 215
<Scalars> データオブジェクト記述子, 218
<Total> 関数
  実行統計との時間の比較, 192
  説明, 215
<Total> データオブジェクト記述子, 218
<Truncated-stack> 関数, 215
<Unknown> 関数
  PC のマッピング, 213
  呼び出し元と呼び出し先, 214
LWP
  er_print ユーティリティーでの選択, 166
  Solaris のスレッドで作成, 198
  選択内容の一覧表示、er_print ユーティリティー, 164

M

MPI タイムラインコントロール, 122
「MPI タイムライン」ビュー, 122
MPI チャートコントロール, 123
「MPI チャート」ビュー, 123
MPI トレース, 195
  collect コマンドでのデータ収集, 73
  コレクタライブラリのプリロード, 90
  トレースされる関数, 33
  メトリック, 35
MPI プログラム
  collect コマンドによるデータ収集, 93
  実験名, 60, 94
  データの収集, 92

N

NFS, 58

O

OMP_preg コマンド, 162
OMP_task コマンド, 162
OpenMP
  er_print の表示出力の設定, 171
  インデックスオブジェクト、情報の出力, 162, 162
  実行の概要, 201
  プロファイリングデータ、マシン表現, 205
  プロファイリングの制限事項, 57
  プロファイルデータのユーザーモードの表示, 203
  メトリック, 204
  ユーザーモード呼び出しスタック, 204
OpenMP アプリケーション内のユーザーモード呼び出しスタック, 204
「OpenMP タスク」ビュー, 118
OpenMP 並列化, 227
「OpenMP 並列領域」ビュー, 118

P

PATH 環境変数, 57
pathmap コマンド, 158
PC
  er_print ユーティリティーでの整列済みリスト, 156
  PLT から, 196
  定義, 195
「PC」ビュー, 119
PLT (プログラムリンケージテーブル), 196
ppgsz コマンド, 92

R

races コマンド, 163
raw ハードウェアカウンタ, 28, 29
rdetail コマンド, 163

S

setpath コマンド, 157
setuid、使用, 50
SP_COLLECTOR_STACKBUFSZ 環境変数, 54, 215

T

TLB (トランスレーションルックアサイドバッファー) ミス, 197, 232

V

VampirTrace, 32
viewmode コマンド, 171

X

-xdebugformat、デバッグシンボル情報の形式の設定, 42