Oracle® Solaris Studio 12.4: Discover および Uncover ユーザーズガイド

Part No: E57226
2015 年 12 月

ORACLE®

Part No: E57226

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

- **概要** - バイナリのメモリー関連エラーを検出するメモリーエラー探索ツール (discover)、およびアプリケーションのコードカバレージを測定するコードカバレージツール (uncover) の使用方法について説明します。
- **対象読者** - アプリケーション開発者、システム開発者、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、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経験

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://docs.oracle.com/cd/E37069_01 で入手可能です。

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 概要

『*Oracle Solaris Studio 12.4 Discover および Uncover ユーザーズガイド*』では、次のツールの使用方法について説明します。

メモリーエラー探索ツール (discover)

メモリーエラー探索ツール (discover) ソフトウェアは、メモリーアクセスエラーを検出するための高度な開発ツールです。discover ユーティリティーは、Sun Studio 12 Update 1、Oracle Solaris Studio 12.2、Oracle Solaris Studio 12.3、または Oracle Solaris Studio 12.4 コンパイラを使用してコンパイルされたバイナリ上で機能します。Solaris 10 10/08 以上のオペレーティングシステム、Oracle Solaris 11、Oracle Enterprise Linux 5.x、または Oracle Enterprise Linux 6.x 以上を実行する SPARC ベースまたは x86 ベースのシステムで動作します。

プログラムのメモリー関連のエラーは、検出が難しいことで知られています。discover ユーティリティーを使用すると、ソースコードに存在している問題の正確な場所を指摘することによって、このようなエラーを簡単に検出できます。たとえば、プログラムが配列を割り当てたが、それを初期化せずに、ある配列の場所から読み取ろうとする場合、プログラムの動作が不安定になることがあります。discover ユーティリティーは、通常の方法でプログラムを実行するときに、この問題を検出できます。

discover によって検出されるほかのエラーには、次のものがあります。

- 非割り当てメモリーからの読み取り、および非割り当てメモリーへの書き込み
- 割り当て済み配列範囲外のメモリーへのアクセス
- 解放されたメモリーの不正使用
- 不正なメモリーブロックの解放
- メモリーリーク

discover はプログラムの実行中にメモリーアクセスエラーを動的に検出して報告するため、実行時にユーザーコードの一部が実行されない場合、その部分のエラーは報告されません。

discover ユーティリティーは簡単に使用できます。コンパイラによって準備されたすべてのバイナリは (完全に最適化されたバイナリでも)、単一のコマンドを使用して計測し、通常の方法で実行できます。discover は実行中にメモリー異常のレポートを生成します。これはテキストファイル形式で、または Web ブラウザに HTML 形式で表示できます。

コードカバレージツール (uncover)

uncover ユーティリティーは、アプリケーションのコードカバレージを計測するための簡単で使いやすいコマンド行ツールです。コードカバレージは、ソフトウェアのテストの重要な部分です。テストで実行されるコードの領域に関する情報を提供することで、テストスイートを向上させ、より多くのコードをテストできるようにします。uncover で報告されるカバレージ情報は、関数、文、基本ブロック、または命令レベルにできます。

uncover ユーティリティーは、*カバレージ外*と呼ばれる独自の機能を提供し、テストされない主要な機能領域をすばやく検出できます。ほかの種類の計測より優れた uncover コードカバレージのほかの利点は、次のとおりです。

- 未計測コードに関連する遅延がかなり少ない。
- uncover はバイナリ上で動作するため、最適化されたバイナリと併用できる。
- 出荷用バイナリを計測することによって測定が簡単にできる。アプリケーションをカバレージテスト用に異なる方法で構築する必要がない。
- uncover ユーティリティーは、バイナリの計測、テストの実行、および結果の表示を行うための簡単な手順を提供する。
- uncover ユーティリティーは、マルチスレッドおよびマルチプロセスに対して安全である。

♦ ♦ ♦ 第 2 章

2 メモリーエラー探索ツール (discover)

メモリーエラー探索ツール (discover) ソフトウェアは、メモリーアクセスエラーを検出するための高度な開発ツールです。

この章には、次の情報が含まれます。

discover を使用するための要件

このセクションでは、discover を使用して最適な結果を得るための要件について説明します。ここで説明する内容は次のとおりです。

バイナリを適切に準備する

discover ユーティリティーは、Sun Studio 12 Update 1、Oracle Solaris Studio 12.2、Oracle Solaris Studio 12.3、または Oracle Solaris Studio 12.4 コンパイラを使用してコンパイルされたバイナリ上で機能します。Solaris 10 10/08 以上のオペレーティングシステ

ム、Oracle Solaris 11、Oracle Enterprise Linux 5.x、または Oracle Enterprise Linux 6.x 以上を実行する SPARC ベースまたは x86 ベースのシステムで動作します。

これらの要件が満たされない場合は、discover ユーティリティーでエラーが発生したり、バイナリが計測されません。ただし、これらの要件を満たさないバイナリを計測し、-l オプションを使用して限定された数のエラーを検出することは可能です。18 ページの「計測オプション」を参照してください。

コンパイルされたバイナリには注釈と呼ばれる情報が含まれ、discover がバイナリを正しく計測するのに役立ちます。このわずかな情報が追加されることで、バイナリのパフォーマンスまたは実行時のメモリー使用量に影響を及ぼすことはありません。

バイナリのコンパイル時に -g オプションを使用してデバッグ情報を生成すると、discover はエラーと警告を報告しながらソースコードと行番号の情報を表示して、より正確な結果を生成できます。バイナリが -g オプションを使用してコンパイルされない場合、discover には対応する機械レベルの命令のプログラムカウンタのみが表示されます。また、-g オプションを使用してコンパイルすると、discover はより正確なレポートを生成できます。discover は多くの最適化バイナリで機能しますが、-g の使用が推奨されます。詳細は、46 ページの「discover エラーメッセージの解釈」を参照してください。

最適な結果を得るため、バイナリは最適化オプションなしで、-g オプションを付けてコンパイルしてください。最適化されたコードは、異なる変数に同じメモリーの場所を使用したり、投機的コードを生成したりするなどの最適化のため、ソースコードと異なることがあります。コンパイル時に高度な最適化オプションを使用することで、discover が正しくないエラーを報告したり、エラーを報告しなかったりする可能性があります。

注記 - discover では、標準メモリー割り当て関数 malloc()、calloc()、memalign()、valloc()、および free() を再定義するバイナリがサポートされます。

詳細は、48 ページの「discover 使用時の制限事項」を参照してください。

プリロードまたは監査を使用するバイナリは互換性がない

discover は実行時リンカーの一部の特定の機能を使用するため、プリロードまたは監査を使用するバイナリと併用することはできません。

discover は特定のシステム関数に割り込む必要があり、関数がプリロードされている場合は割り込めないため、プログラムが LD_PRELOAD 環境変数の設定を必要とする場合、そのプログラムは discover で正しく動作しない可能性があります。

同様に、バイナリが -p オプションまたは -P オプションとリンクされているか、LD_AUDIT 環境変数を設定する必要があるために、プログラムが実行時監査を使用している場合、この監査は discover の監査の使用と競合します。バイナリが監査とリンクされている場合、discover は計

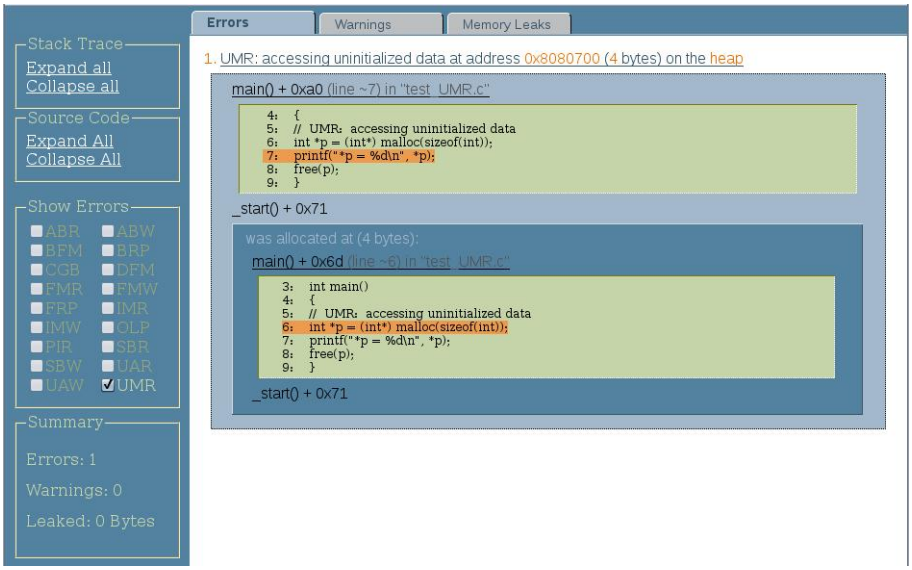
測時に失敗します。実行時に LD_AUDIT 環境変数を設定している場合、結果は定義されません。

簡単なプログラム例

次の例は、プログラムを準備し、discover を使用して計測を行い、それを実行して、検出したメモリーアクセスエラーに関するレポートを生成する方法を示しています。この例は初期化されていないデータにアクセスする単純なプログラムを使用します。

```
% cat test_UMR.c
#include <stdio.h>
#include <stdlib.h>
int main()
{
//  UMR:  accessing uninitialized data
int *p = (int*) malloc(sizeof(int));
printf("*p = %d\n", *p);
free(p);
}

% cc -g -02 test_UMR.c
% a.out
*p = 131464
% discover a.out
% a.out
```

discover の出力は、次の図に示すように、初期化されていないメモリーが使用された場所とそれが割り当てられた場所を、結果のサマリーとともに示します。



準備されたバイナリの計測

ターゲットバイナリを準備したら、次の手順はその計測です。計測は戦略的な場所にコードを追加して、`discover` がバイナリの実行中にメモリー操作を追跡できるようにします。

注記 - SPARC V8 アーキテクチャー上の 32 ビットバイナリの場合、`discover` は計測中に V8plus コードを挿入します。その結果、出力バイナリはバイナリ入力に関係なく常に v8plus になります。

`discover` コマンドを使用して、バイナリを計測します。たとえば、次のコマンドは、バイナリ `a.out` を計測し、入力 `a.out` を計測済みの `a.out` で上書きします。

```
discover a.out
```

計測済みのバイナリを実行する場合、`discover` はプログラムのメモリーの使用をモニターします。実行時に、`discover` はメモリーアクセスエラーを詳述するレポートを Web ブラウザで表示可能な HTML ファイルに書き込みます。デフォルトのファイル名は `a.out.html` です。レポートを ASCII ファイルまたは `stderr` に書き込むように要求するには、バイナリの計測時に `-w` オプションを使用します。

`discover` でバイナリの書き込み専用計測を行うように指定するには、`-n` オプションを使用します。

`discover` がバイナリを計測するときに、注釈が付けられていないために計測できないコードを検出すると、次のような警告が表示されます。

```
discover: (warning): a.out: 80% of code instrumented (16 out of 20 functions)
```

注釈付きでないコードは、バイナリにリンクされているアセンブリ言語コード、またはコンパイラでコンパイルされたモジュール、または11 ページの「バイナリを適切に準備する」にリストされているシステムより古いオペレーティングシステム上から来ている可能性があります。

共有ライブラリのキャッシュ

`discover` はバイナリを計測するときに、そのバイナリにコードを追加します。このコードは実行時リンカーと連携して、実行時に依存共有ライブラリがロードされたときにそれらを計測します。計測済みライブラリは、元のライブラリが最後に計測されてから変更されていない場合には再使用可能なキャッシュに格納されます。デフォルトでは、キャッシュディレクトリは `$HOME/SUNW_Bit_Cache` です。このディレクトリは `-D` オプションを使用して変更できます。

共有ライブラリの計測

すべての共有ライブラリを含む、プログラム全体が計測される場合、`discover` ユーティリティーはもっとも正確な結果を生成します。デフォルトでは、`discover` は実行可能ファイルのメモリーエラーだけを検査して報告します。`discover` で実行可能ファイルのエラーの検査をスキップするように指定するには、`-n` オプションを使用します。

`-c` オプションを使用すると、依存共有ライブラリおよび `dlopen()` によって動的に開かれたライブラリのエラーを `discover` で検査するように指定できます。`-c` オプションを使用して特定のライブラリのエラー検査を回避することもできます。`discover` はそのライブラリのエラーを報告しませんが、メモリーエラーを正しく検出するためにアドレス空間全体のメモリー状態を追跡する必要があるため、すべての共有ライブラリを含むプログラム全体で割り当てとメモリーの初期化を記録します。

`discover` ユーティリティーのランタイムは、リンカーの監査インタフェース (`rtld-audit` または `LD_AUDIT` とも呼ばれる) を使用して、計測される共有ライブラリを `discover` のキャッシュディレクトリから自動的にロードします。Oracle Solaris では、監査インタフェースはデフォルトで使用されます。Linux では、計測されるバイナリの実行中に、コマンド行で `LD_AUDIT` を設定する必要があります。

Oracle Linux 上の 32 ビットアプリケーションの場合:

```
% LD_AUDIT=install-dir/lib/compilers/postopt/bitdl.so a.out
```

Oracle Linux 上の 64 ビットアプリケーションの場合:

```
% LD_AUDIT=install-dir/lib/compilers/postopt/amd64/bitdl.so a.out
```

Oracle Enterprise Linux 5.x を実行しているすべての環境では、このメカニズムは機能しない可能性があります。ライブラリの計測が必要なく、`LD_AUDIT` が設定されていない場合は、`discover` は Oracle Enterprise Linux 5.x 上で問題はありません。

11 ページの「バイナリを適切に準備する」の説明に従って、プログラムで使用されるすべての共有ライブラリを準備する必要があります。デフォルトで、実行時リンカーが準備されていないライブラリを検出する場合、致命的なエラーが発生します。ただし、discover に 1 つ以上のライブラリを無視するように指示できます。

ライブラリの無視

一部のライブラリは、準備または計測できない場合があります。-s、-T、または -N オプション (18 ページの「計測オプション」を参照) を使用して、あるいは bit.rc ファイル (20 ページの「bit.rc 初期化ファイル」を参照) の指定項目を使用して、discover にこれらのライブラリを無視するように指示できます。正確さが多少失われる可能性があります。

ライブラリが計測できず、「無視可能」と指定されていない場合、discover は計測時に失敗したり、プログラムが実行時にエラーメッセージを伴って失敗したりします。

デフォルトでは、discover は bit.rc システムファイルの指定項目を使用して、特定のシステムおよびコンパイラが提供するライブラリを、準備されていないために無視するものとして設定します。discover はもっとも一般的に使用されるライブラリのメモリー特性を認識するため、正確さに対する影響は最小限です。

ライブラリまたは実行可能ファイルの部分的な検査

-c オプションを使用して、実行可能ファイルまたはライブラリを指定できます。メモリーアクセスの検査を特定のオブジェクトファイルに制限することで、ターゲット実行可能ファイルまたはターゲットライブラリをさらに限定できます。

たとえば、ターゲットライブラリが libx.so で、ターゲット実行可能ファイルが a.out の場合は、次のコマンドを使用します。

```
$ discover -c libx.so -o a.out.disc a.out
```

複数のファイルまたはディレクトリをコロンで区切って追加することで、ターゲットの検査を制限することもできます。ファイルには ELF ファイルまたはディレクトリを指定できます。ELF ファイルを指定すると、そのファイルで定義されているすべての関数が検査されます。ディレクトリを指定すると、そのディレクトリ内のすべてのファイルが再帰的に使用されます。

```
$ discover -o a.out.disc a.out:t1.0:dir
```

```
$ discover -c libx.so:l1.o:l2.o -o a.out.disc a.out
```

コマンド行オプション

discover コマンドとともに次のオプションを使用して、バイナリを計測できます。

出力オプション

-a	コードアナライザで使用するためにエラーデータを *binary-name*.analyze/dynamic ディレクトリに書き込みます。
-b*browser*	計測済みのプログラムの実行中に、Web ブラウザ *browser* を自動的に起動します (デフォルトでは off)。
-e *n*	レポートに *n* メモリーエラーのみを表示します (デフォルトでは、すべてのエラーを表示します)。
-E *n*	レポートに *n* メモリーリークのみを表示します (デフォルトは 100 です)。
-f	レポートのオフセットを表示します (デフォルトは非表示です)。
-H*html-file*	discover のバイナリに関するレポートを HTML 形式で *html-file* に書き込みます。このファイルは計測済みバイナリの実行時に作成されます。*html-file* が相対パス名である場合、計測済みバイナリを実行する作業ディレクトリを基準として相対的に配置されます。バイナリを実行するたびにファイル名を一意にするには、ファイル名に文字列 %p を追加して、discover ランタイムにプロセス ID を含めるように指示します。たとえば、オプション -H report.%p.html によって、report.*process-ID*.html というファイル名のレポートファイルが生成されます。ファイル名に複数個の %p を含めると、最初のインスタンスだけがプロセス ID と置き換えられます。 このオプションまたは -w オプションを指定しない場合、レポートは HTML 形式で *output-file*.html に書き込まれます。*output-file* は、計測済みバイナリのベース名です。ファイルは、計測済みバイナリを実行する作業ディレクトリに配置されます。 このオプションおよび -w オプションを指定して、テキストおよび HTML ファイル形式の両方でレポートを書き込むことができます。
-m	レポートの符号化された名前を表示します (デフォルトは符号化されていない名前の表示です)。
-o*file*	計測済みのバイナリを *file* に書き込みます。デフォルトで、計測済みのバイナリは入力バイナリを上書きします。
-S *n*	レポートに *n* スタックフレームのみを表示します (デフォルトは 8 です)。
-w*text-file*	バイナリ上の discover のレポートを *text-file* に書き込みます。計測済みのバイナリを実行するときに、ファイルが作成されます。*text-file* が相対パス名である場合、ファイルは計測済みバイナリを実行する作業ディレクトリを基準として相対的に配置されます。バイナリを実行するたびにファイル名を一意にするには、ファイル名に文字列 %p を追加して、discover

ランタイムに対してプロセス ID を含めるように要求します。たとえば、オプション -w report.%p.txt によって report.*process-ID*.txt というファイル名のレポートファイルが生成されます。ファイル名に複数個の %p を含めると、最初のインスタンスだけがプロセス ID と置き換えられます。-w - を指定すると stderr に出力されます。

このオプションまたは -H オプションを指定しない場合、レポートは HTML 形式で *output-file*.html に書き込まれます。*output-file* は、計測済みバイナリのベース名です。ファイルは、計測済みバイナリを実行する作業ディレクトリに配置されます。

このオプションおよび -H オプションを両方指定して、テキストおよび HTML 形式の両方でレポートを書き込みます。

注記 - -w および -H オプションを使用する場合は、フルパス名を使用することをお勧めします。相対パスが使用されている場合、レポートはプロセスの実行ディレクトリに相対的なディレクトリに生成されます。そのため、アプリケーションがディレクトリを変更し、新しいプロセスを起動する場合に、レポートが誤った場所に配置される可能性があります。アプリケーションが新しいプロセスを作成すると、実行時に libdiscoverADI.so が子プロセスに対して親のエラーレポートのコピーを作成するので、子プロセスはコピーへの書き込みを続行します。子プロセスの実行ディレクトリが異なり、レポートファイルに相対パスが使用されていた場合、その子プロセスが親プロセスを見つけられない可能性があります。フルパス名を使用することで、これらの問題を回避します。

計測オプション

-A [on | off]

割り当て/解放スタックトレースをオンまたはオフにします (デフォルトはスタック深度 8 で on です)。このフラグは、-i adi オプションを使用したハードウェアアシスト検査のための計測時にのみ指定できます。実行時パフォーマンスの向上のため、このオプションで、割り当て/解放スタックトレースの収集をオフにできます。このオプションは、Oracle Solaris Studio 12.4 、4/15 Platform Specific Enhancement (PSE) がインストールされている場合にのみ使用できます。

-c [- | *library [: scope...]* | *file*]

すべてのライブラリ内、指定された *library* 内、または指定された *file* に改行で区切って列挙されているライブラリ内のエラーを検査します。デフォルトでは、ライブラリ内のエラーを検査しません。コロンで区切られたファイルまたはディレクトリを追加することによって、ライブラリの検査のスコープを制限できます。詳細は、16 ページの「ライブラリまたは実行可能ファイルの部分的な検査」を参照してください。

-F [parent | child | both]

discover で計測機構を組み込んだバイナリが実行中にフォークした場合に行う処理を指定します。デフォルトでは、discover は引き続き、親プロセスと子プロセスの両方からメモリーアクセスエラーのデータを収集します。discover が親プロセスにのみ従うようにする場合は、-F parent

を指定します。Discover が子プロセスにのみ従うようにする場合は、`-F child` を指定します。

`-i [datarace | memcheck | adi]`

`discover` の計測タイプを指定します (デフォルトは `memcheck`)。

`datarace` を指定した場合、スレッドアナライザを使用して、データ競合の検出のために計測します。このオプションを使用する場合は、データ競合検出のみが実行時に行われ、他のメモリー検査は行われません。`collect` コマンドを使用して計測済みのバイナリを実行し、パフォーマンスアナライザで表示可能な実験を生成する必要があります。詳細は、『Oracle Solaris Studio 12.4: 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。`memcheck` を指定した場合、メモリーエラー検査のために計測します。`adi` を指定した場合、SPARC M7 プロセッサの ADI 機能を使用して、ハードウェアアシスト検査のために計測します。この機能は SPARC M7 プロセッサで実行されている Oracle Solaris 11.3 でのみ使用できます。`-i adi` オプションは、Oracle Solaris Studio 12.4 、4/15 Platform Specific Enhancement (PSE) をインストールしている場合にのみ使用できます。

`-K`

`bit.rc` 初期化ファイルを読み取らないでください (「20 ページの「`bit.rc` 初期化ファイル」」を参照)。

`-l`

`discover` を簡易モードで実行します。このオプションによって、プログラムの実行速度が向上します。プログラムに特別な準備は必要ありませんが、検出されるエラーの数が制限されます。

`-n`

実行可能ファイルのエラーを検査しません。

`-N` *library*

接頭辞 *library* に一致する依存共有ライブラリを計測しないでください。ライブラリ名の最初の文字が *library* に一致する場合、ライブラリは無視されます。*library* がスラッシュ (/) で始まる場合、ライブラリの完全な絶対パス名でマッチングが行われます。それ以外の場合、ライブラリのベース名でマッチングが行われます。

`-P [on | off]`

正確な ADI モードをオンまたはオフにします。デフォルトは `on` です。このフラグは、`-i adi` オプションを使用したハードウェアアシスト検査のための計測時にのみ指定できます。実行時パフォーマンスの向上のため、このオプションで、正確な ADI モードをオフにできます。このオプションは、Oracle Solaris Studio 12.4 、4/15 Platform Specific Enhancement (PSE) がインストールされている場合にのみ使用できます。

`-s`

計測不可能なバイナリの計測を試みる場合は、警告を発するが、エラーのフラグは立てないでください。

`-T`

指定されたバイナリのみを計測します。依存共有ライブラリを実行時に計測しないでください。

キャッシュオプション

-D *cache-directory* キャッシュされた計測済みバイナリを格納するためのルートディレクトリとして *cache-directory* を使用します。デフォルトでは、キャッシュディレクトリは `$HOME/SUNW_Bit_Cache` です。

-k キャッシュで検出されたライブラリの再計測を強制します。

その他のオプション

-h または -? ヘルプ。短いヘルプメッセージを出力して、終了します。

-v 冗長。`discover` が実行している内容のログを出力します。このオプションを 2 回指定すると、より詳しい情報が出力されます。

-V `discover` のバージョン情報を出力して終了します。

bit.rc 初期化ファイル

`discover` ユーティリティーは、起動時に一連の `bit.rc` ファイルを読み取ることによってその状態を初期化します。システムファイル *Oracle-Solaris-Studio-installation-directory*`/prod/lib/bit.rc` は、特定の変数のデフォルト値を提供します。`discover` ユーティリティーは最初にこのファイルを読み取り、次に `$HOME/.bit.rc` (存在する場合) と *current-directory*`/.bit.rc` (存在する場合) を読み取ります。

`bit.rc` ファイルには、特定の変数値を設定、追加、または削除するコマンドが含まれています。`discover` が `set` コマンドを読み取る場合、変数の前の値がある場合には、それを無効にします。`append` コマンドを読み取る場合、変数の既存の値に (コロンセパレータのあとに) 引数を追加します。`remove` コマンドを読み取る場合、変数の既存の値から引数とそのコロンセパレータを削除します。

`bit.rc` ファイルの変数セットには、計測時に無視するライブラリのリスト、およびバイナリ内の注釈の付いていない (準備されていない) コードの割合を計算する場合に無視する関数または関数接頭語のリストが含まれます。

詳細は、`bit.rc` システムファイルのヘッダーのコメントを参照してください。

計測済みバイナリの実行

`discover` を使用してバイナリを計測したあと、そのバイナリを通常の場合と同じ方法で実行します。通常、特定の組み合わせの入力により、プログラムが予期しない動作を行なった場合

は、そのプログラムを discover を使用して計測し、同じ入力を使用して実行して、潜在的なメモリーの問題を調べます。計測済みのプログラムの実行中に、discover は、選択した形式 (テキスト、HTML、またはその両方) の指定された出力ファイルに、検出されるメモリーエラーに関する情報を書き込みます。レポートの解釈の詳細は、27 ページの「discover レポートの分析」を参照してください。

計測のオーバーヘッドのため、計測後のプログラムは実行速度が大幅に低下する可能性があります。メモリーアクセスの頻度に応じて、50 倍も低速に実行される場合があります。

Silicon Secured Memory (SSM) を使用したハードウェアアシスト検査

Oracle の SPARC M7 プロセッサは、ソフトウェアを高速かつ確実に実行できるようにする Software in Silicon を提供します。Software in Silicon 機能の 1 つが、Silicon Secured Memory (SSM) であり、以前はアプリケーションデータ整合性 (ADI) と呼ばれ、その回路は実行時のデータの破損を引き起こす可能性のある一般的なメモリーアクセスエラーを検出します。

これらのエラーは、誤ったコードまたはサーバーのメモリーへの悪意のある攻撃によって発生することがあります。たとえば、バッファーオーバーフローはセキュリティー上の弱点の主な原因になることがわかっています。さらに、インメモリーデータベースは、重要なデータをメモリー内に保持するため、そのようなエラーがアプリケーションに与える影響が大きくなります。

Silicon Secured Memory は、アプリケーションのメモリーポインタとそれらが指すメモリーにバージョン番号を追加して、最適化された本番コードでのメモリー破損を防ぎます。ポインタバージョン番号が内容のバージョン番号と一致しない場合、メモリーアクセスは中止されます。Silicon Secured Memory は、ソフトウェアエラーによって発生するメモリー破損に対して脆弱な C や C++ などのシステムレベルのプログラミング言語で書かれたアプリケーションで機能します。

Oracle Solaris Studio 12.4 4/15 Platform Specific Enhancement (PSE) には、libdiscoverADI.so ライブラリ (discover ADI ライブラリとも呼ばれる) が含まれ、これは隣接するデータ構造に確実に異なるバージョン番号が指定される更新済みの malloc() ライブラリルーチンを提供します。これらのバージョン番号により、プロセッサの SSM テクノロジがバッファーオーバーフローを検出できます。古いポインタアクセスを防ぐため、メモリー内容のバージョン番号はメモリー構造が解放されるときに変更されます。discover および libdiscoverADI.so によって捕捉されるエラーの詳細については、22 ページの「libdiscoverADI によって捕捉されるエラー」を参照してください。

本番環境で Silicon Secured Memory を使用して潜在的なメモリー破損問題を検出することに加えて、アプリケーション開発時にそれを使用して、アプリケーションのテストと動作保証時にそのようなエラーが捕捉されるようにすることもできます。アプリケーションは破損の発生後かなりたってから破損したデータを検出するため、メモリー破損のバグの発見はきわめて困難です。Oracle Solaris Studio 開発者ツールスイートの一部である discover ツールと

libdiscoverADI.so ライブラリは、誤ったコードの特定と修正を容易にする追加のアプリケーション情報を提供します。

libdiscoverADI ライブラリを使用したメモリーアクセスエラーの検出

discover ADI ライブラリの libdiscoverADI は、無効なメモリーアクセスを引き起こすプログラミングエラーを報告します。次の 2 つの方法で使用できます。

- LD_PRELOAD_64 環境変数でアプリケーションに discover ADI ライブラリをプリロードすることによって。この方法では、アプリケーションのすべての 64 ビットバイナリを ADI モードで実行します。たとえば、server というアプリケーションを通常どおりに実行した場合、コマンドは次のようになります。

  ```
  $ LD_PRELOAD_64=install-dir/lib/compilers/sparcv9/libdiscoverADI.so server
  ```

- 特定のバイナリに対して、-i adi オプションを付けた discover コマンドで ADI モードを使用することによって。

  ```
  % discover -i adi a.out
  % a.out
  ```

エラーはデフォルトで a.out.html ファイルに報告されます。discover レポートの詳細については、27 ページの「discover レポートの分析」および17 ページの「出力オプション」を参照してください。

23 ページの「libdiscoverADI 使用の要件と制限」を参照してください。

libdiscoverADI によって捕捉されるエラー

libdiscoverADI.so ライブラリは次のエラーを捕捉します。

- 配列範囲外アクセス (Array out of Bounds Access) (ABR/ABW)
- 解放済みメモリーへのアクセス (Freed Memory Access) (FMR/FMW)
- 古いポインタアクセス (Stale Pointer Access) (特殊なタイプの FMR/FMW)
- 非割り当て読み取り/書き込み (Unallocated Read/Write) (UAR/UAW)
- メモリーの二重解放 (Double Free Memory) (DFM)

これらの各エラーのタイプの詳細については、41 ページの「メモリーアクセスエラーと警告」を参照してください。

アプリケーションは独自のメモリー割り当ておよび解放リストを管理できます (たとえば、プログラムで大きなチャンクのメモリーを割り当てたり、それを分割したりすることによって)。ADI バージョン管理 API を使用して管理対象メモリーのエラーを捕捉する方法については、アプリケー

ションデータ整合性と Oracle Solaris Studio を使用したメモリーアクセスエラーの検出と修正 (https://community.oracle.com/docs/DOC-912448)に関するドキュメントを参照してください。

完全な例については、24 ページの「discover ADI モードの使用例」を参照してください。

discover ADI モードの計測オプション

次のオプションは、ADI モードでの計測時に discover レポートで生成される情報の精度と量を指定します。

-A [on | off]　このフラグを on に設定すると、discover ADI ライブラリはエラーの場所とエラースタックトレースを報告します。この情報は、エラーを捕捉するには十分ですが、エラーを修正するために必ずしも十分なわけではありません。このフラグは、不正なメモリー領域が割り当てられ、解放された場所に関する情報も生成します。たとえば、エラーがArray out of Bounds Access であったことと、その配列が割り当てられた場所が、出力に示されていることがあります。off に設定されている場合、割り当てとスタックトレースは報告されません。デフォルトは on です。

注記 - -A が on に設定されている場合でも、次のいずれかの理由のため、ABR/ABW が FMR/FMW または UAR/UAW として報告されることがあります。

- バッファーオーバーフローアクセスが、バッファーの末尾のあと、またはバッファーの先頭の前の大きなオフセットで発生している場合。
- libdiscoverADI.so がリソース制限に達した場合。この場合、discover は、エラーがバッファーオーバーフローであるかどうかを判断するために必要な割り当てスタックトレースを維持できる可能性があります。

-P [on | off]　このフラグを off に設定すると、ADI は正確でないモードで実行されます。正確でないモードでは、正確な命令が実行されてからいくつかの命令 (ソース行) が実行されたあとに、メモリー書き込みエラーが捕捉されます。正確なモードを有効にするには、このフラグを on (デフォルト) に設定します。

実行時のパフォーマンスを向上させるために、-A off、-P off を指定でき、または両方のオプションを off に設定できます。

libdiscoverADI 使用の要件と制限

discover の ADI モードは、Oracle Solaris Studio 12.4 4/15 Platform Specific Enhancement (PSE) がインストールされている Oracle Solaris 11.2.8 または Oracle

Solaris 11.3 以上を実行する SPARC M7 チップ上の 64 ビットアプリケーションでのみ使用できます。

メモリー検査のための計測と同様に、`libdiscoverADI.so` の関数が同じ割り当て関数に割り込んだ場合、プリロードされたライブラリが競合することがあります。詳細については、12 ページの「プリロードまたは監査を使用するバイナリは互換性がない」を参照してください。

`libdiscoverADI` でコードを検査する場合のその他の制限には次のものが含まれます。

- ヒープ検査のみ使用できます。スタック検査、静的配列範囲外検査、リーク検出はありません。
- メタデータを格納するために 64 ビットアドレスの未使用のビットを使用するアプリケーションでは動作しません。一部の 64 ビットアプリケーションでは、ロックなど、メタデータを格納するために 64 ビットアドレスの現在未使用の上位ビットを使用することがあります。そのようなアプリケーションでは、`discover` の ADI モードは機能しません。この機能は、バージョン情報を格納するために、64 ビットアドレスの上位 4 ビットを使用して動作するためです。
- たとえば、2 つの連続した割り当ての間の距離など、ヒープアドレスに関する仮定のもとにポインタ演算を行うアプリケーションでは機能しないことがあります。
- `memcheck` モード (`-i memcheck` で計測) と異なり、ADI モードでは、アプリケーションが実行可能ファイル内で標準メモリー割り当て関数を再定義している場合に、エラーを捕捉しません。アプリケーションがライブラリ内で標準メモリー割り当て関数を再定義している場合は、ADI モードが機能します。
- バッファーオーバーフローの解像度は 64 バイトです。64 バイトで整列されている割り当てでは、`libdiscoverADI.so` は 1 バイト以上の単位でオーバーフローを捕捉します。64 バイトで整列されない割り当てでは、数バイト単位でバッファーオーバーフローを見落とす可能性があります。一般に、1 から 63 バイト単位のオーバーフローは、割り当ての整列と `libdiscoverADI.so` がキャッシュ行内に割り当てを配置する場所によって捕捉されないことがあります。
- `-xipo=2` でコンパイルされたバイナリには、誤検出 ADI エラーにつながり、結果としてトラップ処理のためにパフォーマンスの低下をまねくような方法でアドレスを操作するメモリー最適化コードが含まれる可能性がわずかにあります。

`discover` ADI モードの使用例

このセクションでは、ADI モードを使用した `discover` によって捕捉され、報告される配列範囲外エラーのあるコードサンプルについて説明します。

次のサンプルコードが `testcode.c` という名前のファイルにあるものと想定します。

```
#include <stdio.h>
#include <stdlib.h>

int main() {
```

```
  char *x = (char*)malloc(512);
  int *y = (int*)malloc(20*sizeof(int));
  char *z = (char*)malloc(64);

  x[-14] = 0;
  y[-10] = 0;
  z[-4] = 0;
  x[16] = 0;
  y[20] = 0;
  z[64] = 0;
  x[20] = 0;
  y[26] = 0;
  z[120] = 0;

}
```

次のコマンドを使用して、テストコードを構築します。

```
$ cc testcode.c -g -m64
```

このサンプルアプリケーションを ADI モードで実行するには、次のコマンドを使用します。

```
$ discover -w - -i adi -o a.out.adi a.out
$ ./a.out.adi
```

このコマンドは、discover レポートに次の出力を生成します。これらのレポートの見方と内容の詳細については、27 ページの「discover レポートの分析」を参照してください。

```
ERROR 1 (ABW): writing to memory beyond array bounds at address 0x2fffffff7e877ff2:
        main() + 0x3c  <test-abrw.c:10>
                 7:      int *y = (int*)malloc(20*sizeof(int));
                 8:      char *z = (char*)malloc(64);
                 9:
                10:=>    x[-14] = 0;
                11:      y[-10] = 0;
                12:      z[-4] = 0;
                13:      x[16] = 0;
        _start() + 0x108
    was allocated at (512 bytes):
        main() + 0x8  <test-abrw.c:6>
                3:
                4:    int main() {
                5:
                6:=>    char *x = (char*)malloc(512);
                7:      int *y = (int*)malloc(20*sizeof(int));
                8:      char *z = (char*)malloc(64);
                9:
        _start() + 0x108
ERROR 2 (ABW): writing to memory beyond array bounds at address 0x2fffffff7e873ffc:
        main() + 0x50  <test-abrw.c:12>
                 9:
                10:      x[-14] = 0;
                11:      y[-10] = 0;
                12:=>    z[-4] = 0;
                13:      x[16] = 0;
```

```
                14:      y[20] = 0;
                15:      z[64] = 0;
        _start() + 0x108
    was allocated at (64 bytes):
        main() + 0x28  <test-abrw.c:8>
                 5:
                 6:      char *x = (char*)malloc(512);
                 7:      int *y = (int*)malloc(20*sizeof(int));
                 8:=>    char *z = (char*)malloc(64);
                 9:
                10:      x[-14] = 0;
                11:      y[-10] = 0;
        _start() + 0x108
ERROR 3 (ABW): writing to memory beyond array bounds at address 0x2fffffff7e876080:
        main() + 0x64  <test-abrw.c:14>
                11:      y[-10] = 0;
                12:      z[-4] = 0;
                13:      x[16] = 0;
                14:=>    y[20] = 0;
                15:      z[64] = 0;
                16:      x[20] = 0;
                17:      y[26] = 0;
        _start() + 0x108
    was allocated at (128 bytes):
        main() + 0x18  <test-abrw.c:7>
                 4:    int main() {
                 5:
                 6:      char *x = (char*)malloc(512);
                 7:=>    int *y = (int*)malloc(20*sizeof(int));
                 8:      char *z = (char*)malloc(64);
                 9:
                10:      x[-14] = 0;
        _start() + 0x108
ERROR 4 (ABW): writing to memory beyond array bounds at address 0x2fffffff7e874040:
        main() + 0x70  <test-abrw.c:15>
                12:      z[-4] = 0;
                13:      x[16] = 0;
                14:      y[20] = 0;
                15:=>    z[64] = 0;
                16:      x[20] = 0;
                17:      y[26] = 0;
                18:      z[120] = 0;
        _start() + 0x108
    was allocated at (64 bytes):
        main() + 0x28  <test-abrw.c:8>
                 5:
                 6:      char *x = (char*)malloc(512);
                 7:      int *y = (int*)malloc(20*sizeof(int));
                 8:=>    char *z = (char*)malloc(64);
                 9:
                10:      x[-14] = 0;
                11:      y[-10] = 0;
        _start() + 0x108
ERROR 5 (ABW): writing to memory beyond array bounds at address 0x2fffffff7e876098:
```

```
        main() + 0x84  <test-abrw.c:17>
                14:      y[20] = 0;
                15:      z[64] = 0;
                16:      x[20] = 0;
                17:=>    y[26] = 0;
                18:      z[120] = 0;
                19:
                20:    }
        _start() + 0x108
    was allocated at (128 bytes):
        main() + 0x18  <test-abrw.c:7>
                 4:    int main() {
                 5:
                 6:      char *x = (char*)malloc(512);
                 7:=>    int *y = (int*)malloc(20*sizeof(int));
                 8:      char *z = (char*)malloc(64);
                 9:
                10:      x[-14] = 0;
        _start() + 0x108
ERROR 6 (ABW): writing to memory beyond array bounds at address 0x2fffffff7e874078:
        main() + 0x90  <test-abrw.c:18>
                15:      z[64] = 0;
                16:      x[20] = 0;
                17:      y[26] = 0;
                18:=>    z[120] = 0;
                19:
                20:    }
                21:
        _start() + 0x108
    was allocated at (64 bytes):
        main() + 0x28  <test-abrw.c:8>
                 5:
                 6:      char *x = (char*)malloc(512);
                 7:      int *y = (int*)malloc(20*sizeof(int));
                 8:=>    char *z = (char*)malloc(64);
                 9:
                10:      x[-14] = 0;
                11:      y[-10] = 0;
        _start() + 0x108
DISCOVER SUMMARY:
        unique errors   : 6 (6 total)
```

discover レポートの分析

discover レポートは、ソースコードで効果的に自動補完して問題を修正する情報を提供します。

デフォルトでは、レポートは *output-file*.html に HTML 形式で書き込まれます。*output-file* は、計測済みバイナリのベース名です。ファイルは、計測済みバイナリを実行する作業ディレクトリに配置されます。

バイナリを計測する際には、-H オプションを使用して、HTML 出力を指定されたファイルに書き込むように要求するか、-w オプションを使用して、テキストファイルに書き込むように要求できます。

バイナリを計測したあと、その後のプログラム実行でレポートを異なるファイルに書き込む場合は、SUNW_DISCOVER_OPTIONS 環境変数でレポートの -H および -w オプションの設定を変更できます。詳細は、40 ページの「SUNW_DISCOVER_OPTIONS 環境変数」を参照してください。

注記 - コードの計測時に -a オプションを指定する場合、コードアナライザまたは codean コマンドを使用してレポートを読み取る必要があります。

HTML レポートの分析

HTML レポート形式では、プログラムの対話型分析が提供されます。HTML 形式のデータは、電子メールを使用するか、Web ページ上に配置して、開発者間で容易に共有できます。この形式と JavaScript インタラクティブ機能と組み合わせると、discover のメッセージを検索する便利な方法が提供されます。

このセクションでは、HTML レポートについて説明します。ここで説明するタブは次のとおりです。

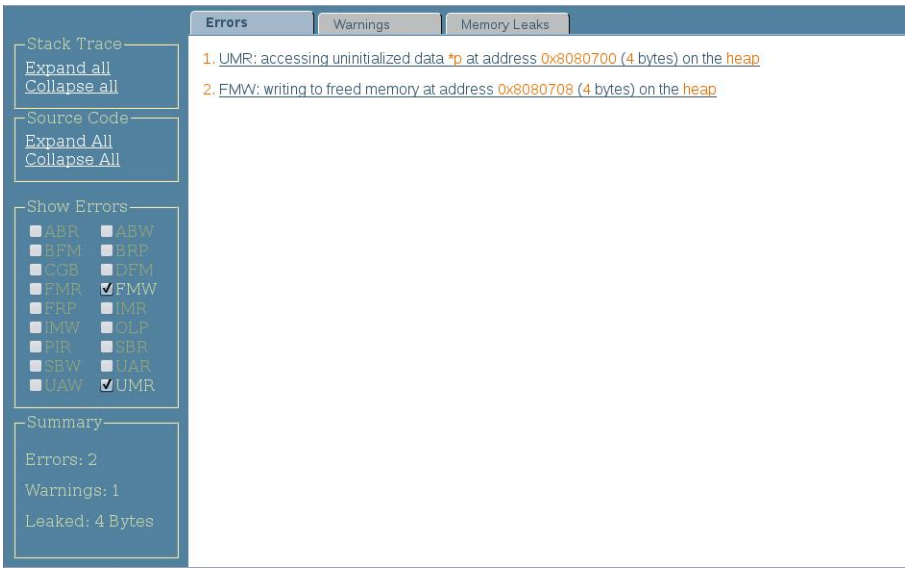
- 28 ページの「「エラー (Errors)」タブの使用法」
- 30 ページの「「警告 (Warnings)」タブの使用法」
- 31 ページの「「メモリーリーク (Memory Leaks)」タブの使用法」

「エラー」タブ、「警告」タブ、および「メモリーリーク」タブでは、エラーメッセージ、警告メッセージ、およびメモリーリークレポートをそれぞれナビゲートできます。

左側のコントロールパネルでは、右側に現在表示されているタブの内容を変更できます。32 ページの「コントロールパネルの使用法」を参照してください。

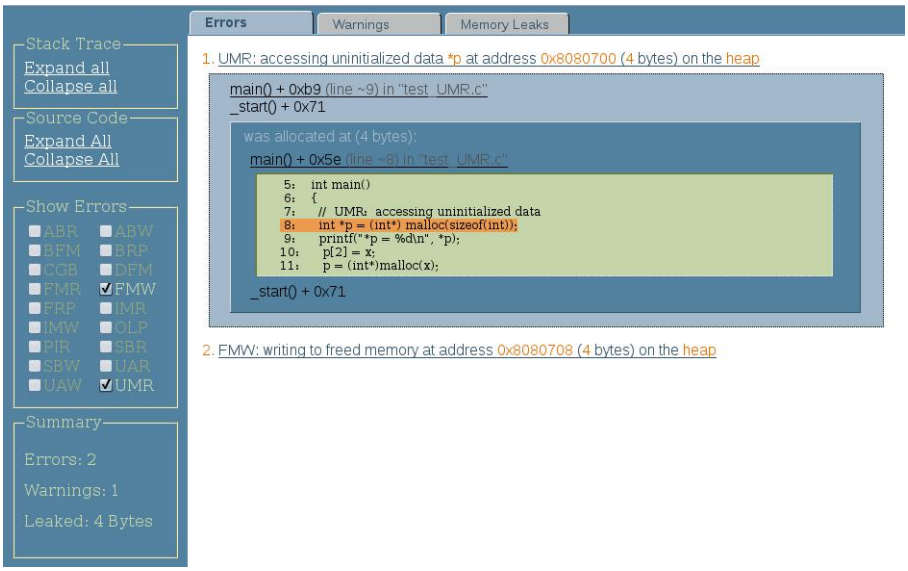
「エラー (Errors)」タブの使用法

ブラウザで最初に HTML レポートを開くと、「エラー」タブが選択され、計測済みのバイナリの実行中に検出されたメモリーアクセスエラーのリストが表示されます。



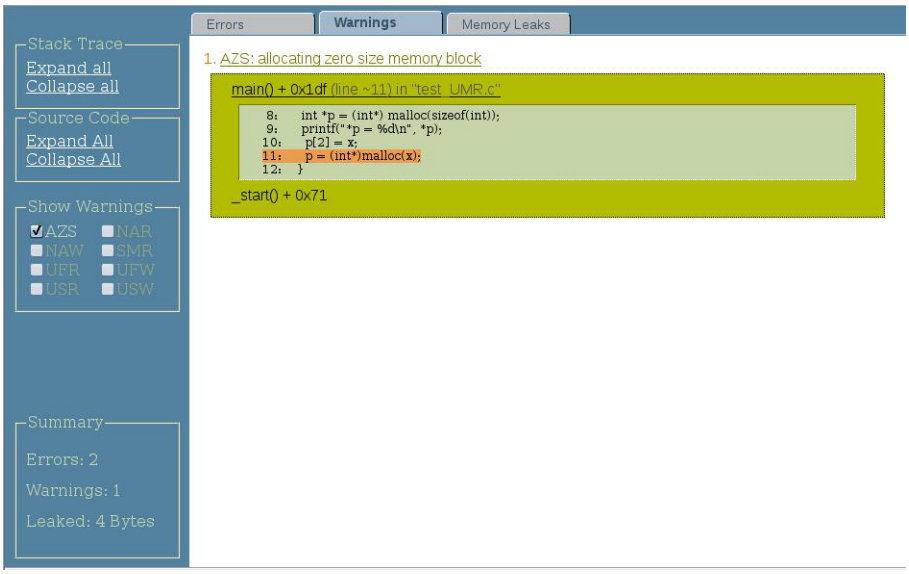
エラーをクリックすると、エラー時のスタックトレースが表示されます。

-g オプションを使用してコードをコンパイルした場合、関数をクリックするとスタックトレースの関数ごとのソースコードを表示できます。



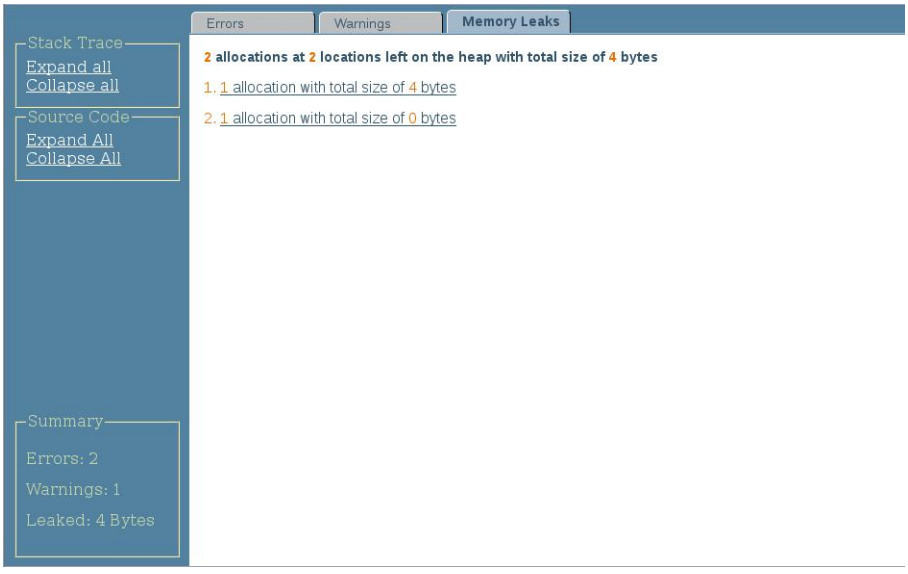
「警告 (Warnings)」タブの使用法

「警告 (Warnings)」タブには、起こり得るアクセスエラーの警告メッセージのすべてが表示されます。警告をクリックすると、警告時のスタックトレースが表示されます。コードを -g オプションを使用してコンパイルした場合、関数をクリックすると、スタックトレースの関数ごとのソースコードを表示できます。

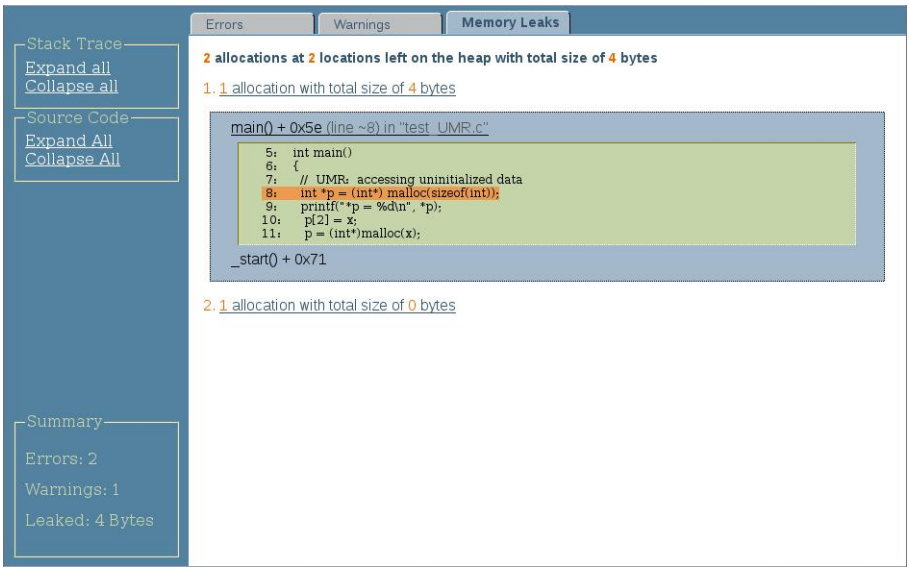


「メモリーリーク (Memory Leaks)」タブの使用法

「メモリーリーク」タブには、下記にリストされているブロック数とともに、最上部にプログラムの実行終了時に割り当てられている残存ブロック総数が表示されます。



ブロックをクリックすると、ブロックのスタックトレースが表示されます。-g オプションを使用してコードをコンパイルした場合、関数をクリックすると、スタックトレースの関数ごとのソースコードを表示できます。



コントロールパネルの使用法

エラー、警告、およびメモリーリークのすべてのスタックトレースを表示するには、コントロールパネルの「スタックトレース」セクションの「すべて展開」をクリックします。関数のすべてのソースコードを表示するには、コントロールパネルの「ソースコード」セクションの「すべて展開」をクリックします。

エラー、警告、およびメモリーリークのすべてのスタックトレースまたはソースコードを非表示にするには、対応する「すべて折りたたむ (Collapse All)」をクリックします。

コントロールパネルの「エラーの表示」または「警告の表示」セクションは、対応するタブを選択したときに表示されます。デフォルトでは、検出されたエラーまたは警告のすべてのオプションがオンになっています。あるエラーまたは警告のタイプを非表示にするには、それを選択解除します。

エラーおよび警告の総数を一覧表示するレポートのサマリー、およびリークしたメモリー量がコントロールパネルの下方に表示されます。

ASCII レポートの分析

discover レポートの ASCII (テキスト) 形式は、スクリプトで処理する場合や、Web ブラウザにアクセスできない場合に適しています。ASCII レポートの例を次に示します。

```
$ a.out

ERROR 1 (UAW): writing to unallocated memory at address 0x50088 (4 bytes) at:
main() + 0x2a0  <ui.c:20>
17:     t = malloc(32);
18:     printf("hello\n");
19:     for (int i=0; i<100;i++)
20:=>    t[32] = 234; // UAW
21:     printf("%d\n", t[2]);   //UMR
22:     foo();
23:     bar();
_start() + 0x108
ERROR 2 (UMR): accessing uninitialized data from address 0x50010 (4 bytes) at:
main() + 0x16c  <ui.c:21>$
18:     printf("hello\n");
19:     for (int i=0; i<100;i++)
20:      t[32] = 234; // UAW
21:=>   printf("%d\n", t[2]);   //UMR
22:     foo();
23:     bar();
24:    }
_start() + 0x108
was allocated at (32 bytes):
main() + 0x24  <ui.c:17>
14:     x = (int*)malloc(size); // AZS warning
15:    }
16:    int main() {
17:=>   t = malloc(32);
18:     printf("hello\n");
19:     for (int i=0; i<100;i++)
20:      t[32] = 234; // UAW
_start() + 0x108
0
WARNING 1 (AZS): allocating zero size memory block at:
foo() + 0xf4  <ui.c:14>
11:    void foo() {
12:     x = malloc(128);
13:     free(x);
14:=>   x = (int*)malloc(size); // AZS warning
15:    }
16:    int main() {
17:     t = malloc(32);
main() + 0x18c  <ui.c:22>
19:     for (int i=0; i<100;i++)
20:      t[32] = 234; // UAW
21:     printf("%d\n", t[2]);   //UMR
22:=>   foo();
```

```
23:     bar();
24:    }
_start() + 0x108

***************** Discover Memory Report *****************

1 block at 1 location left allocated on heap with a total size of 128 bytes

1 block with total size of 128 bytes
bar() + 0x24  <ui.c:9>
6:           7:    void bar() {
8:     int *y;
9:=>   y = malloc(128);  // Memory leak
10:    }
11:    void foo() {
12:     x = malloc(128);
main() + 0x194  <ui.c:23>
20:      t[32] = 234; // UAW
21:     printf("%d\n", t[2]);   //UMR
22:     foo();
23:=>   bar();
24:    }
_start() + 0x108

ERROR 1: repeats 100 times
DISCOVER SUMMARY:
unique errors   : 2 (101 total, 0 filtered)
unique warnings : 1 (1 total, 0 filtered)
```

このレポートはエラーと警告メッセージ、およびそのサマリーで構成されます。

ASCII の警告およびエラーメッセージの説明

エラーメッセージには、ERROR という単語で始まり、3 文字のコード、ID 番号、およびエラーの説明 (この例では、`writing to unallocated memory`) が含まれています。その他の詳細には、アクセスされたメモリーアドレスと、読み取られた、または書き込まれたバイト数が含まれます。説明のあとには、プロセスライフサイクルでエラーの場所を自動補完するエラー時のスタックトレースが表示されます。

-g オプションを使用してプログラムをコンパイルすると、スタックトレースにソースファイル名と行番号が含まれます。ソースファイルにアクセス可能な場合、エラー付近のソースコードが出力されます。各フレームのターゲットソース行は、⇒記号によって示されます。

同じバイト数を持つ同じメモリーの場所の同じエラーの種類が繰り返される場合、スタックトレースを含む完全なメッセージが 1 度だけ出力されます。後続のエラーの出現が数えられ、次の例に表示されるように繰り返し数が、複数回発生する同一エラーごとにレポートの末尾に一覧表示されます。

```
ERROR 1: repeats 100 times
```

不正なメモリーアクセスのアドレスがヒープ上にある場合、対応するヒープブロックに関する情報がスタックトレース後に出力されます。その情報には、ブロック開始アドレスとサイズ、およびブロックが割り当てられた時点のスタックトレースが含まれます。ブロックが解放された場合は、解放ポイントのスタックトレースもレポートに含まれます。

警告メッセージは、WARNING という単語で始まる点を除き、エラーメッセージと同じ形式で表示されます。これらのメッセージは、一般にアプリケーションの機能に影響を及ぼさない状態に対する警告ですが、問題を改善するために使用できる有益な情報を提供します。たとえば、0 サイズのメモリーを割り当てることは有害ではありませんが、頻繁すぎると、パフォーマンスを低下させる可能性があります。

ASCII メモリーリークレポート

メモリーリークレポートには、ヒープ上に割り当てられているがプログラムの終了時にリリースされないメモリーブロックに関する情報が含まれます。メモリーリークレポートの例を次に示します。

```
$ DISCOVER_MEMORY_LEAKS=1 ./a.out
...
***************** Discover Memory Report *****************

2 blocks left allocated on heap with total size of 44 bytes
block at 0x50008 (40 bytes long) was allocated at:
malloc() + 0x168 [libdiscover.so:0xea54]
f() + 0x1c [a.out:0x3001c]
<discover_example.c:9>:
8:    {
9:=>    int *a = (int *)malloc( n * sizeof(int) );
10:      int i, j, k;
main() + 0x1c [a.out:0x304a8]
<discover_example.c:33>:
32:      /* Print first N=10 Fibonacci numbers */
33:=>    a = f(N);
34:      printf("First %d Fibonacci numbers:\n", N);
_start() + 0x5c [a.out:0x105a8]
...
```

ヘッダーに続く最初の行は、ヒープ上に割り当てられて残されているヒープブロック数とその合計サイズを要約しています。レポートされるサイズは、開発者の見解であり、すなわちメモリーアロケータのブックキーピングのオーバーヘッドは含まれません。

ASCII スタックトレースレポート

メモリーリークのサマリーのあとに、割り当てポイントのスタックトレースを持つ未解放ヒープブロックごとの詳細情報が提供されます。スタックトレースレポートは、エラーおよび警告メッセージに対して説明されるレポートと同様です。

ASCII レポートサマリー

discover レポートの最後には、全体的なサマリーが出力されます。かっこ付きの一意の警告およびエラー数、繰り返しを含むエラーおよび警告の総数が報告されます。例:

```
DISCOVER SUMMARY:
unique errors   : 3 (3 total)
unique warnings : 1 (5 total)
```

discover API と環境変数

コード内に指定できる discover API と環境変数がいくつか用意されています。

discover API

Oracle Solaris Studio 12.4 には、プログラムから呼び出してメモリーリークやメモリー割り当ての情報を受け取ることができる 6 つの新しい discover 関数が実装されています。これらの関数は、stderr に情報を出力します。discover は、デフォルトでプログラム出力の最後に、プログラム内のメモリーリークを含む最終的なメモリーレポートを出力します。これらの API を使用するには、アプリケーションのソースファイルに discover 用のヘッダーファイルをインクルードする必要があります (#include <discoverAPI.h>)。

関数と報告される内容は次のとおりです。

discover_report_all_inuse()
: すべてのメモリー割り当てを報告します

discover_report_unreported_inuse()
: 以前に報告されていないすべてのメモリー割り当てを報告します。

discover_mark_all_inuse_as_reported()
: 今までに報告されたすべてのメモリー割り当てをマークします。

discover_report_all_leaks()
: すべてのメモリーリークを報告します。

discover_report_unreported_leaks()
: 以前に報告されていないすべてのメモリーリークを報告します。

discover_mark_all_leaks_as_reported()
: 今までに報告されたすべてのメモリーリークをマークします

このセクションでは、discover API の使用方法について説明します。

注記 - discover API は ADI モードで動作しません。

discover API によるメモリーリークの検出

discover は、コード内に指定した関数ごとに、メモリーが割り当てられた場所のスタックを報告します。メモリーリークとは、プログラム内で到達不可能な割り当てメモリーのことです。

次の例は、これらの API の使用方法を示しています。

```
$ cat -n tdata.C
     1    #include <discoverAPI.h>
     2
     3    void foo()
     4    {
     5      int *j = new int;
     6    }
     7
     8    int main()
     9    {
    10      foo();
    11      discover_report_all_leaks();
    12
    13      foo();
    14      discover_report_unreported_leaks();
    15
    16      return 0;
    17    }
$ CC -g tdata.C
$ discover -w - a.out
$ a.out
```

次の例は、予想される出力を示しています。

```
******** discover_report_all_leaks() Report ********
1 allocation at 1 location left on the heap with a total size of 4 bytes
LEAK 1: 1 allocation with total size of 4 bytes
void*operator new(unsigned) + 0x36
void foo() + 0x5e  <tdata.C:5>
2:
3:    void foo()
4:    {
5:=>    int *j = new int;
6:    }
7:
8:    int main()
main()+0x1a  <tdata.C:10>
9:    {
10:=>    foo();
```

```
11:      discover_report_all_leaks();
12:
13:      foo();         _start() +

************************************************************
******** discover_report_unreported_leaks() Report ********
1 allocation at 1 location left on the heap with a total size of 4 bytes
LEAK 1: 1 allocation with total size of 4 bytes
void*operator new(unsigned) + 0x36
void foo() + 0x5e  <tdata.C:5>
2:
3:    void foo()
4:    {
5:=>    int *j = new int;
6:    }
7:
8:int main()
main() + 0x24  <tdata.C:13>
10:      foo();
11:      discover_report_all_leaks();
12:
13:=>    foo();
14:      discover_report_unreported_leaks();
15:
16:return 0;
_start() + 0x71

************************************************************
***************** Discover Memory Report *****************
2 allocations at 2 locations left on the heap with a total size of 8     bytes
LEAK 1: 1 allocation with total size of 4 bytes
void*operator new(unsigned) + 0x36
void foo() + 0x5e  <tdata.C:5>
2:
3:    void foo()
4:    {
5:=>    int *j = new int;
6:    }
7:
8:    int main()
main() +    0x1a  <tdata.C:10>
7:
8:    int main()
9:    {             10:=>    foo();
11:      discover_report_all_leaks();
12:
13:      foo();         _start() + 0x71
LEAK 2: 1 allocation with total size of 4 bytes
void*operator new(unsigned) + 0x36
void foo() + 0x5e  <tdata.C:5>
2:
3:    void foo()
4:    {
5:=>    int *j = new int;
```

```
6:    }
7:
8:    int main()
main() + 0x24  <tdata.C:13>
10:      foo();
11:      discover_report_all_leaks();
12:
13:=>    foo();
14:         discover_report_unreported_leaks();
15:
16:      return 0;
_start() + 0x71

DISCOVER SUMMARY:
unique errors   : 0 (0 total)
unique warnings : 0 (0 total)
```

サーバーまたは長時間実行プログラム内のリークの検出

終了することがない長時間実行プログラムまたはサーバーがある場合は、コード内に呼び出しを配置しなくても、dbx を使用していつでもこれらの discover 関数を呼び出すことができます。プログラムは、少なくとも -l オプションを使用して discover の軽量モードで実行されている必要があります。dbx は実行中のプログラムに接続できます。次の例は、長時間実行プログラム内のリークを検出する方法を示しています。

例 1 長時間実行プログラム内の 2 つのリークの検出

この例で、a.out ファイルは 2 つのプロセスを持つ長時間実行プログラムであり、各プロセスにリークが 1 つずつあります。各プロセスにはプロセス ID が割り当てられています。

次の rl スクリプトには、このプログラムに対して報告されていないメモリーリークを報告するように要求するコマンドが含まれています。

```
#!/bin/sh
dbx - $1 > /dev/null 2> &1 << END
call discover_report_unreported_leaks()
exit
END
```

プログラムとスクリプトが用意できたら、discover を使用してプログラムを実行できます。

```
% discover -l -w - a.out
% a.out
8252: Parent allocation 64
8253: Child allocation 32
```

別個の端末ウィンドウで、親プロセスに対してスクリプトを実行できます。

```
% rl 8252
```

プログラムは親プロセスに関して次の情報を報告します。

```
******** discover_report_unreported_leaks() Report ********

1 allocation at 1 location left on the heap with a total size of 64 bytes

    LEAK 1: 1 allocation with total size of 64 bytes
    main() + 0x1e  <xx.c:17>
        14:
        15:      if (child > 0) {
        16:
        17:=>      void *p = malloc(64);
        18:        printf("%jd: Parent allocation 64\n", (intmax_t)getpid());
        19:        p = 0;
        20:        for (int j=0; j < 1000; j++) sleep(1);
    _start() + 0x66


************************************************************
```

子プロセスに対して再度プロセスを実行します。

```
% rl 8253
```

プログラムは子プロセスに関して次の情報を報告します。

```
******** discover_report_unreported_leaks() Report ********

1 allocation at 1 location left on the heap with a total size of 32 bytes

    LEAK 1: 1 allocation with total size of 32 bytes
    main() + 0x80  <xx.c:24>
        21:      }
        22:
        23:      else {
        24:=>      void *p = malloc(32);
        25:        printf("%jd: Child allocation 32\n", (intmax_t)getpid());
        26:        p = 0;
        27:        for (int j=0; j < 1000; j++) sleep(1);
    _start() + 0x66


************************************************************
```

スクリプトを繰り返し使用して、新しいリークを検出できます。

SUNW_DISCOVER_OPTIONS 環境変数

計測対象バイナリの実行時の動作を変更するには、SUNW_DISCOVER_OPTIONS 環境変数に、コマンド行オプション -a、-A、-b-e、-E、-f、-F、-H、-l、-L、-m、-P、-S、および -w のリストを設定します。た

とえば、レポートされるエラー数を 50 に変更し、レポート内のスタックの深さを 3 に制限する場合、環境変数を次のように設定します。

```
-e 50 -s 3
```

SUNW_DISCOVER_FOLLOW_FORK_MODE 環境変数

デフォルトでは、discover で計測機構を組み込んだバイナリが実行中にフォークした場合、discover は親および子プロセスからメモリーアクセスエラーのデータを収集し続けます。つまり、デフォルトの動作は both です。たとえば、discover でフォークを追跡し、子プロセスからメモリーアクセスデータを収集する場合は、SUNW_DISCOVER_FOLLOW_FORK_MODE 環境変数を次のように設定します。

```
-F child
```

メモリーアクセスエラーと警告

discover ユーティリティーは、多数のメモリーアクセスエラー、およびエラーである可能性のあるアクセスに関する警告を検出および報告します。

メモリーアクセスエラー

discover は次のメモリーアクセスエラーを検出します。

- ABR: 配列境界を越える読み取り (beyond array bounds read)
- ABW: 配列境界を越える書き込み (beyond array bounds write)
- BFM: 不正な空きメモリー (bad free memory)
- BRP: 不正な realloc アドレスパラメータ (bad reallocate address parameter)
- CGB: 破壊された配列ガードブロック (corrupted array guard block)
- DFM: メモリーの二重解放 (double freeing memory)
- FMR: 解放済みメモリーの読み取り (freed memory read)
- FMW: 解放済みメモリーの書き込み (freed memory write)
- FRP: 解放済み Realloc パラメータ (freed realloc parameter)
- IMR: 無効なメモリーの読み取り (invalid memory read)
- IMW: 無効なメモリーの書き込み (invalid memory write)
- メモリーリーク
- OLP: 送り側と受け側の重複 (overlapping source and destination)
- PIR: 部分的に初期化された読み取り (partially initialized read)

- SBR: スタックフレームの範囲外からの読み取り (beyond stack frame bounds read)
- SBW: スタックフレームの範囲外への書き込み (beyond stack frame bounds write)
- UAR: 割り当てられていないメモリーの読み取り (unallocated memory read)
- UAW: 割り当てられていないメモリーの書き込み (unallocated memory write)
- UMR: 初期化されていないメモリーの読み取り (uninitialized memory read)

次のセクションに、これらのエラーの一部を生成する簡単なサンプルプログラムをリストします。

ABR

```
  //  ABR: reading memory beyond array bounds at address 0x%1x (%d byte%s)
int *a = (int*) malloc(sizeof(int[5]));
printf("a[5] = %d\n",a[5]);
```

discover ユーティリティーは、静的な型の ABR エラーも検出します。

```
    int globalarray[5];

   int main(){
     int i, j;
     for(i = 0; i < 7; i++) {
       j = globalarray[i-1];   // Reading memory beyond static/global array bounds
     }
     return 0;
   }
```

ABW

```
  //  ABW:  writing to memory beyond array bounds
int *a = (int*) malloc(sizeof(int[5]));
a[5] = 5;
```

discover ユーティリティーは、静的な型の ABW エラーも検出します。

```
int globalarray[5];

   int main(){
     int i;
     for(i = 0; i < 7; i++) {
       globalarray[i-1] = i;   // Writing to memory beyond static/global array bounds
     }
     return 0;
   }
```

BFM

```
  //  BFM:  freeing wrong memory block
```

```
int *p = (int*) malloc(sizeof(int));
free(p+1);
```

BRP

```
  //  BRP:  bad address parameter for realloc 0x%1x
int *p = (int*) realloc(0,sizeof(int));
int *q = (int*) realloc(p+20,sizeof(int[2]));
```

CGB

```
// CGB: writing past the end of a dynamically allocated array, or being in the "red zone".
    #include <stdio.h>
   #include <stdlib.h>

   int main() {
     int *p = (int *) malloc(sizeof(int)*4);
     *(p+5) = 10; //  Corrupted array guard block detected (only when the code is not
 annotated)
     free(p);

     return 0;
   }
```

DFM

```
  // DFM: double freeing memory
int *p = (int*) malloc(sizeof(int));
free(p);
free(p);'
```

FMR

```
  //  FMR: reading from freed memory at address 0x%1x  (%d byte%s)
int *p = (int*) malloc(sizeof(int));
free(p);
printf("p = 0x%h\n",p);
```

FMW

```
  //  FMW: writing to freed memory at address 0x%1x (%d byte%s)
int *p = (int*) malloc(sizeof(int));
free(p);
```

```
*p = 1;
```

FRP

```
  //  FRP:  freed pointer passed to realloc
int *p = (int*) malloc(sizeof(int));
free(0);
int *q = (int*) realloc(p,sizeof(int[2]));
```

IMR

```
  //  IMR:  read from invalid memory address
int *p = 0;
int i = *p;   // generates Signal 11...
```

IMW

```
  //  IMW:  write to invalid memory address
int *p = 0;
*p = 1;       // generates Signal 11...
```

メモリーリーク

```
//Memory Leak: memory allocated but not freed before exit or escaping from the function
       int foo()
   {
     int *p = (int*) malloc(sizeof(int));
     if (x) {
      p = (int *) malloc(5*sizeof(int));  // will cause a leak of the 1st malloc
     }
   }                                     // The 2nd malloc leaked here
```

OLP

```
  //  OLP: source and destination overlap
char *s=(char *) malloc(15);
memset(s, 'x', 15);
memcpy(s, s+5, 10);
return 0;
```

PIR

```
  //  PIR:  accessing partially initialized data
```

```
int *p = (int*) malloc(sizeof(int));
*((char*)p) = 'c';
printf("*(p = %d\n",*(p+1));
```

SBR

```
  //  SBR: reading beyond stack frame bounds
int a[2]={0,1};
printf("a[-10]=%d\n",a[-10]);
return 0;
```

SBW

```
  //  SBW: writing beyond stack frame bounds
int a[2]={0,1)'
a[-10]=2;
return 0;
```

UAR

```
  //  UAR" reading from unallocated memory
int *p = (int*) malloc(sizeof(int));
printf("*(p+1) = %d\n",*(p+1));
```

UMR

```
  // UMR: accessing uninitialized data from address 0x%1x (A%d byte%s)
int *p = (int*) malloc(sizeof(int));
printf("*p = %d\n",*p);
```

メモリーアクセスの警告

`discover` ユーティリティーは次のメモリーアクセスの警告を報告します。

- AZS: 0 サイズの割り当て (allocating zero size)
- SMR: 投機的な非初期化メモリーからの読み取り (speculative uninitialized memory read)

次の例は、AZS 警告を生成する簡単なプログラム例を示しています。

```
  //  AZS:  allocating zero size memory block
```

```
int *p = malloc();
```

discover エラーメッセージの解釈

場合によっては、discover は実際にはエラーでないエラーを報告することがあります。そのようなケースは、擬陽性と呼ばれます。discover ユーティリティーでは計測時にコードが分析されるため、同様なツールと比較して擬陽性の発生は少なくなっていますが、それでも場合によっては発生することがあります。このセクションでは、discover レポート内の擬陽性を特定し、可能であれば回避できるようにするためのヒントを提供します。

部分的に初期化されたメモリー

C および C++ のビットフィールドを使用して、コンパクトなデータ型を作成できます。例:

```
struct my_struct {
unsigned int valid : 1;
char         c;
};
```

この例では、構造メンバー my_struct.valid はメモリー内の 1 ビットのみ取得します。ただし、SPARC プラットフォーム上では、CPU はバイト単位でのみメモリーを変更できるため、struct.valid を含むバイト全体が構造メンバーにアクセスまたは変更するためにロードされる必要があります。また、コンパイラは一度に数バイト (たとえば、4 バイトの機械語) をロードする場合があります。discover がこのようなロードを検出する場合、追加情報なしに、すべて 4 バイトが使用されると仮定します。たとえば、フィールド my_struct.valid は初期化されたが、フィールド my_struct.c は初期化されず、両方のフィールドを含む機械語がロードされた場合、discover は部分的に初期化されたメモリーからの読み取り (PIR) にフラグを立てます。

擬陽性の別のソースはビットフィールドの初期化です。1 バイト部分を書き込むには、コンパイラは最初にバイトをロードするコードを生成する必要があります。バイトが読み取るより前に書き込まれていない場合、結果は非初期化メモリーからの読み取りエラー (UMR) となります。

ビットフィールドの擬陽性を回避するには、コンパイル時に -g オプションまたは -g0 オプションを使用します。これらのオプションは、discover に追加のデバッグ情報を提供し、ビットフィールドのロードと初期化を特定するのに役立ち、多くの擬陽性を削除します。何らかの理由で -g オプションを使用してコンパイルできない場合は、memset() などの関数を使用して構造を初期化します。例:

```
...
struct my_struct s;
/* Initialize structure prior to use */
memset(&sm 0, sizeof(struct my_struct));
```

...

投機的ロード

コンパイラは、ロードの結果がすべてのプログラムパスで有効とは限らない条件下で、既知のメモリーアドレスからロードを生成する場合があります。このようなロード命令は分岐命令の遅延スロットに配置できるため、この状況は SPARK プラットフォーム上で発生する場合がよくあります。たとえば、この C コードフラグメントを考えてみます。

```
int i'
if (foo(&i) != 0) { /* foo returns nonzero if it has initialized i */
printf("5d\n", i);
}
```

コンパイラは、このコードから次の例と同等のコードを生成する可能性があります。

```
int i;
int t1, t2'
t1 = foo(&i);
t2 = i; /* value in i is loaded */
if (t1 != 0) {
printf("%d\n", t2);
}
```

この例では、関数 `foo()` が `0` を返し、`i` を初期化しないと仮定します。`i` からのロードは、使用されないにもかかわらず、生成されます。だたし、`discover` はロードを確認して、非初期化変数のロード (UMR) を報告します。

`discover` ユーティリティーはデータフロー分析を使用して、可能な場合には常にそのようなケースを特定しますが、検出できない場合もあります。

最適化レベルを低くしてコンパイルすることによって、これらのタイプの擬陽性の発生を削減できます。

未計測コード

`discover` は、プログラムの 100% を計測できない場合があり、特に、コードの一部がアセンブリ言語のソースファイルまたは再コンパイルできないサードパーティーのライブラリから来ているため、計測できない場合があります。`discover` は、未計測コードがアクセスまたは変更しているメモリーブロックを検出できません。たとえば、サードパーティの共有ライブラリから得られる関数がのちにメイン (計測済み) プログラムによって読み取られるメモリーブロックを初期化すると仮定します。`discover` はメモリーがライブラリで初期化されていることを検出できないため、後続の読み取りにより、非初期化メモリーエラー (UMR) が生成されます。

このような場合の解決策を提供するため、`discover` API には次の関数が含まれています。

```
void __ped_memory_write(unsigned long addr, long size, unsigned long pc);
void __ped_memory_read(unsigned long addr, long size, unsigned long pc);
void __ped_memory_copy(unsigned long src, unsigned lond dst, long size, unsigned long pc);
```

プログラムから API 関数を呼び出して、メモリー領域への書き込み (`__ped_memory_write()`) やメモリー領域からの読み取り (`__ped_memory read()`) などの特定のイベントを `discover` に知らせることができます。どちらの場合も、メモリー領域の開始アドレスは `addr` パラメータで渡され、そのサイズは `size` パラメータで渡されます。`pc` パラメータを `0` に設定します。

`__ped_memory_copy` 関数を使用して、メモリーがある場所から別の場所にコピーされていることを `discover` に知らせます。ソースメモリーの開始アドレスは `src` パラメータで渡され、宛先領域の開始アドレスは `dst` パラメータで渡され、サイズは `size` パラメータで渡されます。`pc` パラメータを `0` に設定します。

API を使用するには、プログラムでこれらの関数を weak と宣言します。たとえば、ソースコードに次のコードフラグメントを含めます。

```
#ifdef __cplusplus
extern "C" {
#endif

extern void __ped_memory_write(unsigned long addr, long size, unsigned long pc);
extern void __ped_memory_read(unsigned long addr, long size, unsigned long pc);
extern void __ped_memory_copy(unsigned long src, unsigned long dst, long size, unsigned long
 pc);

#prgama weak __ped_memory_write
#pragma weak __ped_memory_read
#pragma weak __ped_memory_copy

#ifdef __cplusplus
}
#endif
```

内部 `discover` ライブラリは、計測時にプログラムにリンクされ、API 関数を定義します。ただし、プログラムが計測されない場合、このライブラリはリンクされず、すべての API 関数呼び出しでアプリケーションがハングします。そのため、`discover` 下でプログラムを実行していない場合は、これらの関数を無効にする必要があります。または、API 関数の空の定義を使用して動的ライブラリを作成し、それをプログラムとリンクすることができます。この場合、`discover` を使用しないでプログラムを実行する場合は、ライブラリが使用されますが、`discover` 下で実行する場合は、真の API 関数が自動的に呼び出されます。

discover 使用時の制限事項

このセクションでは、`discover` 使用時の既知の制限事項について説明します。

注釈付きコードのみが計測される

discover ユーティリティーでは、11 ページの「バイナリを適切に準備する」の説明に従って準備されているコードのみを計測できます。注釈の付いていないコードは、バイナリにリンクされているアセンブリ言語コード、またはそのセクションに示されてるものより古いコンパイラまたはオペレーティングシステムでコンパイルされたモジュールから来ている場合があります。

discover ユーティリティーは、asm 文または .il テンプレートを含むアセンブリ言語モジュールまたは関数を計測できません。

さらに、Oracle Solaris Studio コンパイラでは注釈データがコンパイルされないため、Oracle Solaris Studio 12.4 の C++ 実行時ライブラリには注釈データが含まれていません。-std=c++11 オプションを使用した C++ コンパイラで構築されたプログラムで discover を使用する場合、discover は UMR または PIR エラーを捕捉しません。

機械命令はソースコードとは異なる場合がある

discover は機械コード上で動作します。ツールは、ロードやストアなどの機械命令でエラーを検出し、それらのエラーをソースコードと相互に関連付けます。一部のソースコード文には関連付けられている機械命令がないため、discover は明白なユーザーエラーを検出していないように思われる場合があります。たとえば、次の C コードフラグメントを考えてみましょう:

```
  int *p = (int *)malloc(sizeof(int));
int i;

i = *p;  /* compiler may not generate code for this statement */
printf("Hello World!\n");

return;
```

p で示されたアドレスに格納された値を読み取ることは、メモリーが初期化されていないため、潜在的なユーザーエラーとなります。ただし、最適化コンパイラは変数 i が使用されていないことを検出するため、メモリーから読み取って i に割り当てる文のコードは生成されません。この場合、discover は非初期化メモリーの使用 (UMR) を報告しません。

コンパイラオプションは生成されたコードに影響を及ぼす

コンパイラの生成コードは予測できません。コンパイラが生成するコードは、-O*n* 最適化オプションを含む、使用するコンパイラオプションによって異なるため、discover によって報告されるエラーも異なる可能性があります。たとえば、-O1 最適化レベルで生成されたコードで報告されるエラーは、-O4 最適化レベルで生成されたコードには当てはまらない可能性があります。

プログラムが C++11 標準オプションを使用したコンパイラで構築された場合、discover ツールは、UMR および PIR エラーを検出できません。

システムライブラリは報告されたエラーに影響を及ぼす可能性がある

システムライブラリは、オペレーティングシステムとともにインストール済みで、計測用に再度コンパイルできません。discover ユーティリティーは、標準の C ライブラリ (libc.so) からの一般的な関数に対するサポートを提供します。つまり、discover はこれらの関数によってどのメモリーにアクセスされ、どのメモリーが変更されるかを認識しています。ただし、アプリケーションがほかのシステムライブラリを使用する場合、discover レポートで擬陽性を検出する可能性があります。擬陽性が報告される場合、コードから discover API を呼び出してそれらを削除できます。

カスタムメモリー管理はデータの正確さに影響を及ぼす可能性がある

discover ユーティリティーは、malloc()、calloc()、free()、operator new()、および operator delete() などの標準のプログラミング言語メカニズムによって割り当てられている場合にヒープメモリーを検出できます。

アプリケーションが標準の関数とともにカスタムメモリー管理システム (たとえば、malloc() とともに実装されるプール割り当て管理) を使用する場合、discover がリークや解放されたメモリーへのアクセスを正しく報告することは保証されません。

discover ユーティリティーは、次のメモリーアロケータをサポートしていません。

- brk(2)() または sbrk(2)() システム呼び出しを直接使用するカスタムヒープアロケータ
- バイナリに静的にリンクされた標準のヒープ管理関数
- mmap(2)() および shmget(2)() システム呼び出しを使用してユーザーコードから割り当てられたメモリー

sigaltstack(2)() 関数はサポートされていません。

静的および自動配列範囲外は削除できない

discover が配列範囲の検出に使用するアルゴリズムのため、静的および自動 (ローカル) 配列の自動的な範囲外アクセスエラーは検出できません。ただし、discover は静的な配列範囲外アクセスエラーを検出できます。動的に割り当てられた配列のエラーを検出できます。

♦ ♦ ♦ 第 3 章

3 コードカバレージツール (uncover)

コードカバレージツール (uncover) ソフトウェアは、アプリケーションのコードカバレージを測定します。この章では、次のトピックについて説明します。

uncover を使用するための要件

uncover ユーティリティーは、Sun Studio 12 Update 1、Oracle Solaris Studio 12.2、Oracle Solaris Studio 12.3 コンパイラ、または Oracle Solaris Studio 12.4 以降を使用してコンパイルされたバイナリ上で機能します。Solaris 10 10/08 以上のオペレーティングシステム、Oracle Solaris 11、Oracle Enterprise Linux 5.x、または Oracle Enterprise Linux 6.x 以上を実行する SPARC ベースまたは x86 ベースのシステムで動作します。

上記のようにコンパイルされるバイナリには、uncover がカバレージデータの収集用に計測するためにバイナリを確実に逆アセンブリする情報が含まれています。

Uncover でソースコードレベルのカバレージ情報を使用できるようにするには、-g オプションを使用してバイナリのコンパイル時にデバッグ情報を生成します。バイナリが -g オプションを使用してコンパイルされない場合、プログラムカウンタ (PC) ベースのカバレージ情報のみを使用します。

uncover ユーティリティーは、Oracle Solaris Studio コンパイラで構築された任意のバイナリで機能しますが、最適化オプションなしで構築されたバイナリで最適に機能します。以前のリリースの uncover では、少なくとも -O1 最適化レベルが必要でした。最適化オプションを使用してバイナリを構築した場合、uncover の結果は最適化レベルが低いほど (-O1 または -O2) 良くなります。uncover は、バイナリが -g オプションで構築されたときに生成されるデバッグ情報を使用して命令を行番号に関連付けることにより、ソース行レベルのカバレージを派生し

ます。最適化レベル -O3 以上では、実行されることのないコードや冗長なコードがコンパイラで削除される場合があり、その結果、一部のソースコード行にはバイナリ命令が存在しないことがあります。このような場合、それらの行に関するカバレージ情報は報告されません。詳細は、66 ページの「uncover 使用時の制限事項」を参照してください。

uncover の使用法

Uncover を使用してカバレージ情報を生成するには、3 ステップのプロセスがあります。

1. 52 ページの「バイナリの計測」
2. 53 ページの「計測済みバイナリの実行」
3. 53 ページの「カバレージレポートの生成と表示」

このセクションでは、Uncover を使用する 3 つのステップと、Uncover の使用例について説明します。

バイナリの計測

入力バイナリは、実行可能ファイルまたは共有ライブラリとすることができます。個別に分析するバイナリごとに計測を行う必要があります。

uncover コマンドを使用してバイナリを計測します。たとえば、次のコマンドは、バイナリ a.out を計測し、入力の a.out を計測済みの a.out で上書きします。また、このコマンドは、接尾辞 .uc を持つディレクトリ (この場合は a.out.uc) を作成します。この中に、カバレージデータが収集されます。入力バイナリのコピーはこのディレクトリに保存されます。

```
$ uncover a.out
```

バイナリに計測機構を組み込むとき、次のオプションを使用できます。

-c
: 命令、ブロック、および関数の実行カウントの報告を有効にします。デフォルトでは、カバーされているコードまたはカバーされていないコードの情報だけが報告されます。バイナリに計測機構を組み込むときとカバレージレポートを生成するときの両方で、このオプションを指定します。

-d *directory*
: *directory* 内にカバレージデータディレクトリを作成します。このオプションは、すべてのカバレージデータディレクトリが同じディレクトリ内に作成されているため、複数のバイナリ用のカバレージデータを収集する場合に役立ちます。また、異なる場所から同じ計測済みバイナリの異なるインスタンスを実行する場合、このオプションを使用すると、これらの実行のすべてから取得されるカバレージデータが同じカバレージデータディレクトリに確実に蓄積されるようになります。

-d オプションを使用しない場合、カバレージデータディレクトリは現在の実行ディレクトリに作成されます。

`-m on | off` スレッドセーフなプロファイリングを有効または無効にします。デフォルトは on です。このオプションは -c 実行時オプションと組み合わせて使用します。スレッドを使用するバイナリに -m off で計測機構を組み込むと、バイナリは実行時に失敗し、バイナリに -m on で計測機構を組み込み直すよう指示するメッセージが表示されます。

`-o` *output-binary-file* 指定されたファイルに計測機構付きバイナリファイルを書き込みます。デフォルトでは、入力バイナリファイルが計測機構付きファイルで上書きされます。

すでに計測されている入力バイナリ上で uncover コマンドを実行すると、uncover はすでに計測されているためバイナリを計測できないことと、そのバイナリを実行してカバレージデータを生成できることを示すエラーメッセージを発行します。

計測済みバイナリの実行

バイナリを計測したあとで、それを正常に実行できます。計測済みバイナリを実行するたびに、コードカバレージデータは、uncover が計測中に作成した .uc 接尾辞を持つカバレージデータディレクトリに収集されます。uncover のデータコレクションはマルチスレッドおよびマルチプロセスに対して安全であるため、プロセスの同時実行またはスレッド数に制限はありません。カバレージデータは実行およびスレッドのすべてにわたって蓄積されます。

カバレージレポートの生成と表示

カバレージレポートを生成するには、カバレージデータディレクトリ上で uncover コマンドを実行します。例:

```
$ uncover a.out.uc
```

このコマンドは、a.out.uc ディレクトリのカバレージデータから *binary-name*.er と呼ばれる Oracle Solaris Studio パフォーマンスアナライザ実験ディレクトリを生成し、パフォーマンスアナライザ GUI を起動して、実験を表示します。現在のディレクトリまたはホームディレクトリに .er.rc ファイルがある場合は、パフォーマンスアナライザが実験を表示する方法に影響を及ぼすことがあります。.er.rc ファイルの詳細は、『Oracle Solaris Studio 12.4: パフォーマンスアナライザ 』を参照してください

レポートは、HTML 形式で生成して Web ブラウザに表示するか、ASCII 形式で生成して端末ウィンドウに表示できます。また、コードアナライザで分析して表示するためのディレクトリにデータを転送することもできます。

-a	コードアナライザで使用できるように、エラーデータを *binary-name*.analyze/coverage ディレクトリに書き込みます。
-c	命令、ブロック、および関数の実行カウントの報告を有効にします。デフォルトでは、カバーされているコードまたはカバーされていないコードの情報だけが報告されます。(バイナリに計測機構を組み込むときとカバレージレポートを生成するときの両方で、このオプションを指定します。)
-e on | off	カバレージレポート用の実験ディレクトリを生成し、パフォーマンスアナライザ GUI で実験を表示するかどうかを決定します。デフォルトは on です。
-H *html-directory*	カバレージデータを指定のディレクトリに HTML 形式で保存し、それを Web ブラウザで自動的に表示します。
-h または -?	ヘルプを表示します。
-n	カバレージレポートを生成しますが、パフォーマンスアナライザや Web ブラウザなどのビューアを起動しません。
-t *ascii-file*	指定されたファイルで ASCII カバレージレポートを生成します。
-V	uncover バージョンを出力して終了します。
-v	冗長。Uncover が実行する内容のログを出力します。

1 つの出力形式のみが有効になります。複数の出力オプションを指定すると、uncover はコマンド内の最後のオプションを使用します。

例 2 uncover コマンドの例

```
$ uncover a.out
```

このコマンドは、バイナリ a.out を計測し、入力 a.out を上書きして、現作業ディレクトリに a.out.uc カバレージデータディレクトリを作成し、a.out.uc ディレクトリに入力 a.out のコピーを保存します。a.out がすでに計測されている場合、警告メッセージが表示され、計測は実行されません。

```
$ uncover -d coverage a.out
```

このコマンドは、a.out.uc カバレージディレクトリをディレクトリ coverage に作成します。

```
$ uncover a.out.uc
```

このコマンドは、a.out.uc カバレージディレクトリのデータを使用して、作業ディレクトリにコードカバレージの実験 (a.out.er) を作成し、パフォーマンスアナライザを起動してその実験を表示します。

```
$ uncover -H a.out.html a.out.uc
```

このコマンドは、a.out.uc カバレージディレクトリのデータを使用して、ディレクトリ a.out.html に HTML コードカバレージレポートを作成し、Web ブラウザにレポートを表示します。

```
$ uncover -t a.out.txt a.out.uc
```

このコマンドは、a.out.uc カバレージディレクトリのデータを使用して、ファイル a.out.txt に ASCII コードカバレージレポートを作成します。

```
$ uncover -a a.out.uc
```

このコマンドは a.out.c カバレージディレクトリ内のデータを使用して、コードアナライザで使用するためのカバレージレポートを *binary-name*.analyze/coverage ディレクトリ内に作成します。

パフォーマンスアナライザのカバレージレポートを理解する

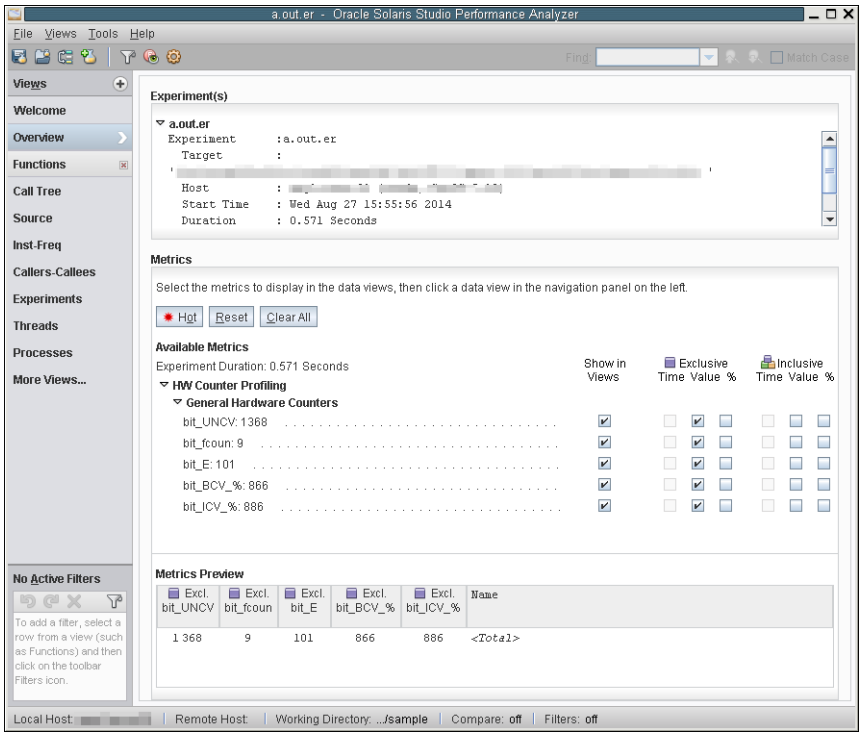
デフォルトでは、カバレージディレクトリ上の uncover コマンドを実行すると、カバレージレポートが Oracle Solaris Studio パフォーマンスアナライザで実験として開きます。このセクションでは、カバレージデータを表示するパフォーマンスアナライザのインタフェースについて説明します。

パフォーマンスアナライザの詳細は、統合ヘルプおよび『Oracle Solaris Studio 12.4: パフォーマンスアナライザ 』を参照してください。

「概要」画面

パフォーマンスアナライザでカバレージレポートを開くと、「概要」画面が表示されます。このビューには、実行中の「実験」、実験の「メトリック」、および「メトリックのプレビュー」が表示されます。

次の図は、パフォーマンスアナライザの「概要」画面を示しています。



「関数」ビュー

ナビゲーションパネルで「関数」ビューをクリックすると、プログラムの関数と排他的メトリックが表示されます。特定のメトリックの値に従ってデータをソートするには、目的の列見出しをクリックします。列ヘッダーの下にある矢印をクリックすると、ソート順が逆になります。

メトリックには次のようなものがあります。

`bit_UNCV` 関数でカバーできるバイト数を示す「カバレージ外」カウンタ。

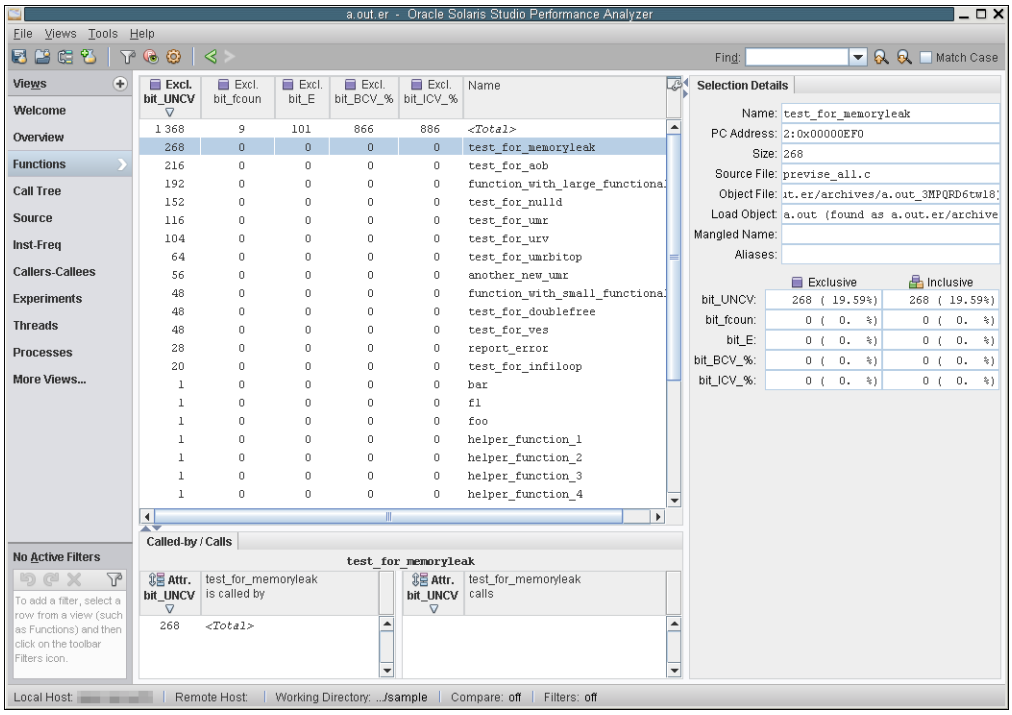
`bit_fcoun` どの関数がカバーされているかを示す「関数」カウンタ。

`bit_E` 関数内で命令が実行されたかどうかを示す「命令の実行」カウンタ。

`bit_BCV_%` 関数内でカバーされているブロックの割合を示す「カバーされているブロックの割合」カウンタ。

`bit_ICV_%` 関数内でカバーされている命令の割合を示す「カバーされている命令の割合」カウンタ。

次の図は、パフォーマンスアナライザ内の bit_UNCV でソートされたカバレージレポートを示しています。



「カバレージ外」カウンタ (bit_UNCV)

bit_UNCV メトリックは、「カバレージ外」カウンタとも呼ばれ、uncover の非常に強力な機能です。この列を降順のソートキーとして使用する場合、最上部に表示される関数は、カバレージを増やす可能性がもっとも高い関数です。前の図では、test_for_memory_leak() 関数が bit_UNCV 列で最大数を持つため、リストの最上部に表示されています。function_with_small_functionality()、test_for_doublefree()、および test_for_ves() 関数は同数を持つため、アルファベット順に一覧表示されています。

test_for_memory_leak() 関数の bit_UNCV の数は、関数が呼び出される原因となるスイートにテストが追加される場合に潜在的にカバーされる可能性のあるコードのバイト数です。カバレージが実際に増加する量は、関数の構造によって異なります。関数に分岐がなく、呼び出すすべての関数も直線関数である場合、カバレージは一定のバイト数で増加します。ただし、通常、カバレージの増加は潜在的に想定されるより (おそらくずっと) 少なくなります。

bit_UNCV 列の 0 以外の値を持つカバーされていない関数は、カバーされていないルート関数と呼ばれ、カバーされている関数によってすべて呼び出されることを意味します。カバーされていない非ルート関数によってのみ呼び出される関数は、独自のカバレージ外の数を持ちません。テストスイートは、潜在性の高いカバーされていない関数をカバーするように改良されるにつれて、これらの関数は、後続の実行で、カバーされるか、またはカバーされないことが明らかにされると想定されます。

カバレージ数は排他的ではありません。

「関数カウント」カウンタ (bit_fcoun)

bit_fcoun は、カバーされている関数とカバーされていない関数を報告します。数が 0 の場合、関数はカバーされません。数が 0 以外の場合、関数はカバーされます。関数の命令が実行される場合、関数はカバーされるとみなされます。

この列で、トップレベル以外のカバーされていない関数を検出できます。ある関数の bit_fcoun が 0 で、bit_UNCV も 0 である場合、その関数はトップレベルのカバーされている関数ではありません。

「命令の実行」カウンタ (bit_E)

bit_E カウンタは、カバーされている命令とカバーされていない命令を表示します。0 のカウントは命令が実行されないことを意味し、0 以外のカウントは命令が実行されることを意味します。

このカウンタは、各関数で実行される命令の総数を「関数」ビューに表示します。このカウンタは、「ソース」ビューと「逆アセンブリ」ビューにも表示されます。

「カバーされているブロックの割合」カウンタ (bit_BCV_%)

bit_BCV_% は、カバーされている関数内の基本的なブロックの割合を関数ごとに表示する「カバーされているブロックの割合」カウンタです。この数値は、関数がどの程度カバーされているかを示します。「合計」行のこの数は無視してください。これは列のパーセンテージの合計であり、意味がありません。

「カバーされている命令の割合」カウンタ (bit_ICV_%)

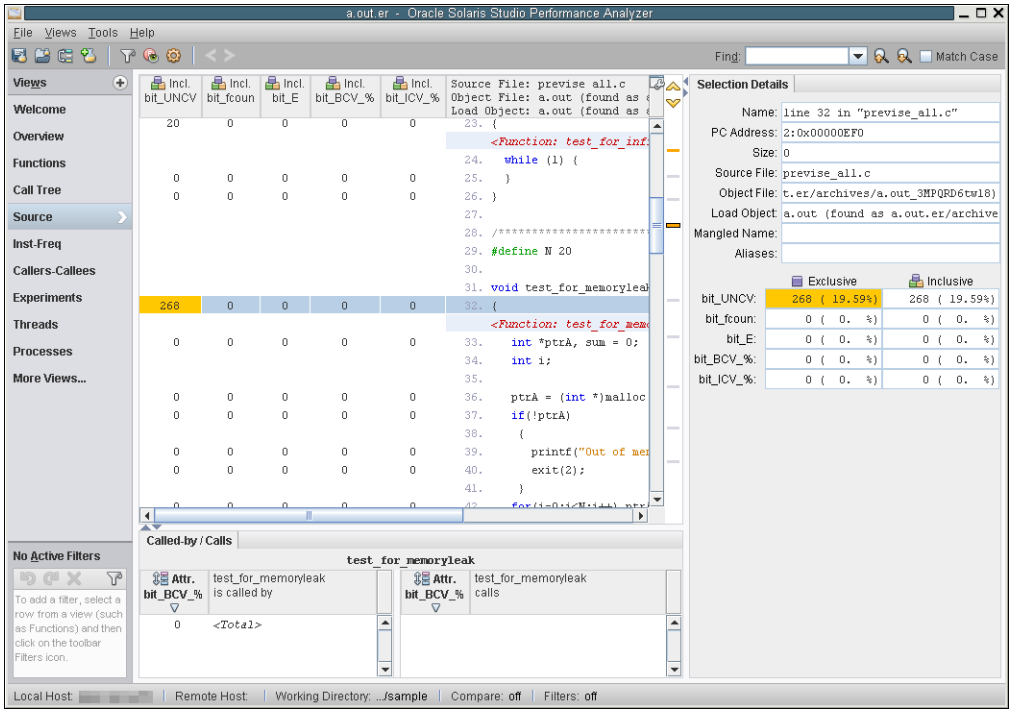
bit_ICV_% カウンタは、カバーされている関数内の命令の割合を関数ごとに表示します。この数値は、関数がどの程度カバーされているかを示します。「合計」行のこの数は無視してください。これは列のパーセンテージの合計であり、意味がありません。

「ソース」ビュー

-g オプションを使用してバイナリをコンパイルすると、「ソース」ビューにプログラムのソースコードが表示されます。uncover はバイナリレベルでプログラムを計測しますが、最適化してプログラムをコンパイルしているため、このビューのカバレージ情報は解釈するのが困難な場合があります。

「ソース」ビューの包括的 bit_E カウンタには、各ソース行で実行される命令の総数が表示され、これは基本的に文レベルのコードカバレージ情報です。0 以外の値は、文がカバーされていることを意味します。0 の値は、文がカバーされていないことを意味します。変数宣言およびコメントには bit_E カウントがありません。

次の図は、開かれた「ソース」ビューの例を示しています。



関連付けられたカバレージ情報がないソースコード行については、行が空白になり、どのフィールドにも数値が表示されません。これらの空の行は、次の理由で発生します。

- コメント、空行、宣言など、実行可能コードを含んでいない言語構造。
- 次のいずれかの理由で、コンパイラの最適化によって行に対応するコードが削除された。
  - コードは実行されることがない (デッドコード)。

- コードは実行可能であるが冗長。

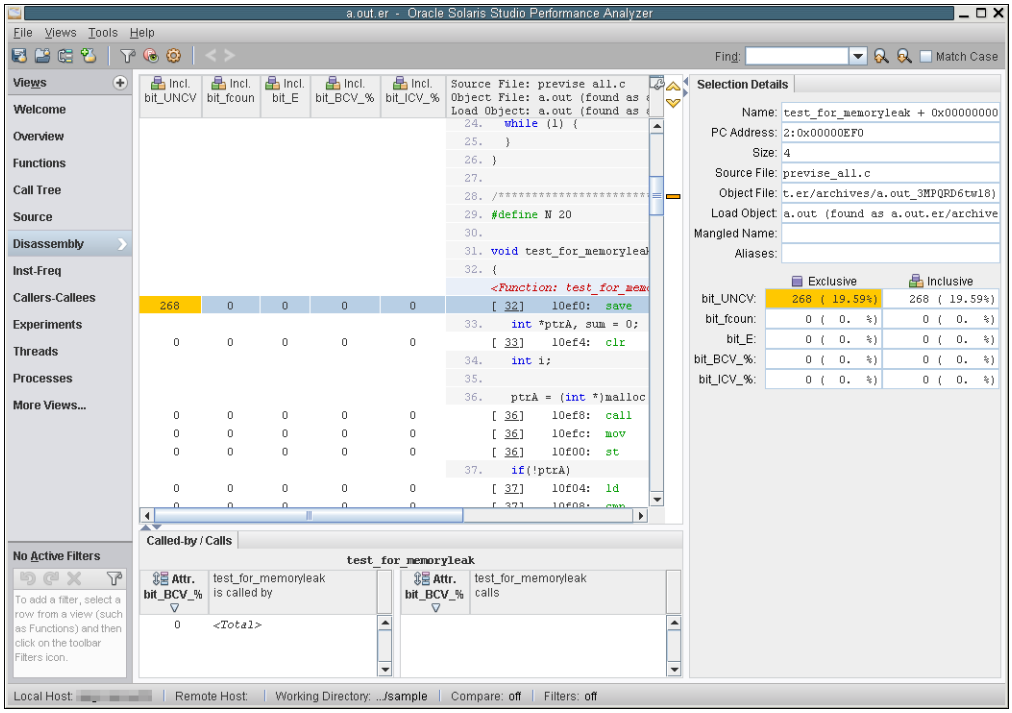
詳細は、66 ページの「uncover 使用時の制限事項」を参照してください。

「逆アセンブリ」ビュー

「ソース」ビューで行を選択してから「逆アセンブリ」ビューを選択すると、パフォーマンスアナライザはバイナリ内の選択された行を検出し、その逆アセンブリを表示しようとします。

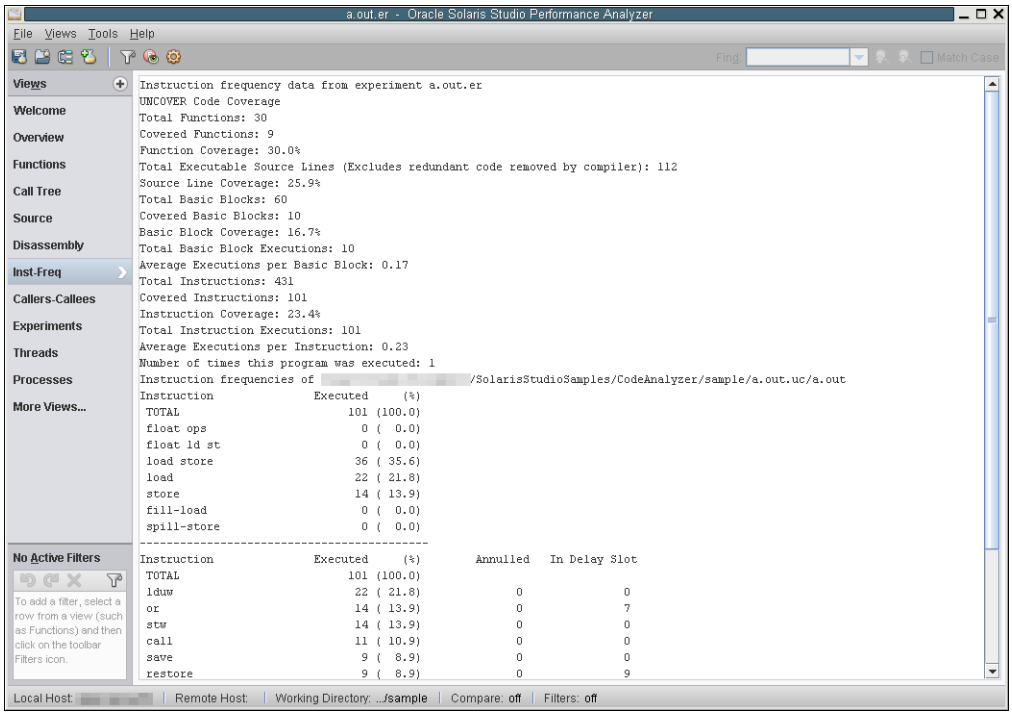
ヒント - 「ビュー」ペインに「逆アセンブリ」が表示されない場合は、「詳細ビュー...」を選択して「逆アセンブリ」オプションをオンにします。

このビューの `bit_E` カウンタには、各命令が実行された回数が表示されます。



「命令頻度」ビュー

「命令頻度」ビューには、全体的なカバレージのサマリーが表示されます。



ASCII カバレージレポートを理解する

カバレージデータディレクトリからカバレージレポートを生成するときに -t オプションを指定すると、uncover はカバレージレポートを指定された ASCII (テキストファイル) に書き込みます。

例 3 サンプル ASCII カバレージレポート

次の例は、サンプル ASCII カバレージレポートを示しています。

```
UNCOVER Code Coverage
Total Functions: 95
Covered Functions: 58
Function Coverage: 61.1%
Total Basic Blocks: 568
Covered Basic Blocks: 258
Basic Block Coverage: 45.4%
Total Basic Block Executions: 564,812,760
Average Executions per Basic Block: 994,388.66
Total Instructions: 6,201
Covered Instructions: 3,006
Instruction Coverage: 48.5%
```

```
Total Instruction Executions: 4,760,934,518
Average Executions per Instruction: 767,768.83
Number of times this program was executed: unavailable
Functions sorted by metric: Exclusive Uncoverage

Excl.       Excl.     Excl.      Excl.       Name
Uncoverage  Function  Block      Instr
Count     Covered %  Covered %
13404       6004876   5464       5384        <Total>
1036             0      0          0        main
980              0      0          0        iofile
748              0      0          0        do_vforkexec
732              0      0          0        callso
708              0      0          0        do_forkexec
648              0      0          0        callsx
644              0      0          0        sigprof
644              0      0          0        sigprofh
556              0      0          0        do_chdir
548              0      0          0        correlate
492              0      0          0        do_popen
404              0      0          0        pagethrash
384              0      0          0        so_cputime
384              0      0          0        sx_cputime
348              0      0          0        itimer_realprof
336              0      0          0        ldso
304              0      0          0        hrv
300              0      0          0        do_system
300              0      0          0        do_burncpu
300              0      0          0        sx_burncpu
288              0      0          0        forkcopy
276              0      0          0        masksignals
256              0      0          0        sigprof_handler
256              0      0          0        sigprof_sigaction
216              0      0          0        do_exec
196              0      0          0        iotest
176              0      0          0        closeso
156              0      0          0        gethrustime
144              0      0          0        forkchild
144              0      0          0        gethrpxtime
136              0      0          0        whrlog
112              0      0          0        masksig
92               0      0          0        closesx
84               0      0          0        reapchildren
36               0      0          0        reapchild
32               0      0          0        doabort
8                0      0          0        csig_handler
0                1     66         72        acct_init
0                1    100        100        bounce
0               63    100         96        bounce_a
0               60    100        100        bounce-b
0               16     71         58        check_sigmask
0                1     83         77        commandline
0                1    100         98        cputime
0                1    100         98        dousleep
```

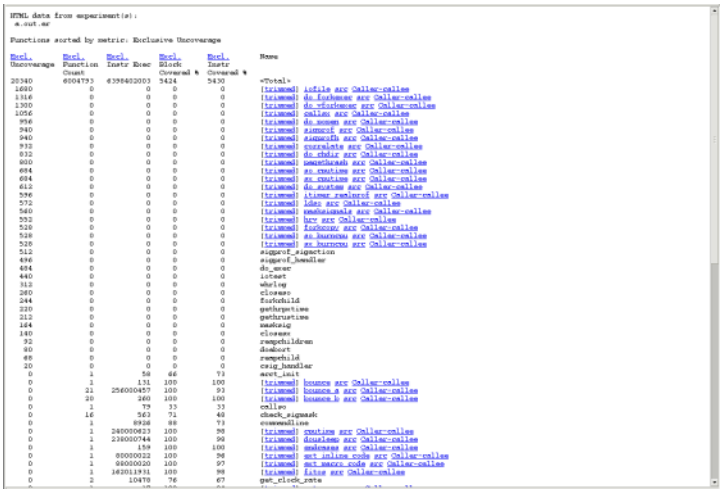
```
0             1    100        100        endcases
0             1    100         95        ext_inline_code
0             1    100         96        ext_macro_code
0             1    100         99        fitos
0             2     81         80        get_clock_rate
0             1    100        100        get_ncpus
0             1    100        100        gpf
0             1    100        100        gpf_a
0             1    100        100        gpf_b
0            10    100         93        gpf_work
0             1    100         97        icputime
0             1    100         96        inc_body
0             1    100         96        inc_brace
0             1    100         95        inc_entry
0             1    100         95        inc_exit
0             1    100         96        inc_func
0             1    100         94        inc_middle
0             1     57         72        init_micro_acct
0             1     50         43        initcksig
0             1    100         95        inline_code
0             1    100         95        macro_code
0             1    100         98        muldiv
0       6000000    100        100        my_irand
0             1    100         98        naptime
0            19     50         83        prdelta
0            21    100        100        prhrdelta
0            21    100        100        prhrvdelta
0             1    100        100        prtime
0           552    100         98        real_recurse
0             1    100        100        recurse
0             1    100        100        recursedeep
0             1    100         95        s_inline_code
0             1    100        100        sigtime
0             1    100         95        sigtime_handler
0            19    100        100        snaptod
0             1    100        100        so_init
0             2     66         75        stpwtch_alloc
0             1    100        100        stpwtch_calibrate
0             2     75         66        stpwtch_print
0          2002    100        100        stpwtch_start
0          2000     90         91        stpwtch_stop
0             1    100        100        sx_init
0             1    100         99        systime
0             3    100         95        tailcall_a
0             3    100         95        tailcall_b
0             3    100         95        tailcall_c
0             1    100        100        tailcallopt
0             1    100         97        underflow
0            21     75         71        whrvlog
0            19    100        100        wlog

Instruction frequency data from experiment a.out.er

Instruction frequencies of /export/home1/synprog/a.out.uc
```

```
Instruction              Executed     ()
TOTAL                  4760934518 (100.0)
float ops              2383657378 ( 50.1)
float ld st            1149983523 ( 24.2)
load store             1542440573 ( 32.4)
load                    882693735 ( 18.5)
store                   659746838 ( 13.9)
-----------------------------------------
Instruction              Executed     ()      Annulled   In Delay Slot
TOTAL                  4760934518 (100.0)
add                     713013787 ( 15.0)           16         1501335
subcc                   558774858 ( 11.7)            0            6002
br                      558769261 ( 11.7)            0               0
stf                     432500661 (  9.1)          726        36299281
ldf                     408226488 (  8.6)           40       103000396
faddd                   391230847 (  8.2)            0               0
fdtos                   366200726 (  7.7)            0               0
fstod                   360200000 (  7.6)            0               0
lddf                    288250336 (  6.1)          500       282200229
stw                     138028738 (  2.9)        26002        25974065
lduw                    118004305 (  2.5)           71        94000270
ldx                      68212446 (  1.4)            0            2000
stx                      68211370 (  1.4)            7        23532716
fitod                    36026002 (  0.8)            0               0
sethi                    36002986 (  0.8)            0             228
fdtoi                    30000001 (  0.6)            0               0
fdivd                    26000088 (  0.5)            0               0
call                     22250348 (  0.5)            0               0
srl                      21505246 (  0 5)            0              21
stdf                     21006038 (  0.4)            0               0
or                       19464766 (  0.4)            0        10981277
fmuls                     6004907 (  0.3)            0               0
jmpl                      6004853 (  0.1)            0               0
save                      6004852 (  0.1)            0               0
restore                   6002294 (  0.1)            0         6004852
sub                       6000019 (  0.1)            0               0
xor                       6000000 (  0.1)            0               0
fitos                     6000000 (  0.1)            0               0
fstoi                     6000000 (  0.1)            0               0
and                       6000000 (  0.1)            0               0
andn                      6000000 (  0.1)            0               0
sll                       3505225 (  0.1)            0               0
nop                       3505219 (  0.1)            0         3505219
fxtod                        7763 (  0.0)            0               0
bpr                          6000 (  0.0)            0               0
fcmped                       4837 (  0.0)            0               0
fbr                          4837 (  0.0)            0               0
fmuld                        2850 (  0.0)            0               0
orcc                          383 (  0.0)            0               0
sra                           241 (  0.0)            0               0
ldsb                          160 (  0.0)            0               0
mulx                           87 (  0.0)            0               0
stb                            31 (  0.0)            0               0
mov                            21 (  0.0)            0               0
```

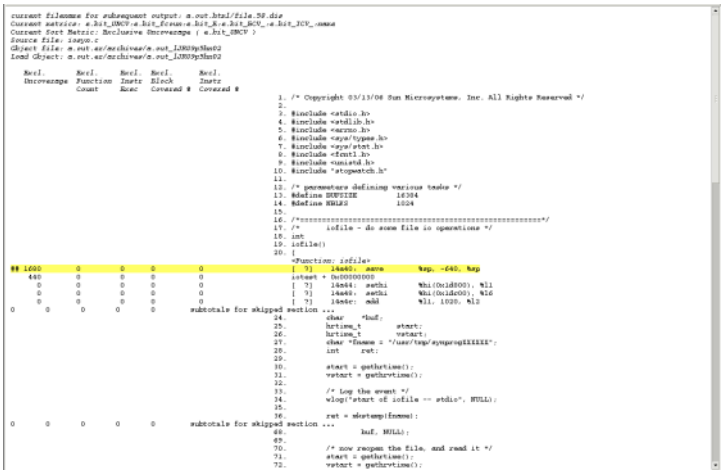
```
fdtox                                15 (  0.0)            0               0
============================================================
```

HTML カバレージレポートを理解する

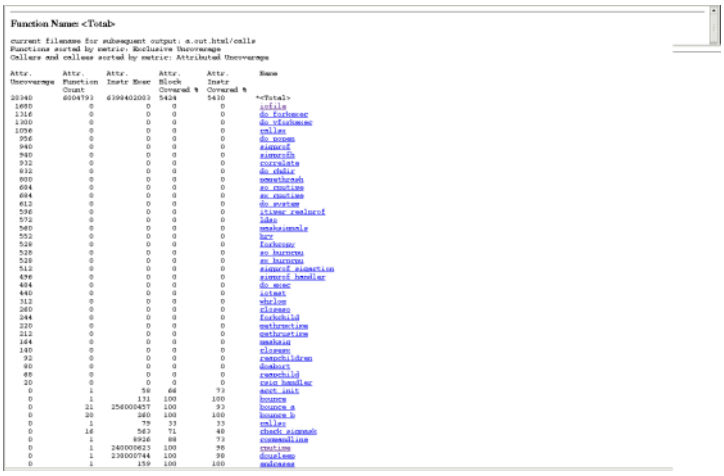
HTML レポートは、パフォーマンスアナライザに表示されるレポートに類似しています。



関数名のリンクまたは関数の `trimmed` のリンクをクリックすると、その関数の逆アセンブリデータが表示されます。



関数の Caller-callee リンクをクリックすると、呼び出し元-呼び出し先データが表示されます。



uncover 使用時の制限事項

このセクションでは、uncover 使用時の既知の制限事項について説明します。

注釈付きコードのみ計測可能

uncover ユーティリティーは、51 ページの「uncover を使用するための要件」の説明に従って準備されているコードのみを計測できます。注釈の付いていないコードは、バイナリにリンクされているアセンブリ言語コード、またはそのセクションに示されてるものより古いコンパイラまたはオペレーティングシステムでコンパイルされたモジュールから来ている場合があります。

uncover は、asm 文または .il テンプレートを含むアセンブリ言語モジュールまたは関数を計測できません。

コンパイラオプションは生成されるコードに影響を及ぼす

uncover は、次のいずれかのコンパイラオプションで構築されたバイナリと互換性がありません。

- -p
- -pg
- -qp
- -xpg
- -xlinkopt

機械命令はソースコードと異なる場合がある

uncover ユーティリティーは機械コード上で動作します。Uncover は機械命令のカバレージを検出し、このカバレージをソースコードと関連付けます。一部のソースコード文は関連した機械命令を持たないため、uncover がそのような文のカバレージを報告しないように見える場合があります。

例 4 簡単な例

次に示すコードの一部を考えてみましょう。

```
#define A 100
#define B 200
...
if (A>B) {
...
}
```

uncover が if 文に対して 0 以外の実行カウントを報告すると予想することがあります。しかし、コンパイラはこのコードを削除する可能性があります。uncover は計測時にそれを検出しないため、これらの命令に関してカバレージは報告されません。

例 5 デッドコードの例

次の例は、デッドコードを示しています。

```
1  void foo()
2  {
3     A();
4     return;
5     B();
6     C();
7     D();
8     return;
9   }
```

B、C、D の呼び出しは実行されることがないため、対応するアセンブリではこのコードが削除されています。

```
foo:
.L900000109:
/* 000000          2 */         save    %sp,-96,%sp
/* 0x0004          3 */         call    A       ! params =      ! Result =
/* 0x0008            */         nop
/* 0x000c          8 */         ret     ! Result =
/* 0x0010            */         restore %g0,%g0,%g0
```

したがって、5-6 行目に関してカバレージは報告されません。

```
   Excl.       Excl.     Excl.  Excl.      Excl.
Uncoverage  Function  Instr  Block      Instr
Count     Exec   Covered %  Covered %
1.  void foo()
## 0           1         1      100        100              2.   {
<Function: foo
## 0           0         2        0          0              3.      A();
4.      return;
5.      B();
6.      C();
7.      D();
8.     return;
## 0           0         2        0          0              9.    }
```

例 6 冗長なコードの例

次の例は、冗長なコードを示しています。

```
      1  int g;
2  int foo() {
3    int x;
4    x = g;
5    for (int i=0; i<100; i++)
6        x++;
7    return x;
8  }
```

低い最適化レベルでは、コンパイラがすべての行にコードを生成する可能性があります。

```
foo:
.L900000107:
/* 000000          3 */         save    %sp,-112,%sp
/* 0x0004          5 */         sethi   %hi(g),%l1
/* 0x0008            */         ld      [%l1+%lo(g)],%l3 ! volatile
/* 0x000c            */         add     %l1,%lo(g),%l2
/* 0x0010          6 */         st      %g0,[%fp-12]
/* 0x0014          5 */         st      %l3,[%fp-8]
/* 0x0018          6 */         ld      [%fp-12],%l4
/* 0x001c            */         cmp     %l4,100
/* 0x0020            */         bge,a,pn        %icc,.L900000105
/* 0x0024          8 */         ld      [%fp-8],%l1
.L17:
/* 0x0028          7 */         ld      [%fp-8],%l1
.L900000104:
/* 0x002c          6 */         ld      [%fp-12],%l3
/* 0x0030          7 */         add     %l1,1,%l2
/* 0x0034            */         st      %l2,[%fp-8]
/* 0x0038          6 */         add     %l3,1,%l4
/* 0x003c            */         st      %l4,[%fp-12]
/* 0x0040            */         ld      [%fp-12],%l5
/* 0x0044            */         cmp     %l5,100
/* 0x0048            */         bl,a,pn %icc,.L900000104
/* 0x004c          7 */         ld      [%fp-8],%l1
/* 0x0050          8 */         ld      [%fp-8],%l1
.L900000105:
/* 0x0054          8 */         st      %l1,[%fp-4]
/* 0x0058            */         ld      [%fp-4],%i0
/* 0x005c            */         ret     ! Result =  %i0
/* 0x0060            */         restore %g0,%g0,%g0
```

高い最適化レベルでは、実行可能なソース行のほとんどに、対応する命令がありません。

```
foo:
/* 000000          5 */         sethi   %hi(g),%o5
/* 0x0004            */         ld      [%o5+%lo(g)],%o4
/* 0x0008          8 */         retl    ! Result =  %o0
/* 0x000c          5 */         add     %o4,100,%o0
```

そのため、一部の行に関してカバレージは報告されません。

```
     Excl.       Excl.     Excl.  Excl.      Excl.
Uncoverage  Function  Instr  Block      Instr
Count     Exec   Covered %  Covered %
1. int g;
0           0         0        0          0               2. int foo() {
<Function foo>
3.   int x;
4.   x = g;

Source loop below has tag L1
Induction variable substitution performed on L1
L1 deleted as dead code
```

```
## 0           1         3      100        100              5.   for (int i=0; i<100; i++)
6.        x++;
7.    return x;
0           0         1        0          0              8. }
```

索引

S

U