Oracle Solaris Studio 12.3：Discover 和 Uncover 用户指南

文件号码 E26445
2011 年 12 月，E26445-01

ORACLE

120220@25097

目录

前言

《Oracle Solaris Studio 2.3：Discover 和 Uncover 用户指南》提供了如何使用内存错误搜索工具 (Discover) 以二进制形式查找内存有关的错误，以及使用代码覆盖工具 (Uncover) 测量应用程序代码覆盖的说明。

受支持的平台

此 Oracle Solaris Studio 发行版支持使用以下体系结构的平台：运行 Oracle Solaris 操作系统的 SPARC 系列处理器体系结构，以及运行 Oracle Solaris 或特定 Linux 系统的 x86 系列处理器体系结构。

本文档使用以下术语说明 x86 平台之间的区别：

- "x86" 泛指 64 位和 32 位的 x86 兼容产品系列。
- "x64" 指特定的 64 位 x86 兼容 CPU。
- “32 位 x86”指出了有关基于 x86 的系统的特定 32 位信息。

在 SPARC 和 x86 系统中，特定于 Linux 系统的信息仅指受支持的 Linux x86 平台，而特定于 Oracle Solaris 系统的信息仅指受支持的 Oracle Solaris 平台。

有关受支持的硬件平台和操作系统发行版的完整列表，请参见《Oracle Solaris Studio 12.3 发行说明》。

Oracle Solaris Studio 文档

可以查找 Oracle Solaris Studio 软件的完整文档，如下所述：

- 产品文档位于 Oracle Solaris Studio 文档 Web 站点，包括发行说明、参考手册、用户指南和教程。
- 代码分析器、性能分析器、线程分析器、dbxtool、DLight 和 IDE 的联机帮助可以在这些工具中通过 "Help"（帮助）菜单以及 F1 键和许多窗口和对话框上的 "Help"（帮助）按钮获取。
- 命令行工具的手册页介绍了工具的命令选项。

相关的第三方 Web 站点引用

本文档引用了第三方 URL，以用于提供其他相关信息。

注 – Oracle 对本文档中提到的第三方 Web 站点的可用性不承担任何责任。对于此类站点或资源中的（或通过它们获得的）任何内容、广告、产品或其他资料，Oracle 并不表示认可，也不承担任何责任。对于因使用或依靠此类站点或资源中的（或通过它们获得的）任何内容、产品或服务而造成的或连带产生的实际或名义损坏或损失，Oracle 概不负责，也不承担任何责任。

开发者资源

对于使用 Oracle Solaris Studio 的开发者，可访问 Oracle 技术网 Web 站点来查找以下资源：

- 有关编程技术和最佳做法的文章
- 软件最新发行版完整文档的链接
- 有关支持级别的信息
- 用户论坛。

获取 Oracle 支持

Oracle 客户可通过 My Oracle Support 获取电子支持。有关信息，请访问 `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info`，或访问 `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs`（如果您听力受损）。

印刷约定

下表介绍了本书中的印刷约定。

表 P–1 印刷约定

字体或符号	含义	示例
`AaBbCc123`	命令、文件和目录的名称；计算机屏幕输出	编辑 `.login` 文件。 使用 `ls -a` 列出所有文件。 `machine_name% you have mail.`
`AaBbCc123`	用户键入的内容，与计算机屏幕输出的显示不同	`machine_name%` **`su`** `Password:`

表 P–1 印刷约定 （续）

字体或符号	含义	示例
aabbcc123	要使用实名或值替换的命令行占位符	删除文件的命令为 `rm` *filename*。
AaBbCc123	保留未译的新词或术语以及要强调的词	这些称为 *Class* 选项。 **注意**：有些强调的项目在联机时以粗体显示。
新词术语强调	新词或术语以及要强调的词	**高速缓存**是存储在本地的副本。 请**勿**保存文件。
《书名》	书名	阅读《用户指南》的第 6 章。

命令中的 shell 提示符示例

下表显示了 Oracle Solaris OS 中包含的缺省 UNIX shell 系统提示符和超级用户提示符。请注意，在命令示例中显示的缺省系统提示符可能会有所不同，具体取决于 Oracle Solaris 发行版。

表 P–2 shell 提示符

shell	提示符
Bash shell、Korn shell 和 Bourne shell	`$`
Bash shell、Korn shell 和 Bourne shell 超级用户	`#`
C shell	`machine_name%`
C shell 超级用户	`machine_name#`

◆ ◆ ◆ 第 1 章

1 简介

《Oracle Solaris Studio 12.3 Discover 和 Uncover 用户指南》提供了下列工具的用法说明：

内存错误搜索工具 (Discover)

内存错误搜索工具 (Discover) 软件是用于检测内存访问错误的高级开发工具。Discover 使用通过 Sun Studio 12 Update 1、Oracle Solaris Studio 12.2 或 Oracle Solaris Studio 12.3 编译器编译的二进制文件，或通过 GCC for Sun Systems 4.2.0 版（或更高版本）编译器编译的二进制文件。Discover 适用于运行 Solaris 10 10/08 操作系统（或更高的 Solaris 10 Update 版本）或 Oracle Solaris 11 的系统。

程序中与内存相关的错误极难发现。通过 Discover，您可以定位问题在源代码中的确切位置，从而轻松地找到此类错误。例如，如果您的程序分配了一个数组但未将其初始化，然后尝试从一个数组位置执行读取操作，程序可能会出现异常行为。当您以正常方式运行程序时，Discover 可以捕捉到此问题。

Discover 可以检测到的其他错误包括：

- 对未分配的内存执行读写
- 访问超出分配数组边界的内存
- 不正确地使用释放的内存
- 释放错误的内存块
- 内存泄漏

由于 Discover 是在程序执行期间动态捕捉并报告内存访问错误，因此，如果运行时用户代码的某个部分未执行，则不会报告该部分的错误。

Discover 简单易用。编译器所准备的任何二进制文件（即使是完全优化的二进制文件）均可使用单个命令进行检测，然后以正常方式运行。运行期间，Discover 会生成内存异常报告，您可以在 Web 浏览器中以文本文件或 HTML 格式查看报告。

代码覆盖工具 (Uncover)

Uncover 是一个简单易用的命令行工具，用于度量应用程序的代码覆盖。代码覆盖是软件测试的重要组成部分。该工具提供了测试时执行或未执行的代码区域的相关信息，使您可以改进测试套件以测试更多代码。Uncover 可以在函数、语句、基本块或指令级别报告覆盖信息。

Uncover 提供了一个称为“未覆盖”的独特功能，使您可以快速找到未测试的主要功能区域。与其他类型的检测相比，Uncover 代码覆盖工具的其他优势包括 :

- 相对于未检测的代码而言，性能只有些许的下降。
- 由于 Uncover 使用二进制文件，因此，它可以处理任何优化的二进制文件。
- 可以通过检测随附的二进制文件来进行度量。要进行覆盖测试，无需以不同的方式构建应用程序。
- Uncover 提供了一套检测二进制文件、运行测试和显示结果的简单过程。
- Uncover 是多线程安全的，并且是多进程安全的。

◆ ◆ ◆ 第 2 章

2 内存错误搜索工具 (Discover)

内存错误搜索工具 (Discover) 软件是用于检测内存访问错误的高级开发工具。

本章包括有关下列内容的信息：

Discover 的使用要求

必须正确准备二进制文件

Discover 使用通过 Sun Studio 12 Update 1、Oracle Solaris Studio 12.2 或 Oracle Solaris Studio 12.3 编译器编译的二进制文件，或通过 GCC for Sun Systems 4.2.0 版（或更高版本）编译器编译的二进制文件。Discover 适用于运行 Solaris 10 10/08 操作系统（或更高的 Solaris 10 Update 版本）或 Oracle Solaris 11 的基于 SPARC 或基于 x86 的系统。

如果未满足这些要求，Discover 会发出错误，并且不检测二进制文件。但是，您可以使用 `-l` 选项（请参见第 16 页中的 “检测选项”）来检测未满足这些要求的二进制文件，并运行该二进制文件来检测有限数量的错误。

按照说明进行编译的二进制文件包括一些称为注释的信息，用于帮助 Discover 正确地检测二进制文件。添加这些少量信息不会影响二进制文件的性能或运行时内存使用情况。

通过在编译二进制文件时使用 -g 选项生成调试信息，Discover 可以在报告错误和警告的同时显示源代码和行号信息，并可以生成更准确的结果。如果在编译二进制文件时未使用 -g 选项，Discover 将仅显示相应计算机级别指令的程序计数器。另外，使用 -g 选项进行编译可帮助 Discover 生成更准确的报告（请参见第 32 页中的 “解释 Discover 错误消息”）。

不能使用使用预装或审计的二进制文件

由于 Discover 使用运行时链接程序的某些特殊功能，因此您不能将其用于使用预装或审计的二进制文件。

如果程序需要设置 LD_PRELOAD 环境变量，则可能无法与 Discover 正确配合，因为 Discover 需要插入某些系统函数，如果函数已预先装入，则无法执行此操作。

同样，如果程序使用了运行时审计（二进制文件通过 -p 选项或 -P 选项进行链接，或者需要设置 LD_AUDIT 环境变量），则此审计将与 Discover 使用的审计相冲突。如果二进制文件链接到审计，Discover 将在检测时失败。如果在运行时设置了 LD_AUDIT 环境变量，结果将无法确定。

可以使用重新定义标准内存分配函数的二进制文件

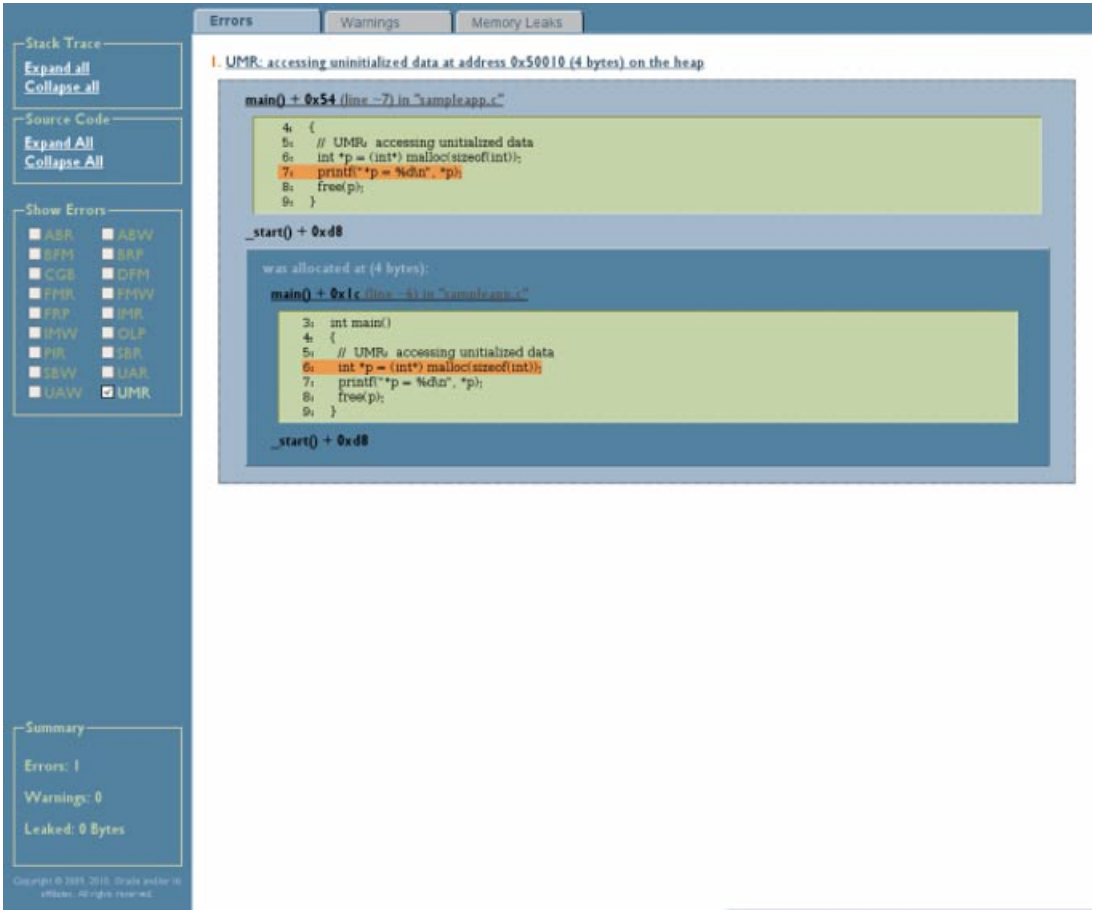
Discover 支持重新定义以下标准内存分配函数的二进制文件：malloc()、calloc()、memalign()、valloc() 和 free()。

快速入门

下例说明了如何准备程序、使用 Discover 对其进行检测、运行程序，以及针对检测到的内存访问错误生成一个报告。此示例使用了一个访问未初始化数据的简单程序。

```
% 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 输出显示了在何处使用了未初始化内存、将其分配至了何处，以及结果摘要。



检测准备好的二进制文件

准备好目标二进制文件后，下一步就是对其进行检测。检测功能会在至关重要的位置添加代码，以便在运行二进制文件时 Discover 能够跟踪内存操作。

可使用 `discover` 命令检测二进制文件。例如，以下命令将检测二进制文件 `a.out`，并使用检测过的 `a.out` 来覆盖输入 `a.out`：

```
discover a.out
```

当您运行检测过的二进制文件时，Discover 将监视该程序对内存的使用情况。在运行期间，Discover 会向 HTML 文件（在本例中，缺省情况下该文件为 `a.out.html`）中写入一份报告，其中详细列出了内存访问错误，可使用 Web 浏览器来查看该文件。检测二进制文件时，可以使用 `-w` 选项请求将报告写入 ASCII 文件或 `stderr`。

可以使用 `-n` 选项指定希望 Discover 对二进制文件执行只写检测。

当 Discover 检测二进制文件时，如果 Discover 发现任何由于未注释而导致无法检测的代码，Discover 将显示如下警告：

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

无注释代码可能来自链接到二进制文件中的汇编语言代码，或者来自使用早于第 11 页中的 “必须正确准备二进制文件”中所列版本的编译器或操作系统编译的模块。

缓存共享库

当 Discover 检测某个二进制文件时，将向该二进制文件中添加代码。当在运行时装入相关的共享库时，这些代码会配合运行时链接程序对这些共享库进行检测。检测过的库存储在高速缓存中；如果原始库自上次检测以来未发生更改，可以重新使用这些库。缺省情况下，高速缓存目录为 `$HOME/SUNW_Bit_Cache`。您可以使用 `-D` 选项来更改该目录。

检测共享库

如果整个程序（包括所有共享库）均已检测，Discover 将生成最准确的结果。缺省情况下，Discover 仅检查并报告可执行文件中的内存错误。可以使用 `-c` 选项指定希望 Discover 检查相关共享库和通过 `dlopen()` 动态打开的库中的错误。可以使用 `-n` 选项指定希望 Discover 跳过检查可执行文件中的错误。

如果使用 `-c` 选项避免检查特定库中的错误，则 Discover 不会报告该库中的任何错误。不过，Discover 需要跟踪整个地址空间的内存状态，以正确检测内存错误，因此 Discover 会记录整个程序（包括所有的共享库）中的分配和内存初始化。

应根据第 11 页中的 “必须正确准备二进制文件”中的说明准备程序使用的所有共享库。缺省情况下，如果运行时链接程序遇到一个未准备好的库，会发生致命错误。但是，您可以让 Discover 忽略一个或多个库。

忽略库

某些库可能无法准备，或者出于其他某种原因无法检测。要解决此问题，您可以使用 `-s`、`-T` 或 `-N` 选项（请参见第 16 页中的 “检测选项”）或 `bit.rc` 文件中的规范（请参见第 17 页中的 “bit.rc 初始化文件”）让 Discover 忽略这些库，但是，这样准确性会有所降低。

如果某个库无法检测，且无法标识为可忽略，Discover 将在检测时失败，或者程序将在运行时失败并出现错误消息。

缺省情况下，Discover 使用 `bit.rc` 系统文件中的规范将未准备好的系统库和编译器提供的库设置为可忽略。由于 Discover 了解最常用库的内存特征，因此，对准确性的影响微乎其微。

命令行选项

您可以将以下选项与 `discover` 命令结合使用来检测二进制文件。

输出选项

`-a` 将错误数据写入 *binary_name*`.analyze/dynamic` 目录以供代码分析器使用。

`-b` *browser* 运行检测过的程序时，将自动启动 Web 浏览器 *browser*（缺省情况下关闭）。

`-o` *file* 将检测过的二进制文件写入 *file*。缺省情况下，检测过的二进制文件会覆盖输入二进制文件。

`-w` *text_file* 将 Discover 的二进制文件报告写入 *text_file*。该文件是在您运行检测过的二进制文件时创建的。如果 *text_file* 是相对路径名，则该文件位于与您运行检测过的二进制文件时所在工作目录相对的位置。要使每次运行二进制文件时该文件名均具有唯一性，请在文件名中添加字符串 `%p`，要求 Discover 运行时包含进程 ID。例如，选项 `-w report.%p.txt` 将生成一个文件名为 `report.`*process_id*`.txt` 的报告文件。如果文件名中包含多处 `%p`，只有第一个实例会替换为进程 ID。

如果不指定此选项或 `-H` 选项，会以 HTML 格式将报告写入 *output_file*`.html`，其中 *output_file* 是检测过的二进制文件的基本名称。该文件位于您运行检测过的二进制文件时所在的工作目录中。

您可以同时指定此选项和 `-H` 选项，同时以文本和 HTML 格式写入报告。

`-H` *html_file* 以 HTML 格式将 Discover 的二进制文件报告写入 *html_file*。此文件是在您运行检测过的二进制文件时创建的。如果 *html_file* 是相对路径名，则该文件位于与您运行检测过的二进制文件时所在工作目录相对的位置。要使每次运行二进制文件时该文件名均具有唯一性，请在文件名中添加字符串 `%p`，要求 Discover 运行时包含进程 ID。例如，选项 `-H report.%p.html` 会生成一个文件名为 `report.`*process_id*`.html` 的报告文件。如果文件名中包含多处 `%p`，只有第一个实例会替换为进程 ID。

如果不指定此选项或 `-w` 选项，会以 HTML 格式将报告写入 *output_file*`.html`，其中 *output_file* 是检测过的二进制文件的基本名称。该文件位于您运行检测过的二进制文件时所在的工作目录中。

您可以同时指定此选项和 -w 选项，同时以文本和 HTML 格式写入报告。

-e *n* 仅在报告中显示 *n* 个内存错误（缺省情况下显示所有错误）。

-E *n* 仅在报告中显示 *n* 个内存泄漏（缺省情况下显示 100 个）。

-f 在报告中显示偏移（缺省情况下隐藏偏移）。

-m 在报告中显示改编名称（缺省情况下显示取消改编名称）。

-S *n* 仅在报告中显示 *n* 个堆栈帧（缺省情况下显示 8 个）。

检测选项

-c [- | *library* | *file*] 检查所有库中、指定的 *library* 中或指定的 *file* 中所列出的库中的错误。

-n 不检查可执行文件中的错误。

-l 在轻量模式下运行 Discover。使用此选项可以更快地执行程序，并且无需特意根据第 11 页中的 “必须正确准备二进制文件”中的说明准备程序，但只能检测到有限数量的错误。

-F [parent | child] 指定希望运行二进制文件时，如果使用 Discover 检测过的该二进制文件派生时会出现的情况。缺省情况下，Discover 继续从父进程收集内存访问错误数据。如果希望 Discover 在派生之后从子进程收集内存访问数据，请指定 -F child。

-i 执行检测以便使用线程分析器进行数据争用检测。如果使用此选项，仅在运行时执行数据争用检测，而不执行其他任何内存检查。必须使用 collect 命令运行检测过的二进制文件，以生成可以在性能分析器中查看的实验（请参见《Oracle Solaris Studio 12.3 线程分析器用户指南》。

-s 尝试检测一个无法检测的二进制文件时发出警告，但不标出错误。

-T 仅检测命名的二进制文件。在运行时不检测任何相关的共享库。

-N *library* 不检测与前缀 *library* 匹配的任何相关共享库。如果库名称的前几个字符与 *library* 匹配，则忽略该库。如果 *library* 以 / 开头，则根据库的绝对完整路径名进行匹配。否则，根据库的基本名称进行匹配。

-K 不读取 bit.rc 初始化文件（请参见第 17 页中的 “bit.rc 初始化文件”）。

缓存选项

`-D` *cache_directory*	将 *cache_directory* 用作存储缓存的检测过的二进制文件的根目录。缺省情况下，高速缓存目录为 `$HOME/SUNW_Bit_Cache`。
`-k`	强制重新检测高速缓存中找到的所有库。

其他选项

`-h` 或 `-?`	帮助。输出简短的用法消息并退出。
`-v`	详细。输出 Discover 正在执行的操作的日志。重复使用该选项可获得更多信息。
`-V`	输出 Discover 版本信息并退出。

bit.rc 初始化文件

Discover 在启动时，会通过读取一系列 `bit.rc` 文件来初始化自身的状态。系统文件 *Oracle_Solaris_Studio_installation_directory* `/prod/lib/postopt/bit.rc` 为某些变量提供了缺省值。Discover 先读取此文件，然后依次读取 `$HOME/.bit.rc`（如果存在）和 *current_directory*`/.bit.rc`（如果存在）。

`bit.rc` 文件包含用于设置某些变量以及在某些变量中进行附加和删除的命令。当 Discover 读取 `set` 命令时，会放弃变量的前一个值（如果有）。当读取 `append` 命令时，会将参数附加到变量的现有值（该参数置于冒号分隔符后面）。当读取 `remove` 命令时，将从变量的现有值中删除参数及其冒号分隔符。

`bit.rc` 文件中设置的变量包括检测时要忽略的库列表，以及计算库中无注释（未准备）代码百分比时要忽略的函数或函数前缀的列表。

有关更多信息，请参阅 `bit.rc` 系统文件头中的注释。

SUNW_DISCOVER_OPTIONS 环境变量

可以通过将 `SUNW_DISCOVER_OPTIONS` 环境变量设置为命令行选项 `-b`、`-e`、`-E`、`-f`、`-F`、`-H`、`-l`、`-L`、`-m`、`-S` 和 `-w` 的列表，以更改检测过的二进制文件的运行时行为。例如，如果要将报告的错误数更改为 50 并将报告中的堆栈深限制为 3，可以将此环境变量设置为 `-e 50 -s 3`。

SUNW_DISCOVER_FOLLOW_FORK_MODE 环境变量

缺省情况下，如果运行已使用 Discover 检测过的二进制文件时，该二进制文件发生派生，则 Discover 会继续从父进程收集内存访问错误数据。如果希望 Discover 在派生之后从子进程收集内存访问数据，请设置 `SUNW_DISCOVER_FOLLOW_FORK_MODE` 环境变量。

运行检测过的二进制文件

使用 Discover 检测二进制文件后，可以如常运行二进制文件。通常，如果特定的输入组合导致程序行为异常，您可以使用 Discover 对其进行检测，然后使用相同的输入运行程序，以调查潜在的内存问题。检测过的程序在运行时，Discover 以选定的格式（文本和/或 HTML）会将发现的与任何内存问题相关的信息写入到指定的输出文件。有关解释报告的信息，请参见第 18 页中的“分析 Discover 报告”。

由于检测会产生系统开销，因此，程序经过检测后运行速度会显著减慢。根据内存访问的频率，运行速度最多可能会慢 50 倍。

分析 Discover 报告

Discover 报告提供的信息可帮助您有效地定位并修复源代码中的问题。

缺省情况下，该报告以 HTML 格式写入到 *output_file*`.html`，其中 *output_file* 是检测过的二进制文件的基本名称。该文件位于您运行检测过的二进制文件时所在的工作目录中。

检测二进制文件时，可以使用 `-H` 选项请求将 HTML 输出写入指定的文件，或使用 `-w` 选项请求将其写入文本文件（请参见第 15 页中的“命令行选项”）。

检测二进制文件后，可以更改第 17 页中的“`SUNW_DISCOVER_OPTIONS` 环境变量”中报告的 `-H` 和 `-w` 选项的设置。例如，如果希望在以后运行程序时将报告写入不同的文件，可以执行此操作。

分析 HTML 报告

使用 HTML 报告格式可以对程序进行交互分析。开发者可以通过使用电子邮件或者通过发布到 Web 页上来轻松共享 HTML 格式的数据。HTML 报告与 JavaScript 交互功能相结合，使得用户可以轻松浏览 Discover 消息。

"Errors"（错误）选项卡（请参见第 19 页中的“使用 "Errors"（错误）选项卡”）、"Warnings"（警告）选项卡（请参见第 22 页中的“使用 "Warnings"（警告）选

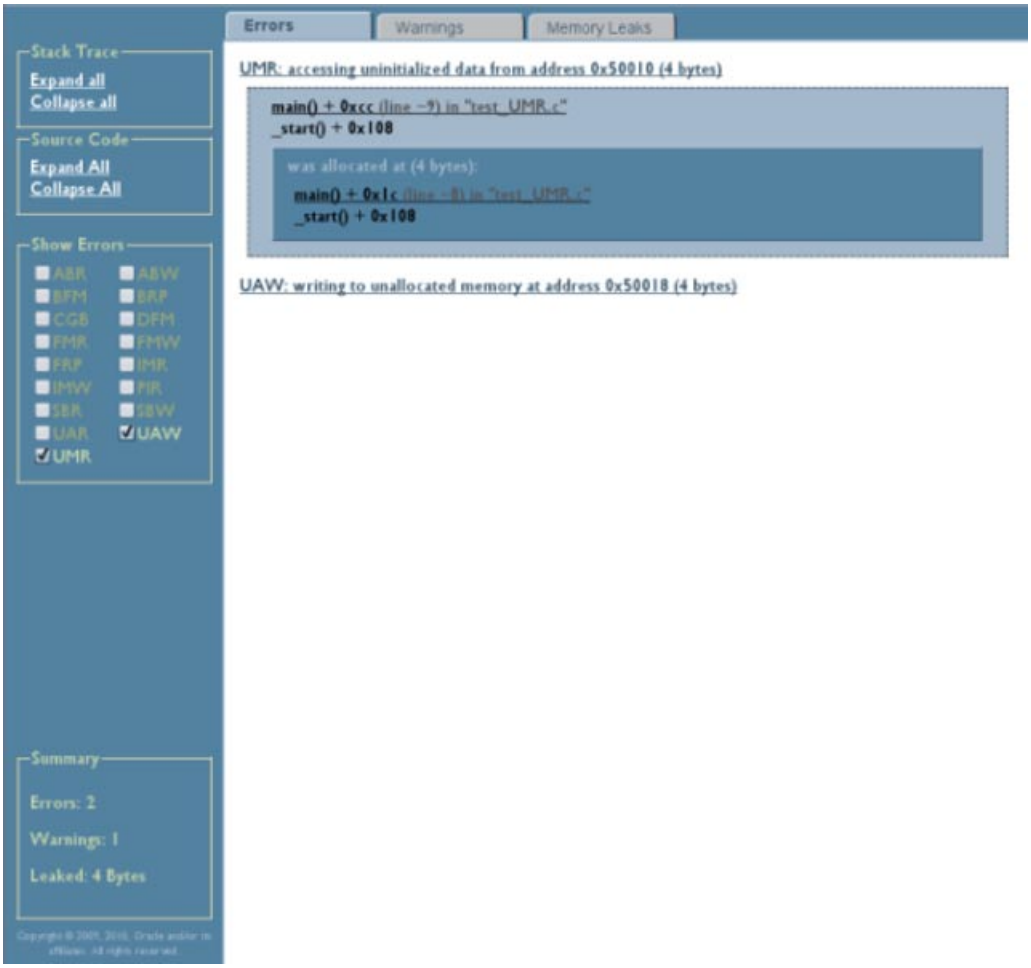
项卡”）和 "Memory Leaks"（内存泄漏）选项卡（请参见第 23 页中的 “使用 "Memory Leaks"（内存泄漏）选项卡”）分别用于浏览错误消息、警告消息和内存泄漏报告。

使用左侧的控制面板（请参见第 25 页中的 “使用控制面板”）可以更改当前显示在右侧的选项卡内容。

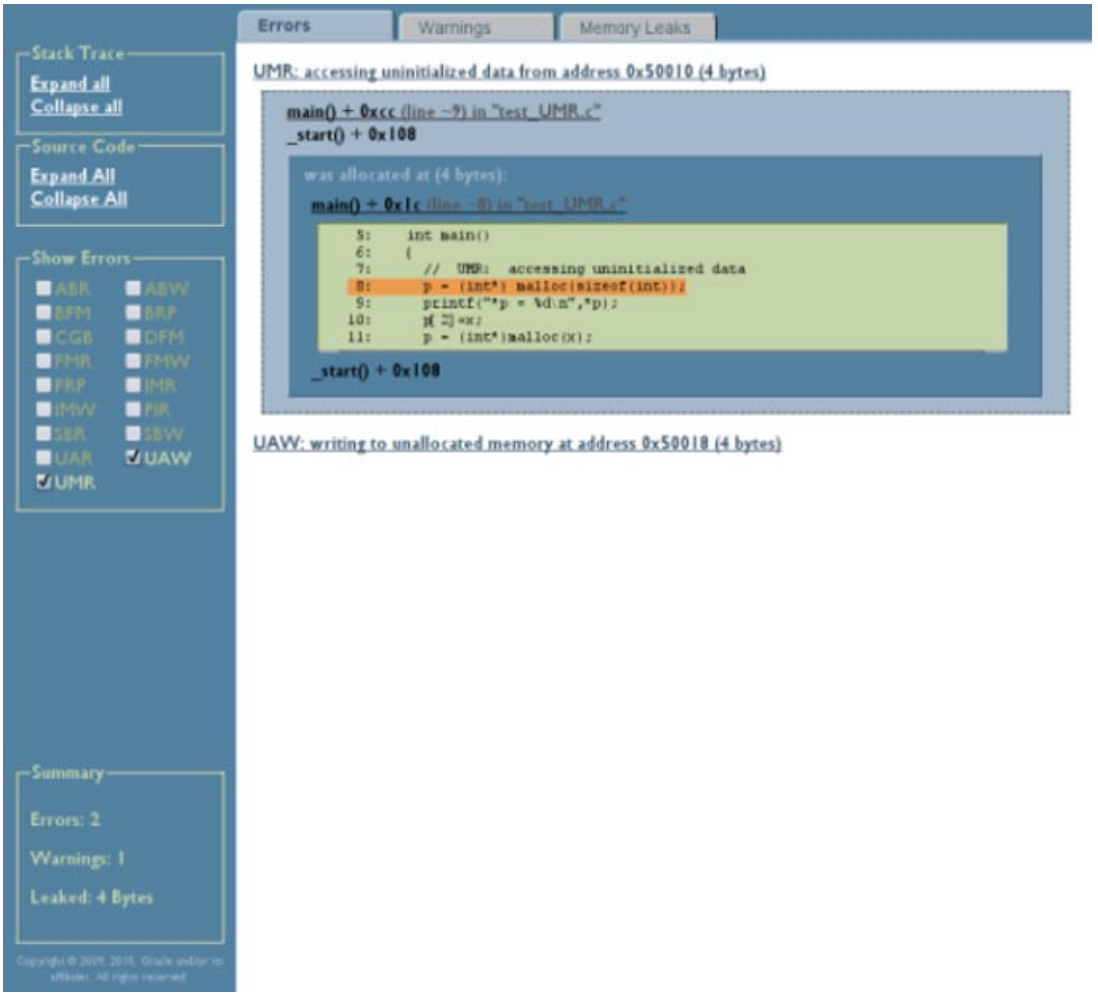
使用 "Errors"（错误）选项卡

在浏览器中首次打开 HTML 报告时，"Errors"（错误）选项卡处于选中状态，并显示执行检测过的二进制文件期间发生的内存访问错误列表。

单击某个错误时，将显示发生该错误时的堆栈跟踪：

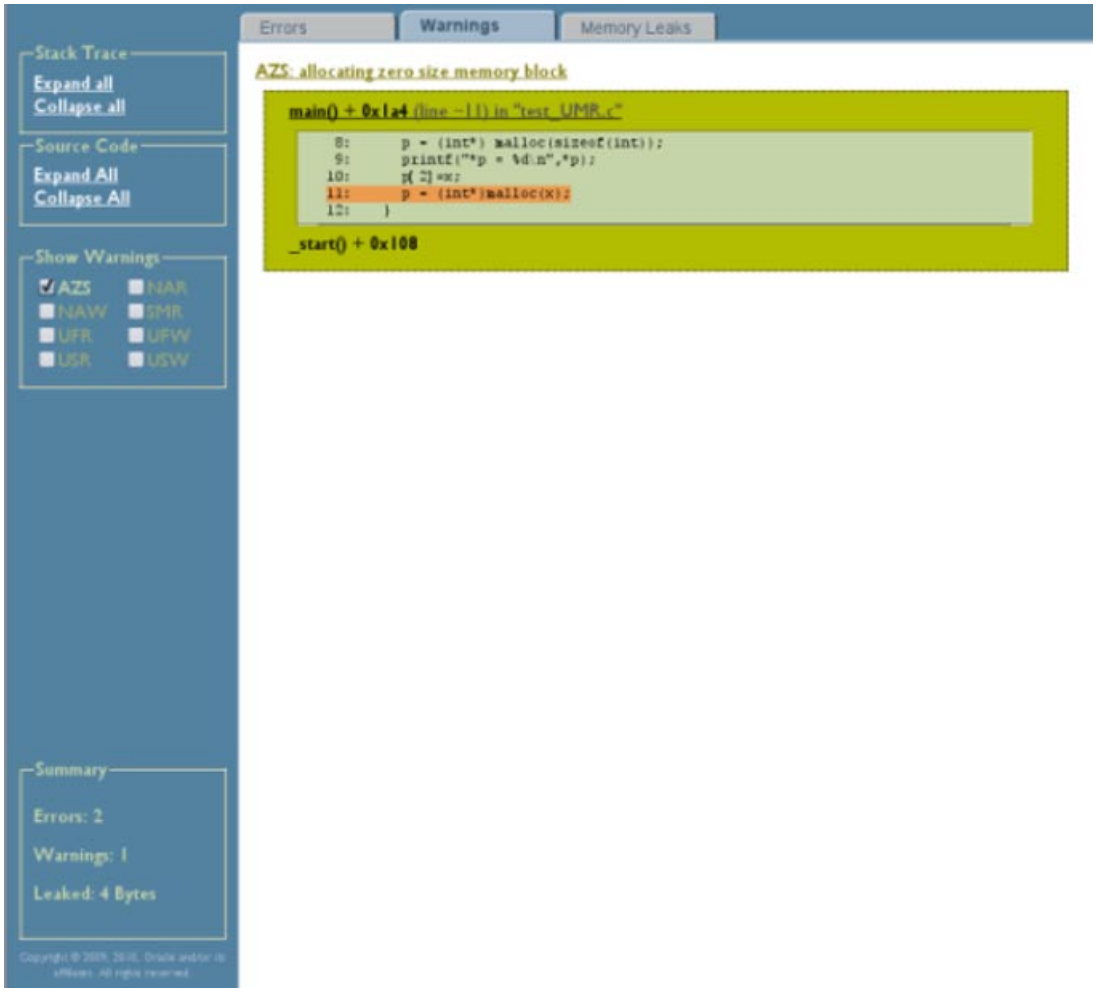


如果代码是使用 -g 选项编译的，可以通过单击堆栈跟踪中的每个函数来查看相应函数的源代码：



使用 "Warnings"（警告）选项卡

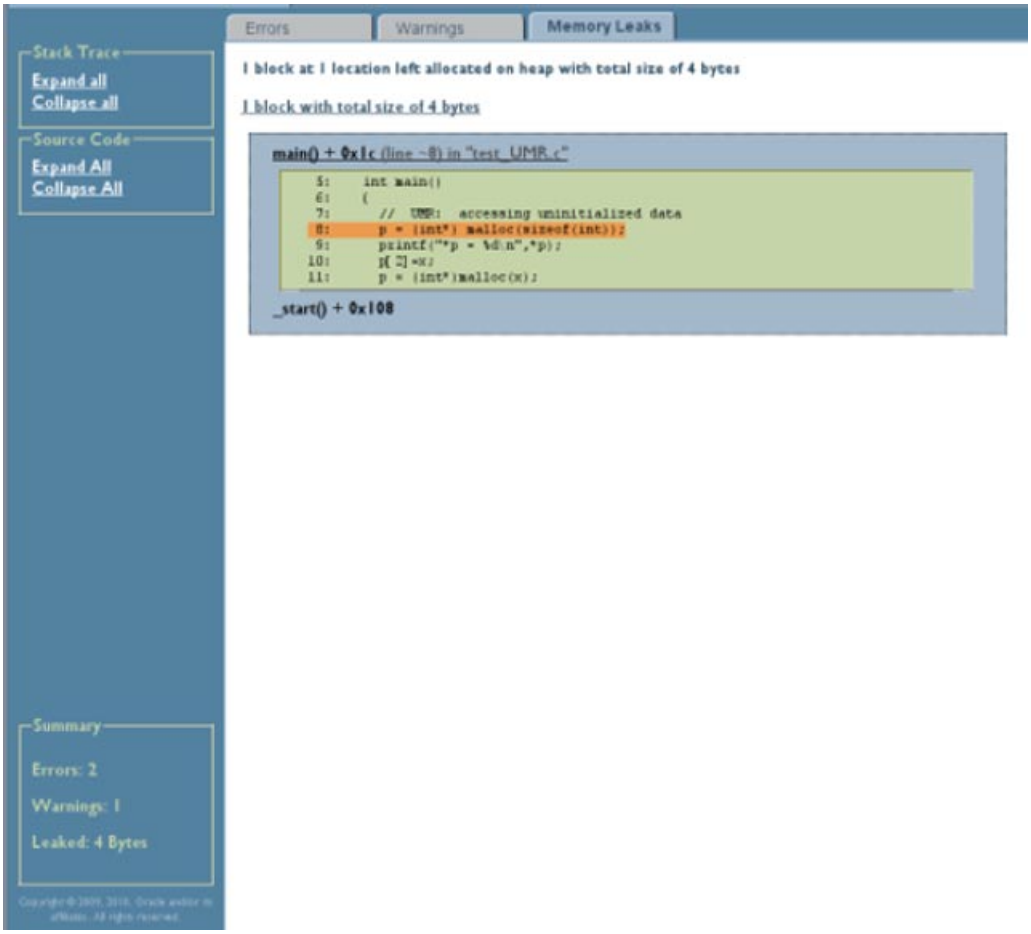
"Warnings"（警告）选项卡显示了有关可能的访问错误的所有警告消息。单击某个警告时，将显示发生该警告时的堆栈跟踪。如果代码是使用 -g 选项编译的，可以通过单击堆栈跟踪中的每个函数来查看相应函数的源代码。



使用 "Memory Leaks"（内存泄漏）选项卡

"Memory Leaks"（内存泄漏）选项卡的顶部显示程序运行结束时仍保持已分配状态的总块数，并在下方列出了相应的块。

单击某个块时，将显示该块的堆栈跟踪。如果代码是使用 `-g` 选项编译的，可以通过单击堆栈跟踪中的每个函数来查看相应函数的源代码。



使用控制面板

要查看所有错误、警告和内存泄漏的堆栈跟踪，请在控制面板的 "Stack Traces"（堆栈跟踪）部分中单击 "Expand All"（全部展开）。要查看所有函数的源代码，请在控制面板的 "Source Code"（源代码）部分中单击 "Expand All"（全部展开）。

要隐藏所有错误、警告和内存泄漏的堆栈跟踪或源代码，请单击相应的 "Collapse All"（全部折叠）。

选择 "Errors"（错误）选项卡后，将显示控制面板的 "Show Errors"（显示错误）部分，可以用来控制要显示的错误类型。缺省情况下，所有检测到的错误的复选框均处于选中状态。要隐藏某种错误类型，请单击对应的复选框以移除复选标记。

选择 "Warning"（警告）选项卡后，将显示控制面板的 "Show Warnings"（显示警告）部分，可以用来控制要显示的警告类型。缺省情况下，所有检测到的警告的复选框均处于选中状态。要隐藏某种警告类型，请单击对应的复选框以移除复选标记。

控制面板的底部显示了报告摘要，其中列出了错误和警告总数，以及泄漏的内存量。

分析 ASCII 报告

按脚本进行处理时，或者无权访问 Web 浏览器时，适合使用 ASCII（文本）格式的 Discover 报告。下面是 ASCII 报告的一个示例。

```
$ 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)
```

该报告包括错误和警告消息，其后是摘要。

错误消息以 ERROR 一词开头，并包含一个三字母代码、一个 ID 号以及错误描述（本例中为 `writing to unallocated memory`）。其他详细信息包括访问的内存地址，以及读取或写入的字节数。在描述的后面，是发生错误时的堆栈跟踪，可以根据堆栈跟踪在进程生命周期中定位错误的位置。

如果程序是使用 `-g` 选项编译的，堆栈跟踪将包括源文件名称和行号。如果源文件可访问，则输出错误位置附近的源代码。每一帧中的目标源代码行均以符号 `=>` 表示。

如果同一内存位置重复出现字节数相同的同一错误类型，完整的消息（包括堆栈跟踪）仅输出一次。对于多次出现的每个相同错误，会统计后续的错误，并在报告的末尾列出重复计数（如下例中所示）。

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

如果出错内存访问的地址位于堆上，则在堆栈跟踪的后面输出相应堆块的信息。这些信息包括块的起始地址和大小，以及分配该块时的堆栈跟踪。如果该块已释放，将还包括解除分配点的堆栈跟踪。

警告消息的输出格式与错误消息相同，只不过是以 WARNING 一词开头。一般来说，这些消息只是提醒您注意那些不会影响应用程序正确性的情况，但提供的有用信息可帮助您改进程序。例如，分配大小为零的内存不会造成损害，但如果经常这样做可能会降低性能。

内存泄漏报告包含有关在堆上分配的、但程序退出时未释放的内存块的信息。下面是内存泄漏报告的一个示例。

```
$ 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]
...
```

标题下面的第一行汇总了在堆上保持已分配状态的堆块数及其总大小。报告的大小是从开发者的立场提供的，也就是说，不包括内存分配器的簿记系统开销。

内存泄漏摘要的后面输出了每个未释放堆块及其分配点堆栈跟踪的详细信息。该堆栈跟踪报告类似于前面介绍错误和警告消息时提到的堆栈跟踪报告。

Discover 报告的结尾是总摘要。该摘要报告了唯一警告和错误的数目，并在括号中提供了错误和警告的总数（包括重复项）。例如：

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

内存访问错误和警告

Discover 会检测和报告许多内存访问错误，并就可能是错误的访问向您发出警告。

内存访问错误

Discover 可检测到以下内存访问错误：

- ABR：数组越界读
- ABW：数组越界写
- BFM：释放错误的内存块
- BRP：错误的重新分配地址参数
- CGB：损坏的数组保护块
- DFM：双重释放内存
- FMR：读取释放的内存
- FMW：写入释放的内存
- FRP：释放的重新分配参数
- IMR：无效的内存读取
- IMW：无效的内存写入
- 内存泄漏
- OLP：重叠源和目标
- PIR：部分初始化的读取
- SBR：堆栈帧越界读
- SBW：堆栈帧越界写
- UAR：读取未分配的内存
- UAW：写入未分配的内存
- UMR：读取未初始化的内存

以下几部分列出了一些简单的示例程序，这些程序会生成上述某些错误。

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]);
```

ABW

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

BFM

```
//  BFM:  freeing wrong memory block
int *p = (int*) malloc(sizeof(int));
free(p+1);
```

BRP

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

DFM

```
// DFM is "double freeing memory"
int *p = (int*) malloc(sizeof(int));
free(p);
free(p);’
```

FMR

```
//  FMR is "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 is "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...
```

OLP

```
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

```
int a[2]={0,1};
printf("a[-10]=%d\n",a[-10]);
return 0;
```

SBW

```
int a[2]={0,1)'
a[-10]=2;
return 0;
```

UAR

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

UAW

```
// UAW is "writing to unallocated memory"
int *p = (int*) malloc(sizeof(int));
*(p+1) = 1;
```

UMR

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

内存访问警告

Discover 会报告下列内存访问警告 :

- AZS : 分配零大小
- SMR : 推测性未初始化内存读取

下面列出了一个会生成 AZS 警告的简单示例程序。

AZS

```
//  AZS:  allocating zero size memory block
int *p = malloc();
```

解释 Discover 错误消息

在某些情况下，Discover 可能会报告一个实际上不算错误的错误。这种情况称为“误报”。与同类工具相比，Discover 在检测时会分析代码以减少误报的次数，不过难免也会出现误报。以下几部分提供了一些提示，可能对识别 Discover 报告中的误报并尽可能地避免出现误报有所帮助。

部分初始化内存

以 C/C++ 语言编写的位字段可让您创建紧凑的数据类型。例如：

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

在本例中，结构成员 `my_struct.valid` 在内存中仅占用一位。但是，在 SPARC 平台上，CPU 只能以字节为单位修改内存，因此，只有装入含有 `struct.valid` 的整个字节才能访问或修改该结构成员。此外，编译器有时会认为一次装入多个字节（例如，由四个字节构成的计算机字）会更高效。当 Discover 检测到此类装入操作时，如果没有更多信息，会假定使用了所有四个字节。例如，字段 `my_struct.valid` 已初始化，但字段 `my_struct.c` 未初始化，如果装入了包含这两个字段的计算机字，则 Discover 会标记部分初始化内存读取 (PIR) 错误。

误报的另一个原因是位字段初始化。要写入某个字节的一部分，编译器必须首先生成用于装入该字节的代码。如果该字节不是在读取之前写入的，将生成未初始化内存读取 (UMR) 错误。

要避免位字段误报，请在编译时使用 `-g` 选项或 `-g0` 选项。这些选项为 Discover 提供了额外的调试信息，可帮助 Discover 识别位字段装入和初始化，从而可以排除大多数误报。如果出于某种原因无法使用 `-g` 选项编译，请使用 `memset()` 等函数初始化结构。例如：

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

可疑装入

有时，当装入的结果对某些程序路径无效时，编译器会从已知的内存地址生成装入。这种情况经常发生在 SPARC 平台上，因为此类装入指令可以放置在分支指令的延迟槽中。下面提供了一个示例代码片段：

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

根据此代码，编译器可能会生成如下代码：

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

假定本例中的函数 `foo()` 返回了 `0` 且未初始化 `i`。仍会从 `i` 生成装入，尽管它并未使用。但是，Discover 将会发现该装入，并报告装入了未初始化变量 (UMR)。

Discover 尽量使用数据流分析来识别此类情况，但有时无法检测到此类情况。

使用较低的优化级别进行编译可以减少这些类型的误报的发生。

未检测的代码

有时，Discover 无法完整地检测您的程序。某些代码可能来自无法重新编译的汇编语言源文件或第三方库，因此无法检测。Discover 并不知道未检测代码正在访问和修改的内存块。例如，假定某个第三方共享库中的某个函数初始化了一个内存块，主程序（检测过的程序）稍后读取了该内存块。由于 Discover 并不知道该库已初始化内存，后续读取操作将生成未初始化内存错误 (UMR)。

为解决此类问题，Discover API 中包含了下列函数：

```
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);
```

您可以从程序调用这些 API 函数，将具体事件告知 Discover，例如，向内存区写入 (`__ped_memory_write()`) 或从内存区读取 (`__ped_memory read()`)。对于这两种情况，内存区的起始地址将通过 `addr` 参数传递，内存区的大小通过 `size` 参数传递。将 `pc` 参数设置为 `0`。

使用 `__ped_memory_copy` 函数告知 Discover 正在将内存从一个位置复制到另一个位置。源内存的起始地址将通过 `src` 参数传递，目标区的起始地址通过 `dst` 参数传递，大小通过 `size` 参数传递。将 `pc` 参数设置为 `0`。

要使用 API，请在程序中将这些函数声明为弱函数。例如，在源代码中包含以下代码片段。

```
#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
```

API 函数是在内部 Discover 库中定义的，该库在检测时链接到程序。但是，如果未检测程序，便不会链接该库，这样，对 API 函数的所有调用都会导致应用程序挂起。因此，如果不是在 Discover 下运行程序，则必须禁用这些函数。另外，您也可以使用 API 函数的空定义创建一个动态库，并将其链接到程序。在此情况下，如果您未在 Discover 下运行程序，将使用您的库，但如果在 Discover 下运行程序，将自动调用实际的 API 函数。

使用 Discover 时的限制

仅检测有注释的代码

Discover 只能检测按照第 11 页中的 “必须正确准备二进制文件”中的说明准备的代码。无注释代码可能来自链接到二进制文件中的汇编语言代码，或者来自使用早于该部分中所列版本的编译器或操作系统编译的模块。

在准备时，特别要排除包含 `asm` 语句或 `.il` 模板的汇编语言模块和函数。

计算机指令可能不同于源代码

Discover 处理计算机代码。该工具可检测到装入和存储等计算机指令中的错误，并将这些错误与源代码相关联。某些源代码语句没有关联的计算机指令，因此，看上去 Discover 没有检测到明显的用户错误。例如，请看以下 C 语言代码片段：

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

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

```
return;
```

由于内存未初始化，读取 p 所指向的地址处存储的值就是一个潜在的用户错误。但是，优化编译器可以检测到未使用变量 i，因此，不会生成让语句读取内存并将其分配到 i 的代码。在此情况下，Discover 不会报告使用了未初始化内存 (UMR)。

编译器选项影响生成的代码

编译器生成的代码不一定始终如您所愿。由于编译器生成的代码因所使用的编译器选项（包括 -O*n* 优化选项）的不同而异，因此，Discover 报告的错误也可能有所不同。例如，在 -O1 优化级别上生成的代码中报告的错误可能不会出现于在 -O4 优化级别上生成的代码中。

系统库可能会影响报告的错误

系统库是随操作系统一起预装的，无法重新编译以进行检测。Discover 为标准 C 库 (libc.so) 中的公用函数提供支持；也就是说，Discover 知道这些函数访问或修改的内存。但是，如果应用程序使用了其他系统库，可能会在 Discover 报告中出现误报。如果出现误报，可以从代码调用 Discover API 来消除误报。

定制内存管理可能会影响数据的准确性

Discover 可以跟踪标准编程语言机制（例如 malloc()、calloc()、free()、operator new() 和 operator delete()）分配的堆内存。

如果应用程序使用在标准函数顶层工作的定制内存管理系统（例如，使用 malloc() 实现的池分配管理），则 Discover 仍会运行，但不能保证正确报告泄漏或对已释放内存的访问。

Discover 不支持下列内存分配器：

- 直接使用 brk(2)() 或 sbrk(2)() 系统调用的定制堆分配器
- 静态链接到二进制文件中的标准堆管理函数
- 使用 mmap(2)() 和 shmget(2)() 系统调用从用户代码中分配的内存

不支持 sigaltstack(2)() 函数。

无法检测到静态和自动数组的超出边界错误

由于检测数组边界所用的算法问题，Discover 无法检测到静态和自动（本地）数组的超出边界访问错误。只能检测动态分配数组的错误。

◆ ◆ ◆ 第 3 章

3 代码覆盖工具 (Uncover)

Uncover 的使用要求

Uncover 使用通过 Sun Studio 12 Update 1、Oracle Solaris Studio 12.2 或 Oracle Solaris Studio 12.3 编译器编译的二进制文件，或通过 GCC for Sun Systems 4.2.0 版（或更高版本）编辑器编译的二进制文件。Uncover 适用于运行 Solaris 10 10/08 操作系统（或更高的 Solaris 10 Update 版本）或 Oracle Solaris 11 的基于 SPARC 或基于 x86 的系统。

按照说明进行编译的二进制文件包含一些信息，Uncover 可使用这些信息可靠地反汇编二进制文件，以便对其进行检测以收集覆盖数据，

在编译二进制文件时使用 `-g` 选项生成调试信息，可以让 Uncover 使用源代码级别的覆盖信息。如果二进制文件不是使用 `-g` 选项编译的，Uncover 只能使用基于程序计数器 (program counter, PC) 的覆盖信息。

Uncover 适用于使用 Oracle Solaris Studio 编译器生成的任何二进制文件，但最适合于在不使用优化选项的情况下生成的二进制文件。（以前的 Uncover 发行版至少需要 `-O1` 优化级别。）如果二进制文件是使用优化选项生成的，则使用较低的优化级别（`-O1` 或 `-O2`）时 Uncover 结果将更佳。在生成二进制文件时使用 `-g` 选项会生成调试信息，通过该信息，Uncover 可以将指令与行编号相关联，从而获得源代码行级别覆盖信息优化级别为 `-O3` 和更高级别时，编译器可能会删除可能从不不执行的或冗余的某一代码，这可能导致没有任何二进制文件指令用于某些源代码行。在此类情况下，将不会为这些行报告任何覆盖信息。有关更多信息，请参见第 52 页中的 “使用 Uncover 时的限制”。

使用 Uncover

使用 Uncover 生成覆盖信息的过程分为三个步骤：

1. 检测二进制文件
2. 运行检测过的二进制文件
3. 生成并查看覆盖报告

检测二进制文件

输入的二进制文件可以是可执行文件或共享库。必须分别检测每个要分析的二进制文件。

使用 `uncover` 命令检测二进制文件。例如，以下命令将检测二进制文件 `a.out`，并使用检测过的 `a.out` 来覆盖输入 `a.out`。该命令还将创建一个后缀为 `.uc` 的目录（本例中为 `a.out.uc`），将在该目录中收集覆盖数据。输入二进制文件的副本保存在此目录中。

```
uncover a.out
```

检测二进制文件时，可以使用以下选项：

`-c`	打开指令、块和函数的执行计数报告。缺省情况下，仅报告已覆盖或未覆盖的代码的信息。（检测二进制文件和生成覆盖报告时，均指定该选项。）
`-d` *directory*	通知 Uncover 在 *directory* 中创建覆盖数据目录。当您为多个二进制文件收集覆盖数据时，此选项十分有用，使您可以在同一个目录中创建所有的覆盖数据目录。此外，如果从不同的位置运行同一个检测过的二进制文件的不同实例，使用此选项可确保在同一个覆盖数据目录中累积所有这些运行中的覆盖数据。 如果不使用 `-d` 选项，将在当前运行目录中创建覆盖数据目录。
`-m on | off`	打开和关闭线程安全分析。缺省情况下为打开。将该选项与 `-c` 运行时选项结合使用。如果使用 `-m off` 检测使用线程的二进制文件，则该二进制文件会在运行时失败，并且显示一条消息，要求您使用 `-m on` 重新检测该二进制文件。
`-o` *output_binary_file*	将检测过的二进制文件写入指定的文件。缺省情况下，使用检测过的文件覆盖输入二进制文件。

如果对某个已检测的输入二进制文件运行 `uncover` 命令，Uncover 将发出错误消息，通知您该二进制文件已经检测，无法再次检测，您可以运行该二进制文件来生成覆盖数据。

运行检测过的二进制文件

检测二进制文件后，您可以按正常方式运行它。每次运行检测过的二进制文件时，都会在 Uncover 执行检测期间创建的、后缀为 .uc 的覆盖数据目录中收集代码覆盖数据。由于 Uncover 数据收集是多线程安全的，并且是多进程安全的，因此，对进程中的并发运行或线程的数量没有限制。覆盖数据将累积所有的运行和线程。

生成并查看覆盖报告

要生成覆盖报告，请对覆盖数据目录运行 `uncover` 命令。例如：

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

此命令将根据 `a.out.uc` 目录中的覆盖数据生成一个名为 *binary_name*`.er` 的 Oracle Solaris Studio 性能分析器实验目录，启动性能分析器 GUI，并显示该实验。如果当前目录或起始目录中存在 `.er.rc` 文件（请参见 Oracle Solaris Studio 12.2 性能分析器手册），分析器显示实验的方式可能会受影响。

还可以使用 `uncover` 命令选项生成可以在 Web 浏览器中查看的 HTML 报告，或者生成可以在终端窗口中查看的 ASCII 报告。或者，将数据定向到代码分析器可以分析和显示数据的目录中。

`-a`	将错误数据写入 *binary_name*`.analyze/coverage` 目录以供代码分析器使用。
`-c`	打开指令、块和函数的执行计数报告。缺省情况下，仅报告已覆盖或未覆盖的代码的信息。（检测二进制文件和生成覆盖报告时，均指定该选项。）
`-e on | off`	生成覆盖报告的实验目录，并在性能分析器 GUI 中显示实验。缺省情况下为打开。
`-H` *html_directory*	在指定的目录中以 HTML 格式保存覆盖数据，并在 Web 浏览器中自动显示这些数据。缺省情况下为关闭。
`-h` 或 `-?`	帮助。
`-n`	生成覆盖报告，但不启动性能分析器或 Web 浏览器等查看器。
`-t` *ascii_file*	在指定的文件中生成 ASCII 覆盖报告。缺省情况下为关闭。
`-V`	输出 Uncover 版本信息并退出。
`-v`	详细。输出 Uncover 正在执行的操作的日志。

仅启用了一种输出格式，因此，如果指定了多个输出选项，Discover 仅使用命令中的最后一个选项。

示例

uncover a.out

此命令将检测二进制文件 a.out，覆盖输入 a.out，在当前目录中创建 a.out.uc 覆盖数据目录，并在 a.out.uc 目录中保存输入 a.out 的副本。如果已检测 a.out，将显示警告消息，并且不执行检测。

uncover -d coverage a.out

此命令将执行第一个示例中执行的所有操作，不过，它会在目录 coverage 中创建 a.out.uc 覆盖目录。

uncover a.out.uc

此命令使用 a.out.uc 覆盖目录中的数据在工作目录中创建代码覆盖实验 (a.out.er)，并启动性能分析器 GUI 以显示该实验。

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

此命令使用 a.out.uc 覆盖目录中的数据在 a.out.html 目录中创建 HTML 代码覆盖报告，并在 Web 浏览器中显示该报告。

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

此命令使用 a.out.uc 覆盖目录中的数据在 a.out.txt 文件中创建 ASCII 代码覆盖报告。

uncover -a a.out.uc

此命令使用 a.out.c 覆盖目录中的数据在 *binary_name*.analyze/coverage 目录中创建覆盖报告，以供代码分析器使用。

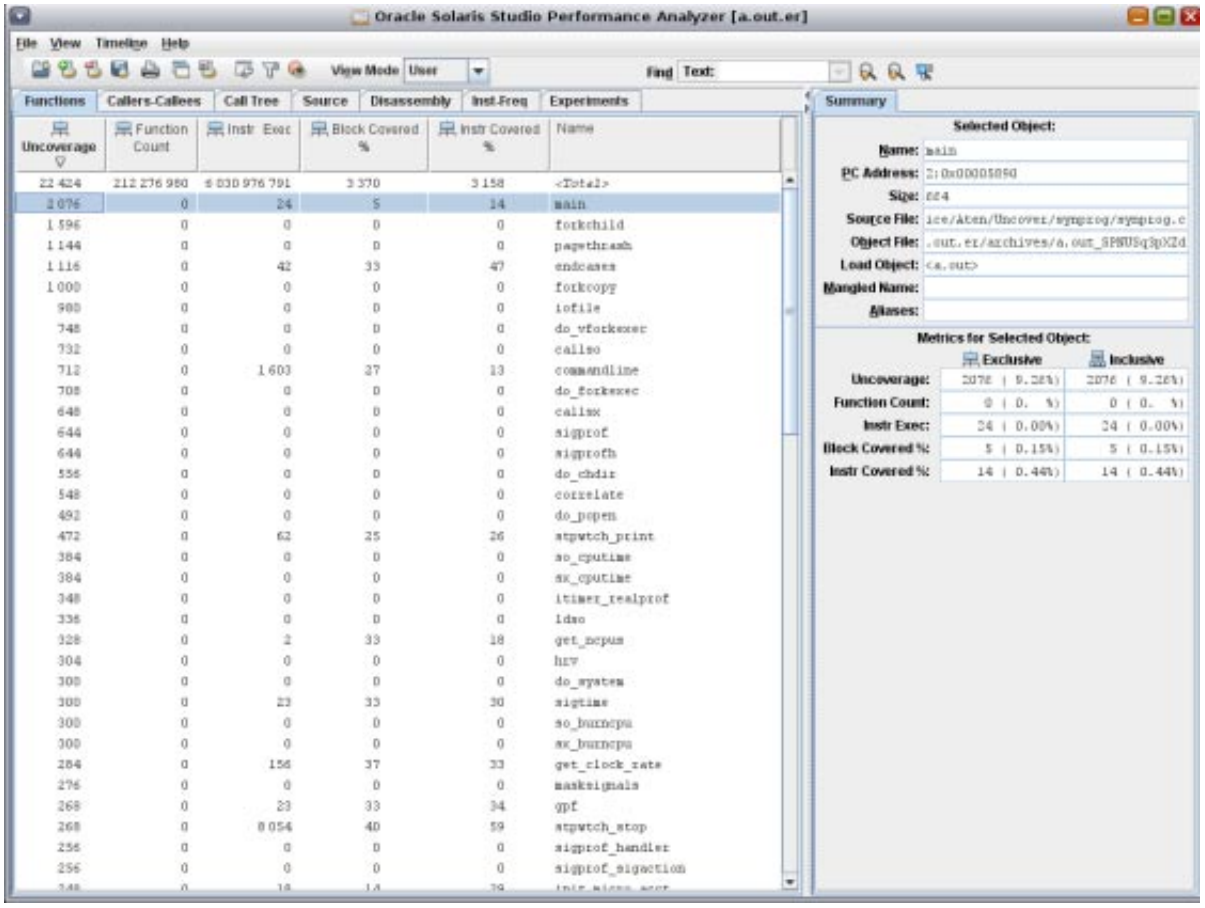
了解性能分析器中的覆盖报告

缺省情况下，在对覆盖目录运行 uncover 命令时，会在 Oracle Solaris Studio 性能分析器中以实验的形式打开覆盖报告。分析器使用 "Functions"（函数）、"Source"（源）、"Disassembly"（反汇编）和 "Inst-Freq"（指令频率）选项卡来显示覆盖数据。

"Functions"（函数）选项卡

在分析器中打开覆盖报告时，"Functions"（函数）选项卡处于选中状态。该选项卡显示了一些列，其中列出了每个函数的 "Uncoverage"（未覆盖）、"Function Count"（函数计

数）、"Instr Exec"（指令执行）、"Block Covered %"（块覆盖率）和 "Instr Covered % "（指令覆盖率）计数器。单击任何一列的标题可以将该列设置为数据的排序键。单击列标题上的箭头可以反转排序顺序。



"Uncoverage"（未覆盖）计数器

"Uncoverage"（未覆盖）度量是一项极其强大的 Uncover 功能。如果使用此列作为排序键，降序排列时，显示在最上面的函数是最有可能提高覆盖率的函数。在本例中，`main()` 函数位于列表的顶端，因为它在 "Uncoverage"（未覆盖）列中的数目最多。（`sigprof()` 和 `sigprofh()` 函数的数目相同，因此它们按字母顺序排列。）

`main()` 函数的未覆盖数目是指，向导致调用该函数的套件添加一个测试时可能覆盖的代码的字节数。根据函数的结构，覆盖率实际增加的量会有所不同。如果该函数没有

分支，并且它调用的所有函数也是直线型函数，则覆盖确实只会增加指定的字节数。但一般而言，覆盖增长会小于潜在值，也许会小很多。

"Uncoverage"（未覆盖）列中使用非零值的未覆盖函数称为根未覆盖函数，表示它们都由覆盖函数调用。仅由非根未覆盖函数调用的函数没有自己的未覆盖数目。可以推定，在后续运行中，随着测试套件的改进而覆盖了潜力较大的未覆盖函数，这些函数将成为覆盖或未覆盖函数。

覆盖数目是非独占性的。

"Function Count"（函数计数）计数器

"Function Count"（函数计数）报告已覆盖的函数和未覆盖的函数。关键在于计数是零还是非零。如果计数为零，表示该函数未覆盖。如果计数非零，表示该函数已覆盖。如果执行了函数中的任一指令，该函数将视为已覆盖。

可以检测到此列中的非顶层未覆盖函数。如果某个函数的函数计数是零，并且未覆盖数目也是零，表示该函数不是顶层覆盖函数。

"Instr Exec"（指令执行）计数器

"Instr Exec"（指令执行）计数器显示已覆盖的指令和未覆盖的指令。零计数表示未执行指令，非零计数表示已执行指令。

在 "Functions"（函数）选项卡中，该计数器显示为每个函数执行的指令总数。此计数器还出现在 "Source"（源）选项卡（请参见第 43 页中的 “"Source"（源）选项卡”）和 "Disassembly"（反汇编）选项卡（请参见第 44 页中的 “"Disassembly"（反汇编）选项卡”）中。

"Block Covered %"（块覆盖率）计数器

对于每个函数，"Block Covered %"（块覆盖率）计数器显示该函数中被覆盖的基本块的百分比。通过此数目可以大致了解函数的覆盖情况。请忽略 `<Total>` 行中的这个数字，它是列中的百分比之和，没有意义。

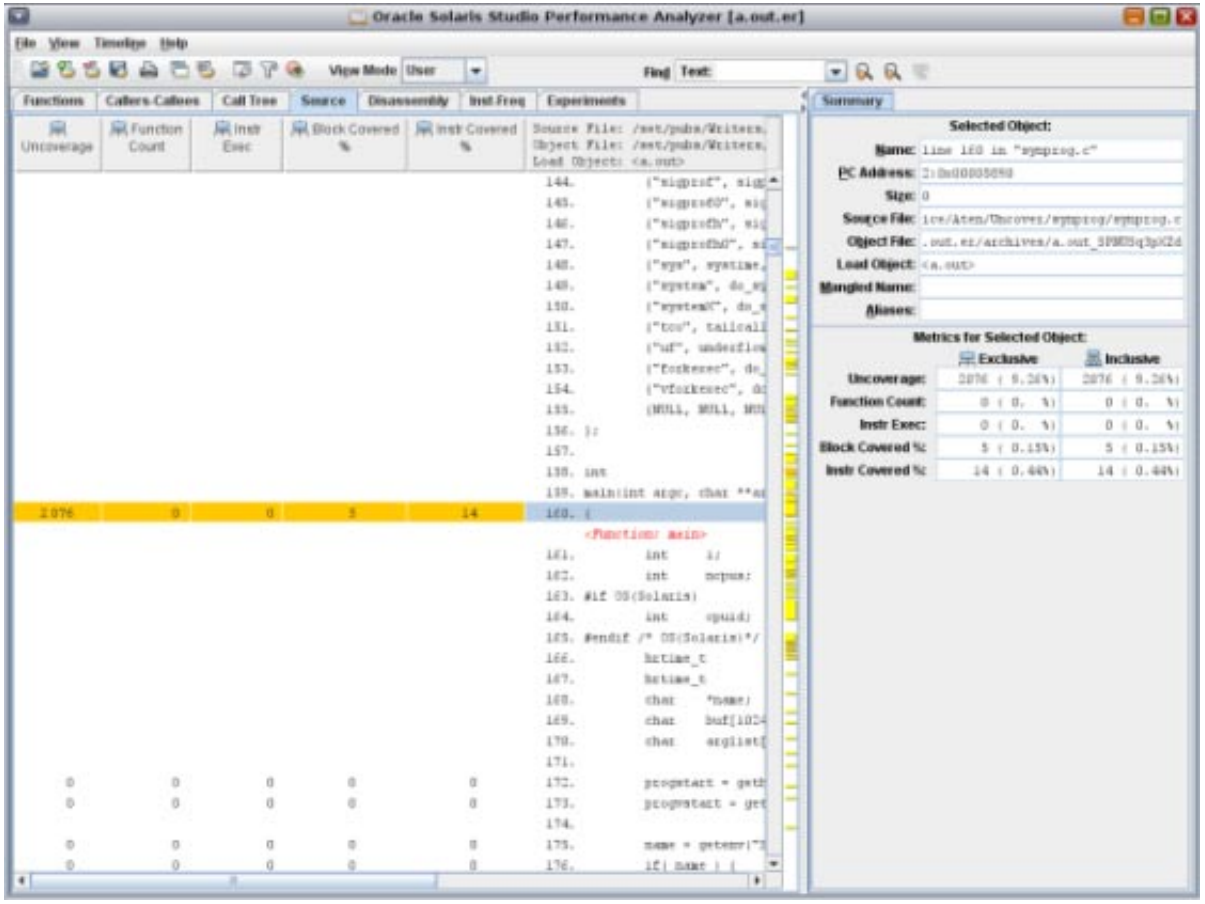
"Instr Covered %"（指令覆盖率）计数器

对于每个函数，"Instr Covered %"（指令覆盖率）计数器显示该函数中被覆盖的指令的百分比。通过此数目也可以大致了解函数覆盖的情况。请忽略 `<Total>` 行中的这个数字，它是列中百分比之和，没有意义。

"Source"（源）选项卡

如果二进制文件是使用 -g 选项编译的，"Source"（源）选项卡将显示程序的源代码。由于 Uncover 在二进制文件级别上检测您的程序，并且已使用优化设置编译程序，因此，此选项卡中的覆盖信息很难解释。

"Source"（源）选项卡中的 "Instr Exec"（指令执行）计数器显示了为每个源代码行执行的指令数，它本质上是语句级别的代码覆盖信息。非零值表示该语句已覆盖；零值表示该语句未覆盖。变量声明和注释没有指令执行计数。



一些源代码行可能不具有与其相关联的任何覆盖信息。在这些情况下，行为空白，并且所有字段中都没有数字。出现这些行的原因是：

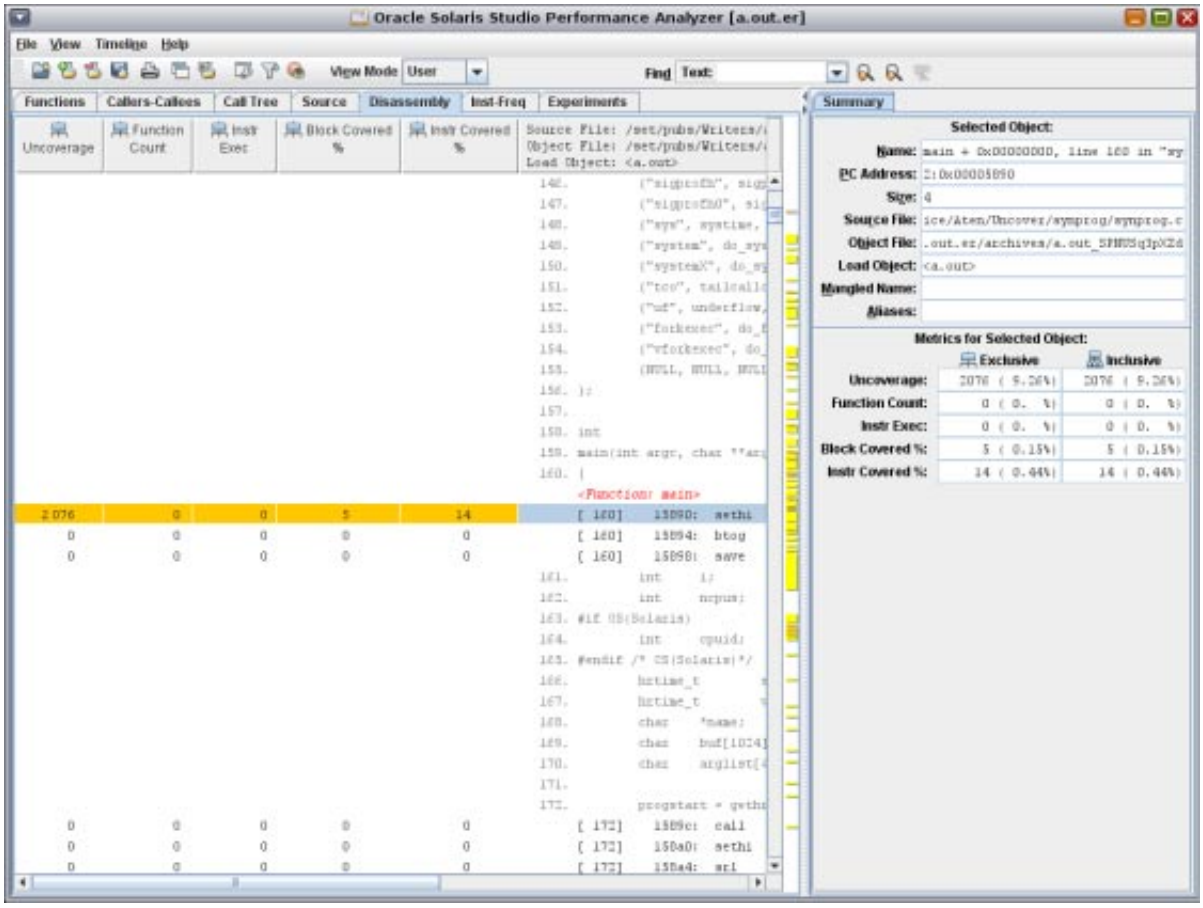
- 注释、空白行、声明和其他语言结构不包含可执行的代码。
- 编译器优化已删除与行对应的代码，原因是：
  - 代码从不执行（死代码）。
  - 代码可以执行，但为冗余代码。

有关更多信息，请参见第 52 页中的“使用 Uncover 时的限制”。

"Disassembly"（反汇编）选项卡

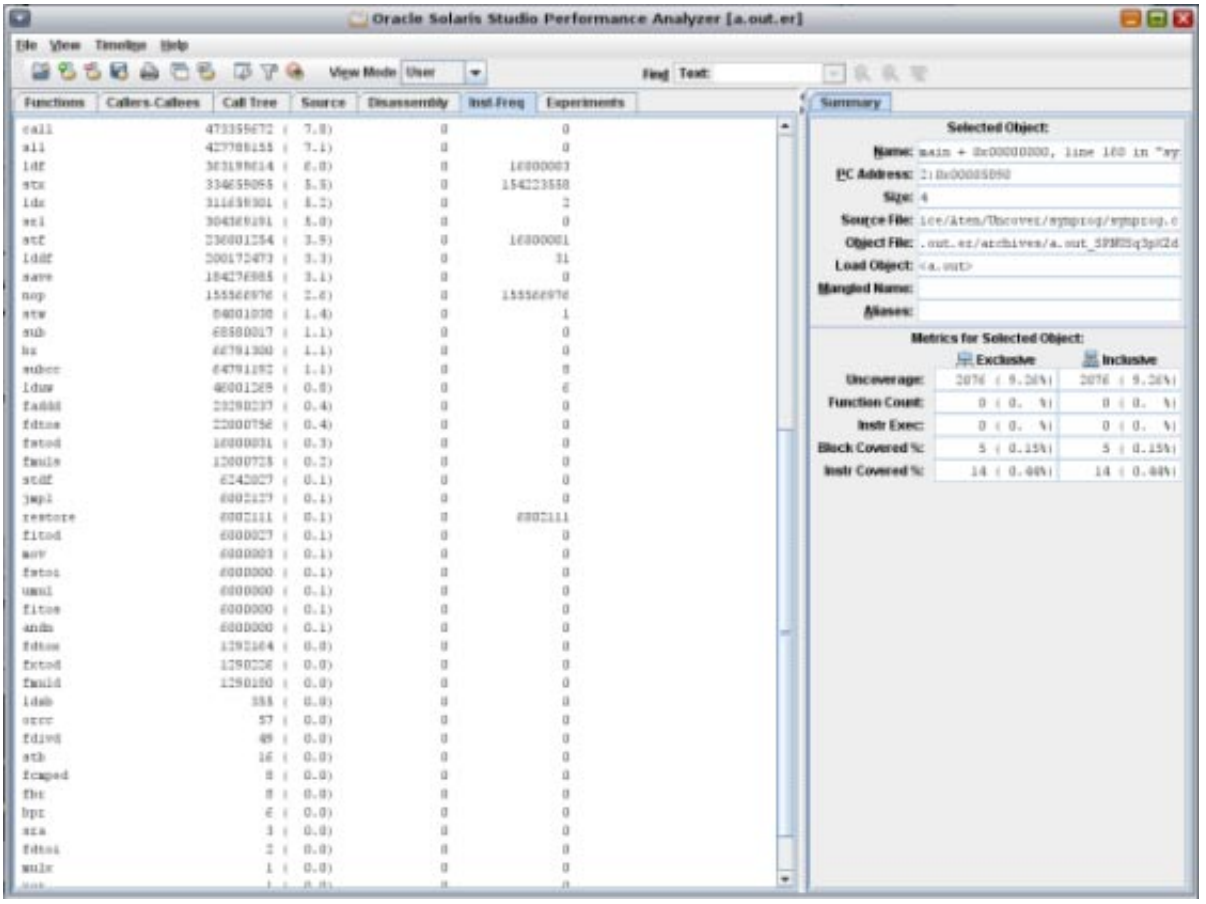
如果在 "Source"（源）选项卡中选择一行，然后选择 "Disassembly"（反汇编）选项卡，分析器将在二进制文件中查找选定的行，并显示其反汇编。

此选项卡中的 "Instr Exec"（指令执行）计数器显示了每个指令的执行次数。



"Inst-Freq"（指令频率）选项卡

"Inst-Freq"（指令频率）选项卡显示总的覆盖摘要。



了解 ASCII 覆盖报告

如果从覆盖数据目录中生成覆盖报告时指定了 `-t` 选项，Uncover 会将覆盖报告写入指定的 ASCII 文件（文本文件）。

```
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%
```

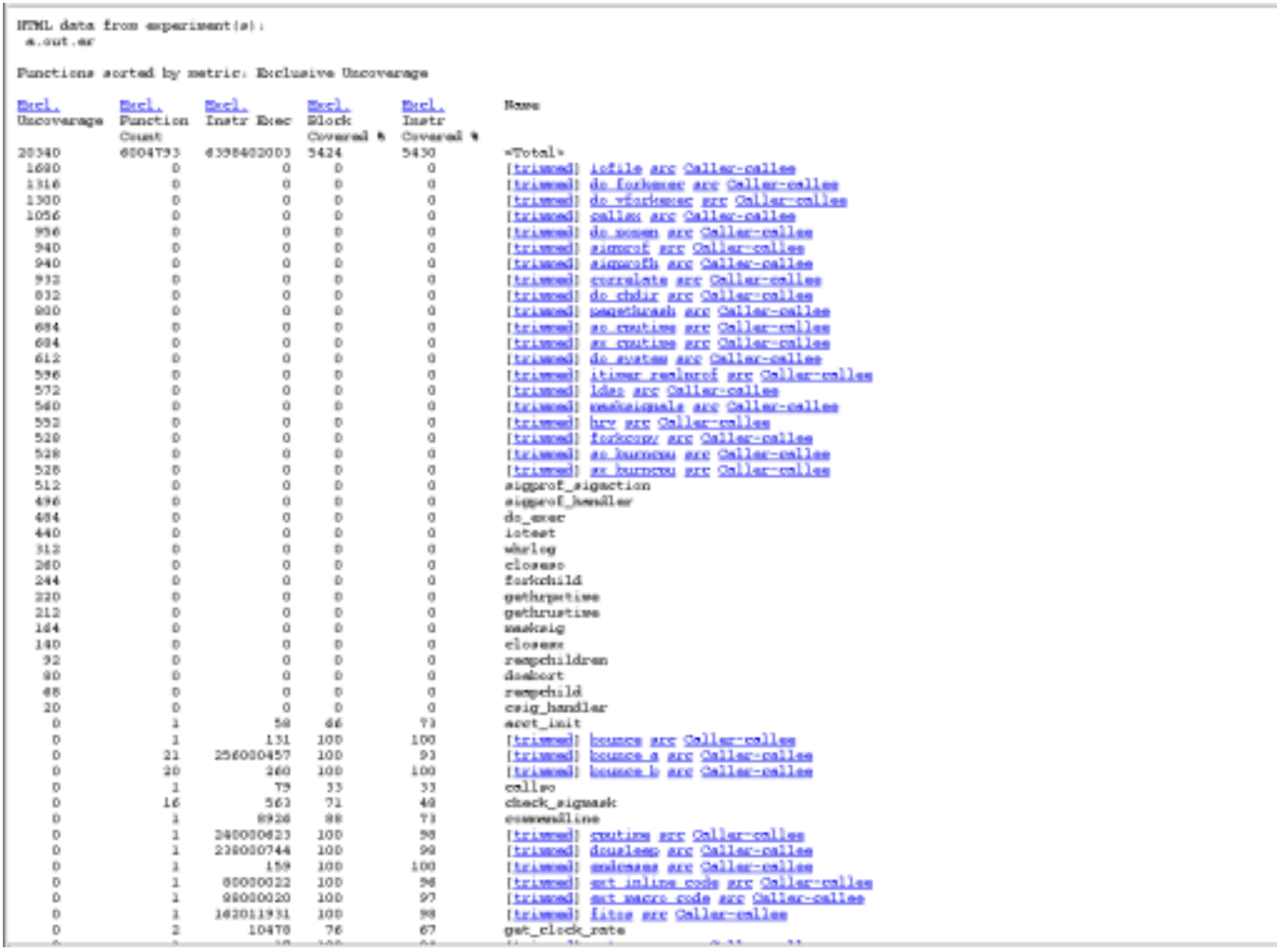
```
Total Instruction Executions: 4,760,934,518
Average Executions per Instruction: 767,768.83
Number of times this program was executed: unavailable
Functions sorted by metric: Exclusive Uncoverage

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
=============================================================
```

了解 HTML 覆盖报告

HTML 报告类似于性能分析器中显示的报告。

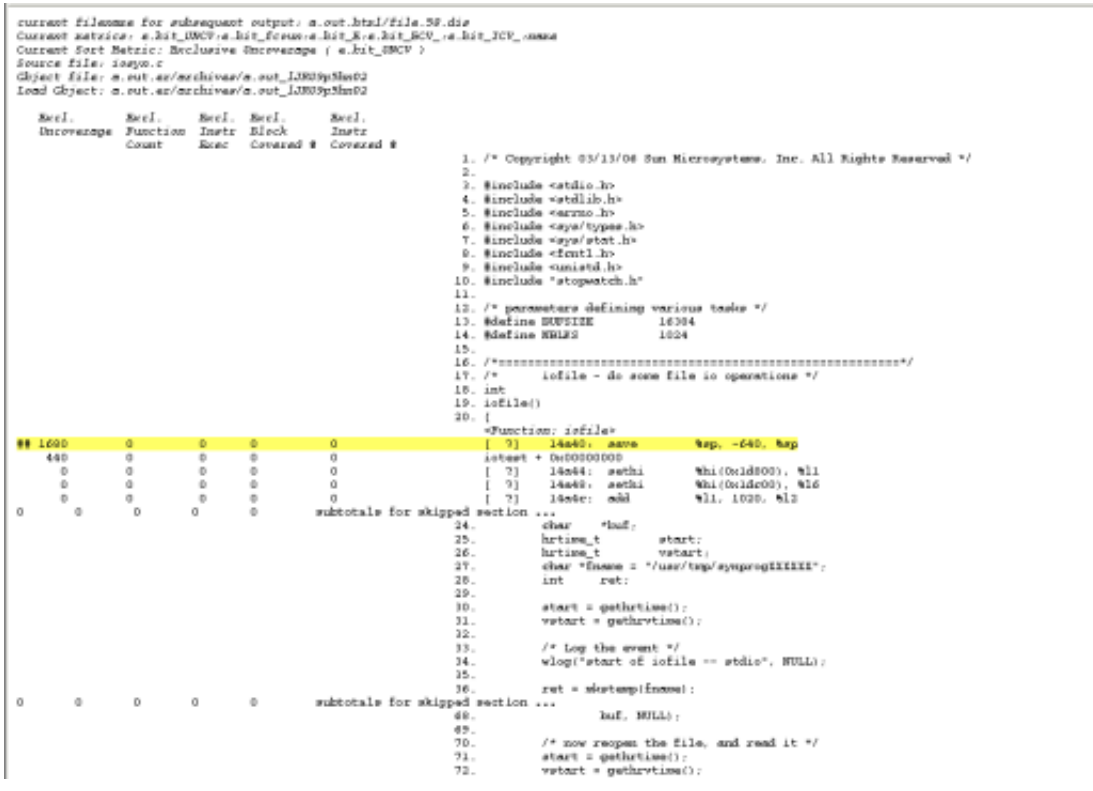


HTML data from experiment(s):
a.out.er

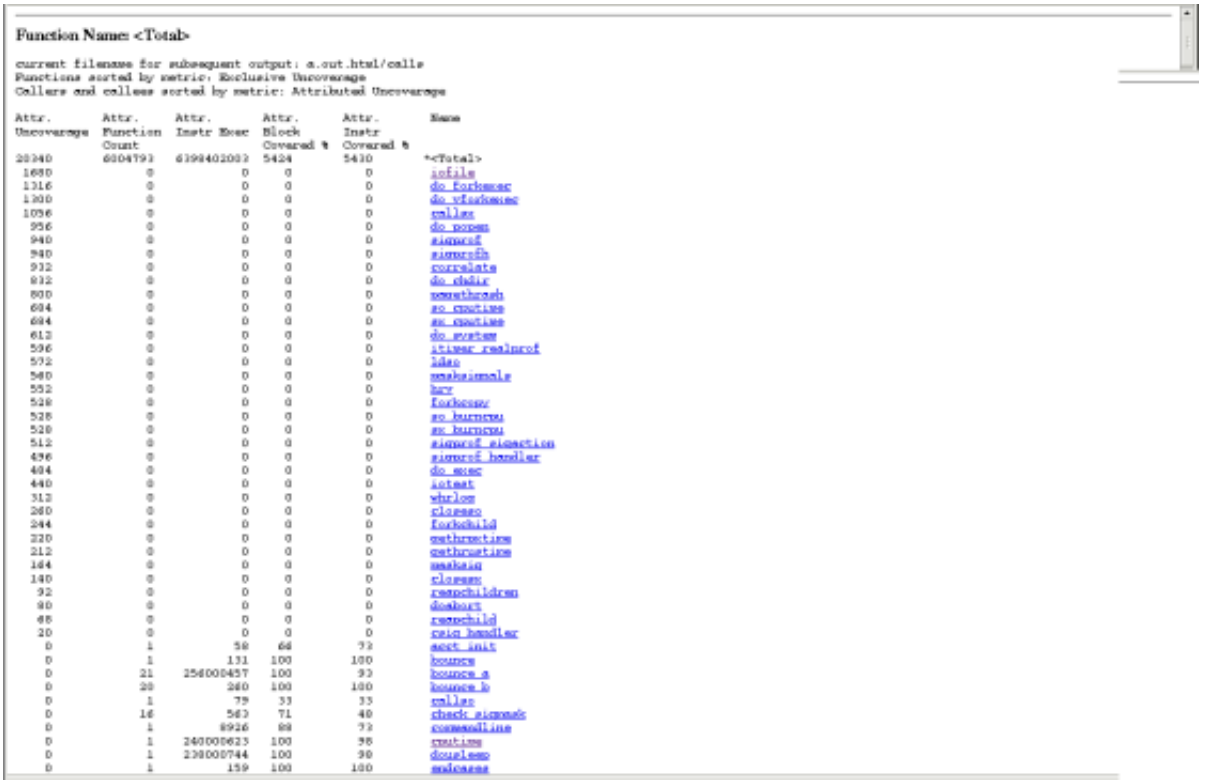
Functions sorted by metric: Exclusive Uncoverage

Excl. Uncoverage	Excl. Function Count	Excl. Instr Exec	Excl. Block Covered %	Excl. Instr Covered %	Name
20340	6004793	6398402003	5424	5430	<Total>
1680	0	0	0	0	[trimmed] isfile src Caller-callee
1316	0	0	0	0	[trimmed] do_forkexec src Caller-callee
1300	0	0	0	0	[trimmed] do_vforkexec src Caller-callee
1056	0	0	0	0	[trimmed] callso src Caller-callee
956	0	0	0	0	[trimmed] do_sosem src Caller-callee
940	0	0	0	0	[trimmed] sigprof src Caller-callee
940	0	0	0	0	[trimmed] sigprofh src Caller-callee
932	0	0	0	0	[trimmed] correlate src Caller-callee
832	0	0	0	0	[trimmed] do_chdir src Caller-callee
800	0	0	0	0	[trimmed] segvhandh src Caller-callee
684	0	0	0	0	[trimmed] so_routine src Caller-callee
684	0	0	0	0	[trimmed] sx_routine src Caller-callee
612	0	0	0	0	[trimmed] do_system src Caller-callee
596	0	0	0	0	[trimmed] itimer_realprof src Caller-callee
572	0	0	0	0	[trimmed] ldso src Caller-callee
560	0	0	0	0	[trimmed] reshsignals src Caller-callee
552	0	0	0	0	[trimmed] hrv src Caller-callee
520	0	0	0	0	[trimmed] forkcopy src Caller-callee
528	0	0	0	0	[trimmed] so_burnsyu src Caller-callee
528	0	0	0	0	[trimmed] sx_burnsyu src Caller-callee
512	0	0	0	0	sigprof_sigaction
496	0	0	0	0	sigprof_handler
484	0	0	0	0	do_exec
440	0	0	0	0	isteet
312	0	0	0	0	whrlog
260	0	0	0	0	closeso
244	0	0	0	0	forkchild
220	0	0	0	0	gethrgetime
212	0	0	0	0	gethrustime
184	0	0	0	0	maskcig
140	0	0	0	0	closesx
92	0	0	0	0	reapchildren
80	0	0	0	0	doshort
68	0	0	0	0	reapchild
20	0	0	0	0	csig_handler
0	1	58	66	73	sret_init
0	1	131	100	100	[trimmed] bounce src Caller-callee
0	21	256000457	100	93	[trimmed] bounce_a src Caller-callee
0	20	260	100	100	[trimmed] bounce_b src Caller-callee
0	1	79	33	33	callso
0	16	563	71	49	check_sigmask
0	1	8926	88	73	commandline
0	1	240000823	100	98	[trimmed] cputime src Caller-callee
0	1	238000744	100	98	[trimmed] dousleep src Caller-callee
0	1	159	100	100	[trimmed] endsesss src Caller-callee
0	1	80000022	100	98	[trimmed] set_inline_code src Caller-callee
0	1	98000020	100	97	[trimmed] set_macro_code src Caller-callee
0	1	162011931	100	98	[trimmed] fitos src Caller-callee
0	2	10478	76	67	get_clock_rate

如果单击某个函数的函数名称链接或 `trimmed` 链接，将显示该函数的反汇编数据。



如果单击某个函数的 `Caller-callee` 链接，将显示调用方-被调用方数据。



Function Name: <Total>

current filename for subsequent output: a.out.html/calls
Functions sorted by metric: Exclusive Uncoverage
Callers and callees sorted by metric: Attributed Uncoverage

Attr. Uncoverage	Attr. Function Count	Attr. Instr Exec	Attr. Block Covered %	Attr. Instr Covered %	Name
28240	6004793	6398402082	5424	5430	*<Total>
1680	0	0	0	0	zofile
1316	0	0	0	0	do_forkmesc
1300	0	0	0	0	do_vforkmesc
1056	0	0	0	0	exiter
956	0	0	0	0	do_execs
940	0	0	0	0	sigmarf
940	0	0	0	0	sigmarfh
932	0	0	0	0	correlate
832	0	0	0	0	do_stdio
800	0	0	0	0	memthresh
684	0	0	0	0	zc_creates
684	0	0	0	0	zc_creates
612	0	0	0	0	do_system
596	0	0	0	0	itimer_realprof
572	0	0	0	0	ldso
560	0	0	0	0	enablesmals
552	0	0	0	0	hcv
528	0	0	0	0	forkcopy
528	0	0	0	0	zc_burncpu
520	0	0	0	0	zc_burncpu
512	0	0	0	0	sigmarf_sigaction
496	0	0	0	0	sigmarf_handler
484	0	0	0	0	do_exec
440	0	0	0	0	istat
312	0	0	0	0	whrlow
260	0	0	0	0	closeo
244	0	0	0	0	forkchild
220	0	0	0	0	gethrvtime
212	0	0	0	0	gethrvtime
184	0	0	0	0	makexp
140	0	0	0	0	closeo
92	0	0	0	0	reapchildren
90	0	0	0	0	doabort
68	0	0	0	0	reapchild
20	0	0	0	0	cpic_handler
0	1	58	66	72	acct_init
0	1	131	100	100	bounce
0	21	254000457	100	93	bounce_a
0	20	260	100	100	bounce_b
0	1	79	33	33	exiter
0	16	563	71	40	check_signals
0	1	8926	88	72	commandline
0	1	240000623	100	95	creatino
0	1	238000744	100	90	doubleso
0	1	159	100	100	endoexe

使用 Uncover 时的限制

只能检测有注释的代码

Uncover 只能检测按照第 37 页中的 “Uncover 的使用要求”中的说明准备的代码。无注释代码可能来自链接到二进制文件中的汇编语言代码，或者来自使用早于该部分中所列版本的编译器或操作系统编译的模块。

在准备时，特别要排除包含 `asm` 语句或 `.il` 模板的汇编语言模块和函数。

计算机指令可能不同于源代码

Uncover 处理计算机代码。它会查找计算机指令的覆盖，然后将此覆盖与源代码相关联。某些源代码语句没有关联的计算机指令，因此，看上去好像是 Uncover 没有报告这些语句的覆盖。

示例 1

考虑以下代码片段：

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

根据您的预期，Uncover 应该对 `if` 语句报告非零执行计数，但编译器很可能会删除此代码，使得 Uncover 在检测期间看不到它。因此，不会针对这些指令报告覆盖数据。

示例 2

以下为死代码的示例：

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

对应的汇编显示删除了 B,C,D 的调用，因为该代码从不执行。

```
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
```

因此，不会针对第 5 行到第 6 行报告覆盖。

```
   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.    }
```

示例 3

以下为冗余代码的示例：

```
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  }
```

在低优化级别时，编译器可能为所有行生成代码：

```
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
```

在高优化级别时，大多数可执行的源代码行不具有任何对应的指令：

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

因此，不会针对某些行报告覆盖。

```
     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

二

共

检

文

无

性

要