Digital UNIX

Program Analysis Using Atom Tools

March 1996

Product Version: Digital UNIX Version 4.0 or higher

This document describes how to develop program-analysis tools using the code instrumentation interface provided by Atom in the programming development environment for the Digital UNIX operating system.

It also provides details on Third Degree, one of the prepackaged program-analysis tools provided with Atom.

(The two chapters in this document were copied from the March 1996 edition of the *Programmer's Guide*.)

Digital Equipment Corporation
Maynard, Massachusetts

Contents

About This Document

1 Using and Developing Atom Tools

Tables

About This Document

This document describes how to use and create Atom-based program analysis tools that run in the programming environment supported by the Digital UNIX® operating system.

Note

The contents of this document were copied from the March 1996 edition of the *Programmer's Guide*.

Audience

This document addresses programmers who need to use or develop Atom-based program analysis tools.

Organization

This document contains two chapters.

Chapter 1	Describes how to design and create custom Atom tools. It also describes how to use prepackaged Atom tools to instrument an application program for various purposes, such as to obtain profiling data or to perform cache-use analysis.
Chapter 2	Describes how to use Atom's Third Degree tool to perform memory access checks and leak detection on an application program.

Related Documents

The following manuals contain information pertaining to program development:

Programming: General

Programmer's Guide

Calling Standard for Alpha Systems

Assembly Language Programmer's Guide

Programming Support Tools

Network Programmer's Guide

Digital Portable Mathematics Library

Writing Software for the International Market

Kernel Debugging

Ladebug Debugger Manual

Programming: Compatibility, Migration, and Standards

ULTRIX to Digital UNIX Migration Guide

VAX System V to Digital UNIX Migration Guide

System V Compatibility User's Guide

POSIX Conformance Document

XPG3 Questionnaire

Programming: Realtime

Guide to Realtime Programming

Programming: Streams

Programmer's Guide: STREAMS

Programming: Multithreaded Applications

Guide to DECthreads

General User Information

Release Notes

Documentation Overview

Conventions

`%` `$`	A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bourne and Korn shells.
`#`	A number sign represents the superuser prompt.
`%` **`cat`**	Boldface type in interactive examples indicates typed user input.
`file`	Italic (slanted) type indicates variable values, placeholders, and function argument names.
`[ | ]` **`{ | }`**	In syntax definitions, brackets indicate items that are optional and braces indicate items that are required. Vertical bars separating items inside brackets or braces indicate that you choose one item from among those listed.

`. . .`	In syntax definitions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.
`cat`(1)	A cross-reference to a reference page includes the appropriate section number in parentheses. For example, `cat`(1) indicates that you can find information on the `cat` command in Section 1 of the reference pages.

Using and Developing Atom Tools 1

Program analysis tools are extremely important for computer architects and software engineers. Computer architects use them to test and measure new architectural designs, and software engineers use them to identify critical pieces of code in programs or to examine how well a branch prediction or instruction scheduling algorithm is performing. Program analysis tools are needed for problems ranging from basic block counting to instruction and data cache simulation. Although the tools that accomplish these tasks may appear quite different, each can be implemented simply and efficiently through code instrumentation.

Atom provides a flexible code instrumentation interface that is capable of building a wide variety of tools. Atom separates the common part in all problems from the problem-specific part by providing machinery for instrumentation and object-code manipulation, and allowing the tool designer to specify what points in the program are to be instrumented. Atom is independent of any compiler and language as it operates on object modules that make up the complete program.

Atom, as provided in the Digital UNIX operating system, provides the following:

- A set of prepackaged tools that may be used to instrument applications for profiling or debugging purposes. Use the following command to apply one of these tools to a given application:

  atom *application_program* **–tool** *toolname* **–env** *environment*

- A command interface and a collection of instrumentation routines that may be used to create custom Atom tools. Use the following command to create a custom-designed Atom tool:

  atom *application_program instrumentation_file analysis_file*

The `atom`(1) reference page describes both forms of the `atom` command.

This chapter contains the following sections:

- Section 1.1 describes the prepackaged Atom tools and how to use them.
- Section 1.2 discusses how you can develop specialized Atom tools.

1.1 Using Prepackaged Atom Tools

The Digital UNIX operating system provides and supports the Atom tools listed in Table 1-1.

Table 1-1: Supported Prepackaged Atom Tools

Tool	Description
Third Degree (`third`)	Performs memory access checks and detects memory leaks in an application. The Third Degree Atom tool is described in Chapter 2 and in the `third`(5) reference page.
`hiprof`	Produces a flat profile of an application that shows the execution time spent in a given procedure and a hierarchical profile that shows the execution time spent in a given procedure and all its descendants. The `hiprof` Atom tool is described in the *Programmer's Guide* and `hiprof`(5).
`pixie`	Partitions an application into basic blocks and counts the number of times each basic block is executed. The `pixie` Atom tool is described in the *Programmer's Guide* and `pixie`(5).

The Digital UNIX operating system provides the unsupported Atom tools listed in Table 1-2 as examples for programmers developing custom-designed Atom tools. These tools are distributed in source form to illustrate Atom's programming interfaces. Some of the tools are further described in Section 1.2.

Table 1-2: Example Prepackaged Atom Tools

Tool	Description
`branch`	Instruments all conditional branches to determine how many are predicted correctly.
`cache`	Determines cache miss rate if application runs in 8K direct-mapped cache.
`dtb`	Determines the number of dtb (data translation buffer) misses if the application uses 8KB pages and a fully associative translation buffer.
`dyninst`	Provides fundamental dynamic counts of instructions, loads, stores, blocks, and procedures.

Table 1-2: (continued)

Tool	Description
`inline`	Identifies potential candidates for inlining.
`iprof`	Prints the number of times each procedure is called as well as the number of instructions executed (dynamic count) by each procedure.
`malloc`	Records each call to the `malloc` function and prints a summary of the application's allocated memory.
`prof`	Prints the number of instructions executed (dynamic count) by each procedure.
`ptrace`	Prints the name of each procedure as it is called.
`trace`	Generates an address trace, logs the effective address of every load and store operation, and logs the address of the start of every basic block as it is executed.

1.2 Developing Atom Tools

An Atom tool consists of the following:

- An instrumentation file – Modifies the application to which it is applied by adding calls at well-defined locations to tool-specific analysis procedures.
- An analysis file – Defines the procedures and data structures required to implement the tool's functionality.

Atom views an application as a hierarchy of components:

1. The program, including the executable and all shared libraries.
2. A collection of objects. An object can be either the main executable or any shared library. An object has its own set of attributes (such as its name) and consists of a collection of procedures.
3. A collection of procedures, each of which consists of a collection of basic blocks.
4. A collection of basic blocks, each of which consists of a collection of instructions.
5. A collection of instructions.

Atom tools insert instrumentation points in an application program at procedure, basic block, or instruction boundaries. For example, basic block counting tools instrument the beginning of each basic block, data cache simulators instrument each load and store instruction, and branch prediction analyzers instrument each conditional branch instruction.

At any instrumentation point, Atom allows a tool to insert a procedure call to an analysis routine. The tool can specify that the procedure call be made before or after an object, procedure, basic block, or instruction.

1.2.1 The ATOM Command Line

The command line used to apply Atom tools to an application is described completely in the `atom`(1) reference page. This section describes the command line and its most commonly used arguments and flags.

The `atom` command line has two forms:

atom *application_program* **–tool** *toolname* [**–env** *environment*] [*flags...*]

This form of the `atom` command is used to build an instrumented version of an application program using a prepackaged Atom tool.

This form requires the `–tool` flag and accepts the `–env` flag. It does not allow either the *`instrumentation_file`* or the *`analysis_file`* parameter.

The `–tool` flag identifies the prepackaged Atom tool to be used. By default, Atom searches for prepackaged tools in the `/usr/lib/cmplrs/atom/tools` and `/usr/lib/cmplrs/atom/examples` directories. You can add directories to the search path by supplying a colon-separated list of additional directories to the `ATOMTOOLPATH` environment variable.

The `–env` flag identifies any special environment (for instance, *`threads`*) in which the tool is to operate. The set of environments supported by a given tool is defined by the tool's creator and listed in the tool's documentation. Atom displays an error if you specify an environment that is undefined for the tool. The prepackaged tools allow you to omit the `–env` flag to obtain a general-purpose environment.

atom application_program *instrumentation_file* [*analysis_file*] [flags...]

This form of the `atom` command is used to apply a tool that instruments an application program. This form requires the *`instrumentation_file`* parameter and accepts the *`analysis_file`* parameter.

The *`instrumentation_file`* parameter specifies the name of a C source file or an object module that contains the Atom tool's instrumentation procedures. By convention, most instrumentation files have the suffix `.inst.c` or `.inst.o`.

The *analysis_file* parameter specifies the name of a C source file or an object module that contains the Atom tool's analysis procedures. Note that you do not need to specify an analysis file if the instrumentation file does not call analysis procedures. By convention, most analysis files have the suffix `.anal.c` or `.anal.o`.

You can have multiple instrumentation and analysis source files. The following example creates composite instrumentation and analysis objects from several source files:

```
% cc -c file1.c file2.c
% cc -c file7.c file8
% ld -r -o tool.inst.o file1.o file2.o
% ld -r -o tool.anal.o file7.o file8.o
% atom hello tool.inst.o tool.anal.o -o hello.tool
```

Note

You can also write analysis procedures in C++. You must assign a type of ‘‘`extern "C"`’’ to each procedure to allow it to be called from the application. You must also compile and link the analysis files before issuing the `atom` command. For example:

```
% cxx -c tool.a.C
% ld -r -o tool.anal.o tool.a.o -lcxx -lexc
% atom hello tool.inst.c tool.anal.o -o hello.tool
```

With the exception of the `–tool` and `–env` flags, both forms of the `atom` command accept any of the remaining flags described in the `atom`(1) reference page. The following are some flags that deserve special mentioning:

`–A1`
: Causes Atom to optimize calls to analysis routines by reducing the number of registers that need to be saved and restored. For some tools, specifying this flag increases the performance of the instrumented application by a factor of 2 (at the expense of some increase in application size). The default behavior is for Atom not to apply these optimizations.

`–debug`
: Allows you to debug instrumentation routines by causing Atom to transfer control to the symbolic debugger at the start of the instrumentation routine. In the following example, the `ptrace` sample tool is run under the `dbx` debugger. The instrumentation is stopped at

line 12, and the procedure name is printed.

```
% atom hello ptrace.inst.c ptrace.anal.c -o hello.ptrace -debug
dbx version 3.11.8
Type 'help' for help.
Stopped in InstrumentAll
(dbx) stop at 12
[4] stop at "/udir/test/scribe/atom.user/tools/ptrace.inst.c":12
(dbx) c
[3] [InstrumentAll:12 ,0x12004dea8] if (name == NULL) name = "UNKNOWN";
(dbx) p name
0x2a391 = "__start"
```

`–g`

Causes Atom to build the analysis procedures with debugging symbol table information, allowing you to run instrumented applications under a symbolic debugger. Atom assumes that the application itself runs correctly, allowing debugger commands to be used only on analysis procedures. For example:

```
% atom hello ptrace.inst.c ptrace.anal.c -o hello.ptrace -g
```

```
% dbx hello.ptrace
dbx version 3.11.8
Type 'help' for help.
(dbx) stop in ProcTrace
[2] stop in ProcTrace
(dbx) r
[2] stopped at [ProcTrace:5 ,0x120005574] fprintf (stderr,"%s\n",name);
(dbx) n
__start
     [ProcTrace:6 ,0x120005598] }
```

`–toolargs`

Passes arguments to the Atom tool's instrumentation routine. Atom passes the arguments in the same way that they are passed to C programs, using the *argc* and *argv* arguments to the `main` program. For example:

```
#include <stdio.h>
unsigned InstrumentAll(int argc, char **argv) {
     int i;
     for (i = 0; i < argc; i++) {
       printf(stderr,"argv[%d]: %s\n",argv[i]);
     }
}
```

The following example shows how Atom passes the `–toolargs` arguments:

```
% atom hello args.inst.c -toolargs="8192 4"
argv[0]: hello
argv[1]: 8192
argv[2]: 4
```

1.2.2 Atom Instrumentation Routine

Atom invokes a tool's instrumentation routine on a given application program when that program is specified as the *application_program* parameter to the atom command, and either of the following is true:

- The tool is a prepackaged tool specified as an argument to the –tool flag of an atom command. By default, Atom looks for prepackaged tools in the /usr/lib/cmplrs/atom/tools and /usr/lib/cmplrs/atom/examples directories.
- The file containing the instrumentation routine is specified as the *instrumentation_file* parameter of an atom command.

The instrumentation routine contains the code that traverses the objects, procedures, basic blocks, and instructions to locate instrumentation points; adds calls to analysis procedures; and builds the instrumented version of an application.

As described in the atom_instrumentation_routines(5) reference page, an instrumentation routine can employ one of the following interfaces based on the needs of the tool:

Instrument (int *iargc,* **char *****iargv,* **Obj ****obj* **)**

Atom calls the Instrument routine for each object in the application program. As a result, an Instrument routine does not need to use the object navigation routines (such as GetFirstObj). Because Atom automatically writes each object before passing the next to the Instrument routine, the Instrument routine should never call the BuildObj, WriteObj, or ReleaseObj routine. When using the Instrument interface, you can define an InstrumentInit routine to perform tasks required before Atom calls Instrument for the first object (such as defining analysis routine prototypes, adding program level instrumentation calls, and performing global initializations). You can also define an InstrumentFini routine to perform tasks required after Atom calls Instrument for the last object (such as global cleanup).

InstrumentAll (int *iargc,* **char *****iargv* **)**

Atom calls the InstrumentAll routine once for the entire application program, thus allowing a tool's instrumentation code itself to determine how to traverse the application's objects. With this method, there are no InstrumentInit or InstrumentFini routines. An InstrumentAll routine must call the Atom object navigation routines and use the BuildObj, WriteObj, or ReleaseObj routine to manage the application's objects.

Regardless of the instrumentation routine interface, Atom passes the arguments specified in the `-toolargs` flag to the routine. In the case of the `Instrument` interface, Atom also passes a pointer to the current object.

1.2.3 Atom Instrumentation Interfaces

Atom provides a comprehensive interface for instrumenting applications. The interface supports the following types of activities:

- Navigating among a program's objects, procedures, basic blocks, and instructions. See Section 1.2.3.1.
- Building, releasing, and writing objects. See Section 1.2.3.2.
- Obtaining information about the different components of an application. See Section 1.2.3.3.
- Resolving procedure names and call targets. See Section 1.2.3.4.
- Adding calls to analysis routines at desired locations in the program. See Section 1.2.3.5.

1.2.3.1 Navigating Within a Program

The Atom application navigation routines, described in the `atom_application_navigation`(5) reference page, allow an Atom tool's instrumentation routine to find locations in an application at which to add calls to analysis procedures.

- The `GetFirstObj`, `GetLastObj`, `GetNextObj`, and `GetPrevObj` routines navigate among the objects of a program. For nonshared programs, there is only one object. For call-shared programs, the first object corresponds to the main program. The remaining objects are each of its dynamically linked shared libraries.
- The `GetFirstObjProc` and `GetLastObjProc` routines return a pointer to the first or last procedure, respectively, in the specified object. The `GetNextProc` and `GetPrevProc` routines navigate among the procedures of an object.
- The `GetFirstBlock`, `GetLastBlock`, `GetNextBlock`, and `GetPrevBlock` routines navigate among the basic blocks of a procedure.
- The `GetFirstInst`, `GetLastInst`, `GetNextInst`, and `GetPrevInst` routines navigate among the instructions of a basic block.
- The `GetInstBranchTarget` routine returns a pointer to the instruction that is the target of a specified branch instruction.

- The GetProcObj routine returns a pointer to the object that contains the specified procedure. Similarly, the GetBlockProc routine returns a pointer to the procedure that contains the specified basic block, and the GetInstBlock routine returns a pointer to the basic block that contains the specified instruction.

1.2.3.2 Building Objects

The Atom object management routines, described in the atom_object_management(5) reference page, allow an Atom tool's InstrumentAll routine to build, write, and release objects.

The BuildObj routine builds the internal data structures Atom requires to manipulate the object. An InstrumentAll routine must call the BuildObj routine before traversing the procedures in the object and adding analysis routine calls to the object. The WriteObj routine writes the instrumented version the specified object, deallocating the internal data structures the BuildObj routine previously created. The ReleaseObj routine deallocates the internal data structures for the given object, but does not write out the instrumented version the object.

The IsObjBuilt routine returns a nonzero value if the specified object has been built with the BuildObj routine but not yet written with the WriteObj routine or unbuilt with the ReleaseObj routine.

1.2.3.3 Obtaining Information About an Application's Components

The Atom application query routines, described in the atom_application_query(5) reference page, allow an instrumentation routine to obtain static information about a program and its objects, procedures, basic blocks, and instructions.

The GetAnalName routine returns the name of the analysis file, as passed to the atom command. This routine is useful for tools that have a single instrumentation file and multiple analysis files. For example, multiple cache simulators might share a single instrumentation file but each have a different analysis file.

The GetProgInfo routine returns the number of objects in a program.

Table 1-3 lists the routines that provide information about a program's objects.

Table 1-3: Atom Object Query Routines

Routine	Description
GetObjInfo	Returns information about an object’s text, data, and bss segments; the number of procedures, basic blocks, or instructions it contains; its object ID; or a Boolean hint as to whether the given object should be excluded from instrumentation.
GetObjInstArray	Returns an array consisting of the 32-bit instructions included in the object.
GetObjInstCount	Returns the number of instructions in the array included in the array returned by the GetObjInstArray routine.
GetObjName	Returns the original filename of the specified object.
GetObjOutName	Returns the name of the instrumented object.

The following instrumentation routine, which prints statistics about the program’s objects, demonstrates the use of Atom object query routines:

```
#include <stdio.h>
#include <cmplrs/atom.inst.h>
unsigned InstrumentAll(int argc, char **argv)
{
   Obj *o; Proc *p;
   const unsigned int *textSection;
   long textStart;
   for (o = GetFirstObj(); o != NULL; o = GetNextObj(o)) {
     BuildObj(o);
     textSection = GetObjInstArray(o);
     textStart = GetObjInfo(o,ObjTextStartAddress);
     printf("Object %d\n", GetObjInfo(o,ObjID));
     printf("   Object name: %s\n", GetObjName(o));
     printf("   Text segment start: 0x%lx\n", textStart);
     printf("   Text size: %ld\n", GetObjInfo(o,ObjTextSize));
     printf("   Second instruction: 0x%x\n", textSection[1]);
     ReleaseObj(o);
   }
   return(0);
}
```

Because the instrumention routine adds no procedures to the executable, there is no need for an analysis procedure. The following example demonstrates the process of compiling and instrumenting a program with this tool. A sample run of the instrumented program prints the object identifier, the compile-time starting address of the text segment, the size of the text segment, and the binary for the second instruction. The disassembler provides a convenient method for finding the corresponding instructions.

```
% cc hello.c -o hello

% atom hello info.inst.c -o hello.info
Object 0
  Object Name: hello
  Start Address: 0x120000000
  Text Size: 8192
  Second instruction: 0x239f001d
Object 1
  Object Name: /usr/shlib/libc.so
  Start Address: 0x3ff80080000
  Text Size: 901120
  Second instruction: 0x239f09cb
% dis hello | head -3
  0x120000fe0: a77d8010     ldq t12, -32752(gp)
  0x120000fe4: 239f001d     lda at, 29(zero)
  0x120000fe8: 279c0000     ldah at, 0(at)
% dis /ust/shlib/libc.so | head -3
  0x3ff800bd9b0: a77d8010   ldq t12,-32752(gp)
  0x3ff800bd9b4: 239f09cb   lda at,2507(zero)
  0x3ff800bd9b8: 279c0000   ldah at, 0(at)
```

Table 1-4 lists the routines that provide information about an object's procedures:

Table 1-4: Atom Procedure Query Routines

Routine	Description
`GetProcInfo`	Returns information pertaining to the procedure's stack frame, register-saving, register-usage, and prologue characteristics as defined in the *Calling Standard for Alpha Systems* and the *Assembly Language Programmer's Guide*. Such values are important to tools, like Third Degree, that monitor the stack for access to uninitialized variables. It can also return such information about the procedure as the number of basic blocks or instructions it contains, its procedure ID, its lowest or highest source line number, or an indication if its address has been taken.
`ProcFileName`	Returns the name of the source file that contains the procedure.
`ProcName`	Returns the procedure's name.
`ProcPC`	Returns the compile-time program counter (PC) of the first instruction in the procedure.

Table 1-5 lists the routines that provide information about a procedure's basic blocks:

Table 1-5: Atom Basic Block Query Routines

Routine	Description
`BlockPC`	Returns the compile-time program counter (PC) of the first instruction in the basic block.
`GetBlockInfo`	Returns the number of instructions in the basic block or the block ID. The block ID is unique to this basic block within its containing object.
`IsBranchTarget`	Indicates if the block is the target of a branch instruction.

Table 1-6 lists the routines that provide information about a basic block's instructions:

Table 1-6: Atom Instruction Query Routines

Routine	Description
`GetInstBinary`	Returns a 32-bit binary representation of the assembly language instruction.
`GetInstClass`	Returns the instruction class (for instance, floating-point load or integer store) as defined by the *Alpha Architecture Reference Manual*. An Atom tool uses this information to determine instruction scheduling and dual issue rules.
`GetInstInfo`	Parses the entire 32-bit instruction and obtains all or a portion of that instruction.
`GetInstRegEnum`	Returns the register type (floating-point or integer) from an instruction field as returned by the `GetInstInfo` routine.
`GetInstRegUsage`	Returns a bit mask with one bit set for each possible source register and one bit set for each possible destination register.
`InstPC`	Returns the compile-time program counter (PC) of the instruction.
`InstLineNo`	Returns the instruction's source line number.

Table 1-6: (continued)

Routine	Description
`IsInstType`	Indicates whether the instruction is of the specified type (load instruction, store instruction, conditional branch, or unconditional branch).

1.2.3.4 Resolving Procedure Names and Call Targets

Resolving procedure names and subroutine targets is trivial for nonshared programs because all procedures are contained in the same object. However, the target of a subroutine branch in a call-shared program could be in any object.

The Atom application procedure name and call target resolution routines, described in the `atom_application_resolvers`(5) reference page, allow an Atom tool's instrumentation routine to find a procedure by name and to find a target procedure for a call site:

- The `ResolveTargetProc` routine attempts to resolve the target of a procedure call.
- The `ResolveNamedProc` routine returns the procedure identified by the specified name string.
- The `ReResolveProc` routine completes a procedure resolution if the procedure initially resided in an unbuilt object.

1.2.3.5 Adding Calls to Analysis Routines to a Program

The Atom application instrumentation routines, described in the `atom_application_instrumentation`(5) reference page, add arbitrary procedure calls at various points in the application:

- You must use the `AddCallProto` routine to specify the prototype of each analysis procedure to be added to the program. In other words, an `AddCallProto` call must define the procedural interface for each analysis procedure used in calls to `AddCallProgram`, `AddCallObj`, `AddCallProc`, `AddCallBlock`, and `AddCallInst`. Atom provides facilities for passing integers and floating-point numbers, arrays, branch condition values, effective addresses, cycle counters, as well as procedure arguments and return values.
- Use the `AddCallProgram` routine in an instrumentation routine to add a call to an analysis procedure before a program starts execution or after it completes execution. Typically, such an analysis procedure does

something that applies to the whole program, such as opening an output file or parsing command line options.

- Use the `AddCallObj` routine in an instrumentation routine to add a call to an analysis procedure before an object starts execution or after it completes execution. Typically such an analysis procedure does something that applies to the single object, such as initializing some data for its procedures.
- Use the `AddCallProc` routine in an instrumentation routine to add a call to an analysis procedure before a procedure starts execution or after it completes execution.
- Use the `AddCallBlock` routine in an instrumentation routine to add a call to an analysis procedure before a basic block starts execution or after it completes execution.
- Use the `AddCallInst` routine in an instrumentation routine to add a call to an analysis procedure before a given instruction executes or after it executes.
- Use the `ReplaceProcedure` routine to replace a procedure in the instrumented program. For example, the Third Degree Atom tool replaces memory allocation functions such as `malloc` and `free` with its own versions to allow it to check for invalid memory accesses and memory leaks.

1.2.4 Atom Description File

An Atom tool's description file, as described in the `atom_description_file`(5) reference page, identifies and describes the tool's instrumentation and analysis files. It can also specify the flags to be used by the `cc`, `ld`, and `atom` commands when it is compiled, linked, and invoked. Each Atom tool must supply at least one description file.

There are two types of Atom description file:

- A description file providing an environment for generalized use of the tool. A tool can provide only one general-purpose environment. The name of this type of description file has the format:

  *tool.***desc**

- A description file providing an environment for use of the tool in specific contexts, such as in a multithreaded application or in kernel mode. A tool can provide several special-purpose environments, each of which has its own description file. The name of this type of description file has the format:

  *tool.environment.***desc**

The names supplied for the *tool* and *environment* portions of these description file names correspond to values the user specifies to the `–tool` and `–env` flags of an `atom` command when invoking the tool.

An Atom description file is a text file containing a series of tags and values. See `atom_description_file`(5) for a complete description of the file's syntax.

1.2.5 Writing Analysis Procedures

An instrumented application calls analysis procedures to perform the specific functions defined by an Atom tool. An analysis procedure can use any system call or library function, even if the same call or function is instrumented within the application. The routines used by the analysis routine and the instrumented application are physically distinct.

1.2.5.1 Input/Output

An analysis procedure that uses the standard I/O library should take care to explicitly close file descriptors before the instrumented application exits. The standard I/O library buffers read and write requests to optimize disk accesses. It flushes an output buffer to disk either when it is full or when a procedure calls the `fflush` function. If the instrumented application exits before an analysis procedure properly closes its output file descriptors, the procedure's output may not be completely written.

Some Atom tool analysis procedures may print results to `stdout` or `stderr`. Because the file descriptors for these I/O streams are closed when an instrumented application calls the `exit` function, an analysis routine that is called from an instrumentation point set by a call to the `ProgramAfter` routine can no longer send output to either. Analysis procedures written in C++ must also take care when using the `cout` and `cerr` functions. Because these streams are buffered by the class library, an analysis routine must call `cout.flush()` or `cerr.flush()` before the instrumented application exits.

1.2.5.2 Fork and Exec System Calls

If a process calls a `fork` function but does not call an `exec` function, the process is cloned and the child inherits an exact copy of the parent's state. In many cases, this is exactly the behavior than an Atom tool expects. For example, an instruction-address tracing tool sees references for both the parent and the child, interleaved in the order in which the references occurred.

In the case of an instruction-profiling tool (for example, the `trace` tool referenced in Table 1-2), the file is opened at a `ProgramBefore` instrumentation point and, as a result, the output file descriptor is shared between the parent and the child processes. If the results are printed at a ProgramAfter instrumentation point, the output file contains the parent's data, followed by the child's data (assuming that the parent process finishes first).

For tools that count events, the data structures that hold the counts should be returned to zero in the child process after the fork call because the events occurred in the parent, not the child. This type of Atom tool can support correct handling of fork calls by instrumenting the fork library procedure and calling an analysis procedure with the return value of the fork routine as an argument. If the analysis procedure is passed a return value of 0 (zero) in the argument, it knows that it was called from a child process. It can then reset the counts variable or other data structures so that they tally statistics for only the child process.

1.2.6 Determining the Instrumented PC from an Analysis Routine

The Atom Xlate routines, described in Xlate(5), allow you to determine the instrumented PC for selected instructions. You can use these functions to build a table that translates an instruction's PC in the instrumented application to its PC in the uninstrumented application.

To enable analysis code to determine the instrumented PC of an instruction at runtime, an Atom tool's instrumentation routine must select the instruction and place it into an address translation buffer (XLATE).

An Atom tool's instrumentation routine creates and fills the address translation buffer by calling the CreateXlate and AddXlateAddress routines, respectively. An address translation buffer can only hold instructions from a single object.

The AddXlateAddress routine adds the specified instruction to an existing address translation buffer.

An Atom tool's instrumentation passes an address translation buffer to an analysis routine by passing it as a parameter of type XLATE *, as indicated in the analysis routine's prototype definition in an AddCallProto call.

Another way to determine an instrumented PC is to specify a formal parameter type of REGV in an analysis routine's prototype and pass the REG_IPC value.

An Atom tool's analysis routine uses the following interfaces to access an address translation buffer passed to it:

- The XlateNum routine returns the number of addresses in the specified address translation buffer.
- The XlateInstTextStart routine returns the starting address of the text segment for the instrumented object corresponding to the specified address translation buffer.

- The `XlateInstTextSize` routine returns the size of the text segment.
- The `XlateLoadShift` routine returns the difference between the runtime addresses in the object corresponding to the specified address translation buffer and the compile-time addresses.
- The `XlateAddr` routine returns the instrumented runtime address for the instruction in the specified position of the specified address translation buffer. Note that the runtime address for an instruction in a shared library is not necessarily the same as its compile-time address.

The following example demonstrates the use of the Xlate routines by the instrumentation and analysis files of a tool that uses the Xlate routines. This tool prints the target address of every jump instruction. To use it, issue the following instruction:

% **atom progname xlate.inst.c xlate.anal.c -all**

The following source listing (`xlate.inst.c`) contains the instrumentation for the `xlate` tool:

```
#include <stdlib.h>
#include <stdio.h>
#include <alpha/inst.h>
#include <cmplrs/atom.inst.h>

static void             address_add(unsigned long);
static unsigned         address_num(void);
static unsigned long *  address_paddrs(void);
static void             address_free(void);

void InstrumentInit(int iargc, char **iargv)
{
    /* Create analysis prototypes. */
    AddCallProto("RegisterNumObjs(int)");
    AddCallProto("RegisterXlate(int, XLATE *, long[0])");
    AddCallProto("JmpLog(long, REGV)");

    /* Pass the number of objects to the analysis routines. */
    AddCallProgram(ProgramBefore, "RegisterNumObjs",
        GetProgInfo(ProgNumberObjects));
}

Instrument(int iargc, char **iargv, Obj *obj)
{
    Proc *                      p;
    Block *                     b;
    Inst *                      i;
    Xlate *                     pxlt;
    union alpha_instruction     bin;
    ProcRes                     pres;
    unsigned long               pc;
    char                        proto[128];

    /*
     * Create an XLATE structure for this Obj.  We use this to translate
     * instrumented jump target addresses to pure jump target addresses.
```

```
     */
    pxlt = CreateXlate(obj, XLATE_NOSIZE);

    for (p = GetFirstObjProc(obj);  p;  p = GetNextProc(p)) {
        for (b = GetFirstBlock(p);  b;  b = GetNextBlock(b)) {
            /*
             * If the first instruction in this basic block has had its
             * address taken, it’s a potential jump target.  Add the
             * instruction to the XLATE and keep track of the pure address
             * too.
             */
            i = GetFirstInst(b);
            if (GetInstInfo(i, InstAddrTaken)) {
                AddXlateAddress(pxlt, i);
                address_add(InstPC(i));
            }

            for (;  i;  i = GetNextInst(i)) {
                bin.word = GetInstInfo(i, InstBinary);
                if (bin.common.opcode == op_jsr &&
                    bin.j_format.function == jsr_jmp)
                {
                    /*
                     * This is a jump instruction.  Instrument it.
                     */
                    AddCallInst(i, InstBefore, "JmpLog",  InstPC(i),
                        GetInstInfo(i, InstRB));
                }
            }
        }
    }

    /*
     * Re-prototype the RegisterXlate() analysis routine now that we
     * know the size of the pure address array.
     */
    sprintf(proto, "RegisterXlate(int, XLATE *, long[%d])", address_num());
    AddCallProto(proto);

    /*
     * Pass the XLATE and the pure address array to this object.
     */
    AddCallObj(obj, ObjBefore, "RegisterXlate", GetObjInfo(obj, ObjID),
        pxlt, address_paddrs());

    /*
     * Deallocate the pure address array.
     */
    address_free();
}

/*
** Maintains a dynamic array of pure addresses.
*/
static unsigned long *  pAddrs;
static unsigned         maxAddrs = 0;
static unsigned         nAddrs = 0;

/*
** Add an address to the array.
```

```
*/
static void address_add(
    unsigned long       addr)
{
    /*
     * If there’s not enough room, expand the array.
     */
    if (nAddrs >= maxAddrs) {
        maxAddrs = (nAddrs + 100) * 2;
        pAddrs = realloc(pAddrs, maxAddrs * sizeof(*pAddrs));
        if (!pAddrs) {
            fprintf(stderr, "Out of memory\n");
            exit(1);
        }
    }

    /*
     * Add the address to the array.
     */
    pAddrs[nAddrs++] = addr;
}


/*
** Return the number of elments in the address array.
*/
static unsigned address_num(void)
{
    return(nAddrs);
}


/*
** Return the array of addresses.
*/
static unsigned long *address_paddrs(void)
{
    return(pAddrs);
}

/*
** Deallocate the address array.
*/
static void address_free(void)
{
    free(pAddrs);
    pAddrs = 0;
    maxAddrs = 0;
    nAddrs = 0;
}
```

The following source listing (`xlate.anal.c`) contains the analysis routine for the `xlate` tool:

```
#include <stdlib.h>
#include <stdio.h>
#include <cmplrs/atom.anal.h>

/*
 * Each object in the application gets one of the following data
 * structures.  The XLATE contains the instrumented addresses for
 * all possible jump targets in the object.  The array contains
 * the matching pure addresses.
 */
typedef struct {
    XLATE *             pXlt;
    unsigned long *     pAddrsPure;
} ObjXlt_t;

/*
 * An array with one ObjXlt_t structure for each object in the
 * application.
 */
static ObjXlt_t *       pAllXlts;
static unsigned         nObj;
static int      translate_addr(unsigned long, unsigned long *);
static int      translate_addr_obj(ObjXlt_t *, unsigned long,
                    unsigned long *);

/*
**  Called at ProgramBefore.  Registers the number of objects in
**  this application.
*/
void RegisterNumObjs(
    unsigned    nobj)
{
    /*
     * Allocate an array with one element for each object. The
     * elements are initialized as each object is loaded.
     */
    nObj = nobj;
    pAllXlts = calloc(nobj, sizeof(pAllXlts));
    if (!pAllXlts) {
        fprintf(stderr, "Out of Memory\n");
        exit(1);
    }
}

/*
**  Called at ObjBefore for each object.  Registers an XLATE with
**  instrumented addresses for all possible jump targets.  Also
**  passes an array of pure addresses for all possible jump targets.
*/
void RegisterXlate(
    unsigned            iobj,
    XLATE *             pxlt,
    unsigned long *     paddrs_pure)
{
    /*
     * Initialize this object’s element in the pAllXlts array.
     */
```

```
    pAllXlts[iobj].pXlt = pxlt;
    pAllXlts[iobj].pAddrsPure = paddrs_pure;
}

/*
**  Called at InstBefore for each jump instruction.  Prints the pure
**  target address of the jump.
*/
void JmpLog(
    unsigned long       pc,
    REGV                targ)
{
    unsigned long       addr;

    printf("0x%lx jumps to - ", pc);
    if (translate_addr(targ, &addr))
        printf("0x%lx\n", addr);
    else
        printf("unknown\n");
}

/*
**  Attempt to translate the given instrumented address to its pure
**  equivalent.  Set '*paddr_pure' to the pure address and return 1
**  on success.  Return 0 on failure.
**
**  Will always succede for jump target addresses.
*/
static int translate_addr(
    unsigned long       addr_inst,
    unsigned long *     paddr_pure)
{
    unsigned long       start;
    unsigned long       size;
    unsigned            i;

    /*
     * Find out which object contains this instrumented address.
     */
    for (i = 0;  i < nObj;  i++) {
        start = XlateInstTextStart(pAllXlts[i].pXlt);
        size = XlateInstTextSize(pAllXlts[i].pXlt);
        if (addr_inst >= size && addr_inst < start + size) {
            /*
             * Found the object, translate the address using that
             * object's data.
             */
            return(translate_addr_obj(&pAllXlts[i], addr_inst,
                paddr_pure));
        }
    }

    /*
     * No object contains this address.
     */
    return(0);
}

/*
**  Attempt to translate the given instrumented address to its
```

```
**  pure equivalent using the given object’s translation data.
**  Set ’*paddr_pure’ to the pure address and return 1 on success.
**  Return 0 on failure.
*/
static int translate_addr_obj(
    ObjXlt_t *          pObjXlt,
    unsigned long       addr_inst,
    unsigned long *     paddr_pure)
{
    unsigned    num;
    unsigned    i;

    /*
     * See if the instrumented address matches any element in the XLATE.
     */
    num = XlateNum(pObjXlt->pXlt);
    for (i = 0;  i < num;  i++) {
        if (XlateAddr(pObjXlt->pXlt, i) == addr_inst) {
            /*
             * Matches this XLATE element, return the matching pure
             * address.
             */
            *paddr_pure = pObjXlt->pAddrsPure[i];
            return(1);
        }
    }

    /*
     * No match found, must not be a possible jump target.
     */
    return(0);
}
```

1.2.7 Sample Tools

This section describes the basic tool building interface by using three simple examples: procedure tracing, instruction profiling, and data cache simulation.

1.2.7.1 Procedure Tracing

The `ptrace` tool prints the names of procedures in the order in which they are executed. The implementation adds a call to each procedure in the application. By convention, the instrumentation for the `ptrace` tool is placed in the file `ptrace.inst.c`.

```
#include <stdio.h>
#include <cmplrs/atom.inst.h>  [1]

unsigned InstrumentAll(int argc, char **argv)  [2]
{
   Obj *o; Proc *p;
   AddCallProto("ProTrace(char *)");  [3]
   for (o = GetFirstObj(); o != NULL; o = GetNextObj(o)) {  [4]
     if (BuildObj(o) return 1;  [5]
     for (p = GetFirstObjProc(o); p != NULL; p = GetNextProc(p)) {  [6]
       const char *name = ProcName(p);  [7]
       if (name == NULL) name = "UNKNOWN";  [8]
```

```
13         AddCallProc(p,ProcBefore,"ProcTrace",name);  9
14       }
15       WriteObj(o);  10
16     }
17     return(0);
18  }
```

1. Includes the definitions for Atom instrumentation routines and data structures.
2. Defines the `InstrumentAll` procedure. This instrumentation routine defines the interface to each analysis procedure and inserts calls to those procedures at the correct locations in the applications it instruments.
3. Calls the `AddCallProto` routine to define the `ProcTrace` analysis procedure. `ProcTrace` takes a single argument of type `char *`.
4. Calls the `GetFirstObj` and `GetNextObj` routines to cycle through each object in the application. If the program was linked nonshared, there is only a single object. If the program was linked call-shared, it contains multiple objects – one for the main executable and one for each dynamically-linked shared library. The main program is always the first object.
5. Builds the first object. Objects must be built before they can be used. In very rare circumstances, the object cannot be built. The `InstrumentAll` routine reports this condition to Atom by returning a nonzero value.
6. Calls the `GetFirstObjProc` and `GetNextProc` routines to step through each procedure in the application program.
7. For each procedure, calls the `ProcName` procedure to find the procedure name. Depending on the amount of symbol table information that is available in the application, some procedure names, such as those defined as `static`, may not be available. (Compiling applications with the `–g1` flag provides this level of symbol information.) In these cases, Atom returns `NULL`.
8. Converts the `NULL` procedure name string to ‘‘`UNKNOWN`’’.
9. Calls the `AddCallProc` routine to add a call to the procedure pointed to by `p`. The `ProcBefore` argument indicates that the analysis procedure is to be added before all other instructions in the procedure. The name of the analysis procedure to be called at this instrumentation point is `ProcTrace`. The final argument is to be passed to the analysis procedure. In this case, it is the procedure named obtained on Line 11.
10. Writes the instrumented object file to disk.

The instrumentation file added calls to the `ProcTrace` analysis procedure. This procedure is defined in the analysis file `ptrace.anal.c` as shown in the following example:

```
#include <stdio.h>

void ProcTrace(char *name)
{
  fprintf(stderr, "%s\n",name);
}
```

The ProcTrace analysis procedure prints, to stderr, the character string passed to it as an argument. Note that an analysis procedure cannot return a value.

Once the instrumentation and analysis files are specified, the tool is complete. To illustrate the application of this tool, we compile and link the following application:

```
#include <stdio.h>
main()
{
   printf("Hello world!\n");
}
```

The following example builds a nonshared executable, applies the ptrace tool, and runs the instrumented executable. This simple program calls almost 30 procedures.

```
% cc -non_shared hello.c -o hello
% atom hello ptrace.inst.c ptrace.anal.c -o hello.ptrace
% hello.ptrace
    __start
    main
    printf
    _doprnt
    __getmbcurmax
    strchr
    strlen
    memcpy
    .
    .
    .
```

The following example repeats this process with the application linked call-shared. The major difference is that the LD_LIBRARY_PATH environment variable must be set to the current directory because Atom creates an instrumented version of the libc.so shared library in the local directory.

```
% cc hello.c -o hello
% atom hello ptrace.inst.c ptrace.anal.c -o hello.ptrace
% setenv LD_LIBRARY_PATH ‘pwd‘
% hello.ptrace
    __start
```

```
_call_add_gp_range
__exc_add_gp_range
malloc
cartesian_alloc
cartesian_growheap2
__getpagesize
__sbrk
.
.
.
```

The call-shared version of the application calls almost twice the number of procedures that the nonshared version calls.

Note that only calls in the original application program are instrumented. Because the call to the `ProcTrace` analysis procedure did not occur in the original application, it does not appear in a trace of the instrumented application procedures. Likewise, the standard library calls that print the names of each procedure are also not included. If the application and the analysis program both call the `printf` function, Atom would link into the instrumented application two copies of the function. Only the copy in the application program would be instrumented. Atom also correctly instruments procedures that have multiple entry points.

1.2.7.2 Profile Tool

The `prof` example tool counts the number of instructions a program executes. It is useful for finding critical sections of code. Each time the application is executed, `prof` creates a file called `prof.out` that contains a profile of the number of instructions that are executed in each procedure.

The most efficient place to compute instruction counts is inside each basic block. Each time a basic block is executed, a fixed number of instructions are executed. The following example shows how the `prof` tool's instrumentation procedure (`prof.inst.c`) performs these tasks:

```
#include <stdio.h>
#include <cmplrs/atom.inst.h>

unsigned InstrumentAll(int argc, char **argv)
{
  Obj *o; Proc *p; Block *b; Inst *i;
  int n = 0;
  AddCallProto("OpenFile(int)");  1
  AddCallProto("Count(int,int)");
  AddCallProto("Print(int,char *)");
  AddCallProto("CloseFile()");
  for (o = GetFirstObj(); o != NULL; o = GetNextObj(o)) {  2
    if (BuildObj(o)) return (1);  3
    for (p = GetFirstObjProc(o); p != NULL; p = GetNextProc(p)) {  4
      const char *name = ProcName(p);  5
      if (name == NULL) name = "UNKNOWN";
```

```
17        for (b = GetFirstBlock(p); b != NULL; b = GetNextBlock(b)) {  6
18          AddCallBlock(b,BlockBefore,"Count",n,  7
              GetBlockInfo(b,BlockNumberInsts));
19        }
20        AddCallProgram(ProgramAfter,"Print",n,name);  8
21        n++;  9
22      }
23      WriteObj(o);  10
24    }
25    AddCallProgram(ProgramBefore,"OpenFile",n);  11
26    AddCallProgram(ProgramAfter,"CloseFile");  12
27    return (0);
28  }
```

1 Defines the interface to the analysis procedures.

2 Loops through each object in the program.

3 Builds an object.

4 Loops through each procedure in the object.

5 Determines the procedure name.

6 Loops through each basic block in the procedure.

7 Adds a call to the `Count` analysis procedure before any of the instructions in this basic block are executed. The argument types of the `Count` are defined in the prototype on Line 9. The first argument is a procedure index of type `int`; the second argument, also an `int`, is the number of instructions in the basic block. The `Count` analysis procedure adds the number of instructions in the basic block to a per-procedure data structure.

8 Adds a call to the `Print` analysis procedure to the end of the program. The `Print` analysis procedure prints a line summarizing this procedure's instruction use.

9 Increments the procedure index.

10 Writes the object file.

11 Adds a call to the `OpenFile` analysis procedure to the beginning of the program, passing it an `int` representing the number of procedures in the application. The `OpenFile` procedure allocates the per-procedure data structure that tallies instructions and opens the output file.

12 Adds a call to the `CloseFile` analysis procedure to the end of the program.

The analysis procedures used by the `prof` tool are defined in the `prof.anal.c` file as shown in the following example:

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

long *instrPerProc;
FILE *file;

void OpenFile(int n)
{
  instrPerProc = (long *) calloc(sizeof(long),n);  1
  assert(instrPerProc != NULL);
  file = fopen("prof.out","w");
  assert(file != NULL);
  fprintf(file,"%30s %15s %10s\n","Procedure","Instructions","Percentage");
}
void Count(int n, int instructions)
{
  instrTotal += instructions;
  instrPerProc[n] += instructions;
}
void Print(int n, char *name)
{
  if (instrPerProc[n] > 0) {  2
    fprintf(file,"%30s %15ld %9.3f\n", name, instrPerProc[n],
            ((float) instrPerProc[n] / instrTotal)*100.0);
  }
}
void CloseFile()  3
{
  fprintf(file,"\n%30s %15ld %9.3f\n", "Total", instrTotal,100.0);
  fclose(file);
}
```

1 Allocates the counts data structure. The `calloc` function zero-fills the counts data.

2 Filters procedures that are never called.

3 Closes the output file. Tools must explicitly close files that are opened in the analysis procedures.

Once the instrumentation and analysis files are specified, the tool is complete. To illustrate the application of this tool, we compile and link the "Hello" application:

```
#include <stdio.h>
main()
{
   printf("Hello world!\n");
}
```

The following example builds a call-shared executable, applies the `prof` tool, and runs the instrumented executable. In contrast to the `ptrace` tool described in Section 1.2.7.1, the `prof` tool sends its output to a file instead of stdout.

```
% cc hello.c -o hello
% atom hello prof.inst.c prof.anal.c -o hello.prof
% setenv LD_LIBRARY_PATH ‘pwd‘
% hello.prof
Hello world!
% more prof.out
                   Procedure    Instructions Percentage
                      __start            159      4.941
                         main             14      0.435
                        .
                        .
                        .
            _call_add_gp_range            41      1.274
         _call_remove_gp_range            35      1.088

                        Total           3218    100.000
% unsetenv LD_LIBRARY_PATH
```

1.2.7.3 Data Cache Simulation Tool

Instruction and data address tracing has been used for many years as a technique for capturing and analyzing cache behavior. Unfortunately, current machine speeds make this increasingly difficult. For example, the Alvinn SPEC92 benchmark executes 961,082,150 loads, 260,196,942 stores, and 73,687,356 basic blocks, for a total of 2,603,010,614 Alpha instructions. Storing the address of each basic block and the effective address of all the loads and stores would take in excess of 10GB and slow down the application by a factor of over 100.

The `cache` tool uses on-the-fly simulation to determine the cache miss rates of an application running in an 8KB direct mapped cache. The following example shows its instrumentation routine:

```
#include <stdio.h>
#include <cmplrs/atom.inst.h>

unsigned InstrumentAll(int argc, char **argv)
{
  Obj *o; Proc *p;  Block *b;  Inst *i;

  AddCallProto("Reference(VALUE)");
  AddCallProto("Print()");
  for (o = GetFirstObj(); o != NULL; o = GetNextObj(o)) {
    if (BuildObj(o)) return (1);
    for (p=GetFirstProc(); p != NULL; p = GetNextProc(p)) {
      for (b = GetFirstBlock(p); b != NULL;  b = GetNextBlock(b)) {
        for (i = GetFirstInst(b); i != NULL; i = GetNextInst(i)) {  1
          if (IsInstType(i,InstTypeLoad) || IsInstType(i,InstTypeStore)) {
            AddCallInst(i,InstBefore,"Reference",EffAddrValue);  2
          }
        }
```

```
      }
    }
    WriteObj(o);
  }
AddCallProgram(ProgramAfter,"Print");
return (0);
}
```

1. Examines each instruction in the current basic block.
2. If the instruction is a load or a store, adds a call to the `Reference` analysis procedure, passing the effective address of the data reference.

The analysis procedures used by the `cache` tool are defined in the `cache.anal.c` file as shown in the following example:

```
#include <stdio.h>
#include <assert.h>
#define CACHE_SIZE 8192
#define BLOCK_SHIFT 5
long tags[CACHE_SIZE >> BLOCK_SHIFT];
long references, misses;

void Reference(long address) {
  int index = (address & (CACHE_SIZE-1)) >> BLOCK_SHIFT;
  long tag = address >> BLOCK_SHIFT;
  if tags[index] != tag) {
    misses++;
    tags[index] = tag;
  }
  references++;
}
void Print() {
  FILE *file = fopen("cache.out","w");
  assert(file != NULL);
  fprintf(file,"References: %ld\n", references);
  fprintf(file,"Cache Misses: %ld\n", misses);
  fprintf(file,"Cache Miss Rate: %f\n", (100.0 * misses) / references);
  fclose(file);
}
```

Once the instrumentation and analysis files are specified, the tool is complete. To illustrate the application of this tool, we compile and link the "Hello" application:

```
#include <stdio.h>
main()
{
   printf("Hello world!\n");
}
```

The following example applies the `cache` tool to instrument both the nonshared and call-shared versions of the application:

```
% cc hello.c -o hello
% atom hello cache.inst.c cache.anal.c -o hello.cache -all
% setenv LD_LIBRARY_PATH ‘pwd‘
% hello.cache
Hello world!
% more cache.out
References: 1091
Cache Misses: 225
Cache Miss Rate: 20.623281
% cc -non_shared hello.c -o hello
% atom hello cache.inst.c cache.anal.c -o hello.cache -all
% hello.cache
Hello world!
% more cache.out
References: 382
Cache Misses: 93
Cache Miss Rate: 24.345550
```

Debugging Programs with Third Degree 2

Third Degree is an Atom tool. It performs memory access checks and memory leak detection of C and C++ programs at run time. It accomplishes this by using Atom to instrument executable objects. Instrumentation is the process of inserting instructions into existing executable objects to perform program analysis. See Chapter 1 or `atom`(1) for details on Atom.

Third Degree instruments the entire program, adding code to perform run-time checks for all of its data references. The instrumented program locates many occurrences of the worst types of bugs in C and C++ programs: array overflows, memory smashing, and errors in the use of the `malloc` and `free` functions. It also helps you determine the allocation habits of your application by listing the heap and finding memory leaks.

Except for being larger and running slower than the original application and having its uninitialized data filled with a special pattern, the instrumented program runs like the original. The Atom instrumentation code logs all specified errors and generates the requested reports.

You can use Third Degree for the following types of applications:

- Applications that allocate memory by using the `malloc`, `calloc`, `realloc`, `valloc`, `alloca`, and `sbrk` functions and the C++ `new` function. You can use Third Degree to instrument programs using other memory allocators, such as the `mmap` function, but it will not check accesses to the memory thus obtained.

  Third Degree detects and forbids calls to the `brk` function. Furthermore, if your program allocates memory by partitioning large blocks it obtained by using the `sbrk` function, Third Degree may not be able to precisely identify memory blocks in which errors occur.

- Applications using POSIX threads (`pthread`) interfaces and applications using a supported coroutine package. Most coroutine packages are supported. If your application uses a custom threads or coroutine package, you may not be able to use Third Degree. See Section 2.1.2 for details.

2.1 Running Third Degree on an Application

You invoke the Third Degree tool by using the `atom` command, as follows:

```
% atom app -tool third
```

In this example, `app` is the name of an application. When it is run, the instrumented version of the application (`app.third`) behaves exactly like the original application (`app`), with the following exceptions:

- The code is larger and runs more slowly because of the additional instrumentation code that is inserted.
- Each allocated heap memory object is larger because Third Degree pads it to allow boundary checking. You can adjust the amount of padding by specifying the `object_padding` option in the `.third` file. (See Section 2.2.1 for a description of the `.third` customization file.)
- To detect errant use of uninitialized data, Third Degree initializes all otherwise uninitialized data to a special pattern. This can cause the instrumented program to behave differently, behave incorrectly, or crash (particularly if this special pattern is used as a pointer). All of these behaviors indicate a bug in the program.

  You can disable Third Degree's initialization with the `–uninit_heap` and `–uninit_stack` option in the `.third` customization file.

The instrumented version of the application generates a log file (`app.3log`) containing information about allocated objects and potential leaks.

Note

> Third Degree writes `.3log` messages in a format similar to that used by the C compiler. If you use `emacs` or a similar editor that automatically points, in sequence, to each compilation error, you can use the same editor to follow Third Degree errors. In `emacs`, compile with a command such as `cat app.3log`, and step through the Third Degree errors as if they were compilation errors.

You can control the name used for the output log file by specifying one of the following flags to the `–toolargs` flag on the `atom` command line that invokes the Third Degree tool:

`–pids`
: Appends the process identification number to the log file name.

`–nopids`
: Does not append the process identification number to the log file name. This is the default.

`–dirname`*fname*
Specifies the directory path in which Third Degree creates its log file.

Depending upon the flag supplied to Third Degree in the `atom` command's `–toolargs` flag, the log file's name will be as follows:

Flag	Filename	Use
`–nopids`	`app.3log`	Default
`–pids`	`app.12345.3log`	Include pid
`–dirname /tmp`	`/tmp/app.3log`	Set directory
`–dirname /tmp –pids`	`/tmp/app.12345.3log`	Set directory and pid

2.1.1 Using Third Degree with Shared Libraries

Errors in an application, such as passing too small a buffer to the `strcpy` function, are often caught in library routines. Third Degree supports the instrumentation of shared libraries; it instruments programs linked with the `–non_shared` or `–call_shared` flags.

The `atom` command provides the following flags to allow you to determine which shared libraries are instrumented by Third Degree:

`–all`
Instruments all statically loaded shared libraries in the shared executable.

`–excobj` *objname*
Excludes the named shared library from instrumentation. You can use the `–excobj` flag more than once to specify several shared libraries.

`–incobj` *objname*
Instruments the named shared library. You can use the `–incobj` flag more than once to specify several shared libraries.

When Atom finishes instrumenting the application, the current directory contains an instrumented version of each specified shared library. The instrumented application uses these versions of the libraries. Define the `LD_LIBRARY_PATH` environment variable to tell the instrumented application where the instrumented shared libraries reside.

By default, Third Degree does not instrument any of the shared libraries used by the application; this makes the instrumentation operation much faster and causes the instrumented application to run faster as well. Third Degree detects and reports errors in the instrumented portion normally, but terminates stack traces at the first uninstrumented procedure. It does not detect errors in the uninstrumented libraries. If your partially instrumented application crashes or malfunctions and you have fixed all of the errors reported by Third Degree, reinstrument the application with all of its shared libraries and run the new instrumented version.

2.1.2 Using Third Degree with Threaded Applications

Third Degree supports applications that use threads. To instrument a threaded application, add the `–env threads` flag to the `atom` command line that invokes the Third Degree tool.

2.2 Step-by-Step Example

Assume that you must debug the small application represented by the following source code (`ex.c`):

```
/* ex.c */
#include <assert.h>;

int Bug() {
    int q;
    return q;           /* q is uninitialized */
}

long* Booboo(int n) {
    long* t = (long*) malloc(n * sizeof(long));
    t[0] = Bug();
    t[0] = t[1]+1;      /* t[1] is uninitialized */
    t[1] = -1;
    t[n] = n;           /* array bounds error*/
    if (n<10) free(t);  /* may be a leak */
    return t;
}

main() {
    long* t = Booboo(20);
    t = Booboo(4);
    free(t);            /* already freed */
    exit(0);
}
```

2.2.1 Customizing Third Degree

An optional customization file named `.third` is used to turn on and off various capabilities of the Third Degree tool and to set the tool's internal parameters. Third Degree looks for a `.third` file first in the local directory, then in your home directory. The `.third` customization file is further discussed throughout this chapter and its syntax is described in the `third`(5) reference page.

If you do not specify a `.third` customization file, Third Degree uses its default settings:

- List memory errors
- Detect leaks at program exit
- No heap history

2.2.2 Modifying the Makefile

Add the following entry to the application's Makefile:

```
ex.third: ex
        atom ex -tool third -o ex.third
```

Build `ex.third` as follows:

```
> make ex.third
atom ex -tool third -o ex.third
> ex.third
```

Now run the instrumented application `ex.third` and check the log `ex.3log`.

2.2.3 Examining the Third Degree Log File

The `ex.3log` file contains several sections, described in the following sections.

2.2.3.1 Copy of the .third File

If you supplied a `.third` customization file, Third Degree copies it to the log file. The short customization file used in this example requests a summary of the contents of heap-allocated memory blocks when the program finishes:

```
//////////////  begin .3rd  ///////////////////
-----------------------------------------------
heap_history    yes
-----------------------------------------------
//////////////  end  .3rd ///////////////////
```

2.2.3.2 List of Runtime Memory Access Errors

The types of errors that Third Degree can detect at runtime include such conditions as reading uninitialized memory, reading or writing unallocated memory, freeing invalid memory, and certain serious errors likely to cause an exception. For each error, an error entry is generated with the following items:

- A banner line with the type of error and number – The error banner line contains a three-letter abbreviation of each error (see Section 2.3 for a list of the abbreviations). If the process that caused the error is not the root process (for instance, because the application forks one or more child processes), the process id of the process that caused the error also appears in the banner line.
- An error message line formatted to look like a compiler error message – Third Degree lists the file name and line number nearest to the location where the error occurred. Usually this is the precise location where the error occurred, but if the error occurs in a library routine, it may well point to the place where the library call occurred.
- One or more stack traces – The last part of an error entry is a stack trace. The first procedure listed in the stack trace is the procedure in which the error occurred.

The following examples show entries from the log file:

- The following log entry indicates that a local variable of procedure `Bug` was read before being initialized. The line number confirms that `q` was never given a value.

```
---------------------------------------------------- rus -- 0 --
ex.c: 6: reading uninitialized local variable q of Bug
    Bug                               ex.c, line 6
    Booboo                            ex.c, line 11
    main                              ex.c, line 20
    __start                           crt0.s, line 370
```

- The following log entry indicates that an error occurred at line 12:

  `t[0] = t[1]+1`

  Because the array was not initialized, the program is using the uninitialized value of `t[1]` in the addition. The memory block containing array `t` is identified by the call stack that allocated it.

```
---------------------------------------------------- ruh -- 1 --
ex.c: 12: reading uninitialized heap at byte 8 of 160-byte block
    Booboo                            ex.c, line 12
    main                              ex.c, line 20
    __start                           crt0.s, line 370

This block at address 0x38000000f10 was allocated at:
    malloc                            malloc.c, line 585
    Booboo                            ex.c, line 10
    main                              ex.c, line 20
    __start                           crt0.s, line 370
```

- The following log entry indicates that the program has written to the memory location one position past the end of the array, potentially overwriting important data or even Third Degree internal data structures. Keep in mind that certain errors reported later could be a consequence of this error.

```
---------------------------------------------------- wih -- 2 --
ex.c: 14: writing invalid heap 1 byte beyond 160-byte block
    Booboo                         ex.c, line 14
    main                           ex.c, line 20
    __start                        crt0.s, line 370

This block at address 0x38000000f10 was allocated at:
    malloc                         malloc.c, line 585
    Booboo                         ex.c, line 10
    main                           ex.c, line 20
    __start                        crt0.s, line 370
```

- The following log entry indicates that an error occurred while freeing memory that was previously freed. For errors involving calls to the `free` function, Third Degree usually gives three call stacks:
  - The call stack where the error occurred
  - The call stack where the object was allocated.
  - The call stack where the object was freed.

  Upon examining the program, it is clear that the second call to `Booboo` (line 20) frees the object (line 14), and that another attempt to free the same object occurs at line 21.

```
---------------------------------------------------- fof -- 3 --
ex.c: 22: freeing already freed heap at byte 0 of 32-byte block
    free                           malloc.c, line 833
    main                           ex.c, line 22
    __start                        crt0.s, line 370

This block at address 0x380000011a0 was allocated at:
    malloc                         malloc.c, line 585
    Booboo                         ex.c, line 10
    main                           ex.c, line 21
    __start                        crt0.s, line 370

This block was freed at:
    free                           malloc.c, line 833
    Booboo                         ex.c, line 15
    main                           ex.c, line 21
    __start                        crt0.s, line 370
```

2.2.3.3 Memory Leaks

The following excerpt shows the report generated when leak detection on program exit, the default, is selected. The report shows a list of memory leaks sorted by importance and by call stack.

```
----------------------------------------------------------------
----------------------------------------------------------------
Searching for new leaks in heap after program exit

160 bytes in 1 object were found:

160 bytes in 1 leak (including 1 super leak) created at:
    malloc                         malloc.c, line 585
    Booboo                         ex.c, line 10
    main                           ex.c, line 20
    __start                        crt0.s, line 370
```

Upon examining the source, it is clear that the first call of `Booboo` did not free the memory object, nor was it freed anywhere else in the program. Moreover, no pointer to this object exists anywhere in the program, so it qualifies as a super leak. The distinction is often useful to find the real culprit for large memory leaks.

Consider a large tree structure and assume that the pointer to the root has been erased. Every object in the structure is a leak, but losing the pointer to the root is the real cause of the leak. Because all objects but the root still have pointers to them, albeit only from other leaks, only the root will be identified as a super leak, and therefore the likely cause of the memory loss.

2.2.3.4 Heap History

When heap history is enabled, Third Degree collects information about dynamically allocated memory. It collects this information for every object that is freed by the application and for every object that still exists (including memory leaks) at the end of the program's execution. The following excerpt shows a heap allocation history report:

```
----------------------------------------------------------------
----------------------------------------------------------------
                Heap Allocation History for parent process


Legend for object contents:
     There is one character for each 32-bit word of contents.
     There are 64 characters, representing 256 bytes of memory
     per line.
     '.' : word never written in any object.
     'z' : zero in every object.
     'i' : a non-zero non-pointer value in at least one object.
     'pp': a valid pointer or zero in every object.
     'ss': a valid pointer or zero in some but not all objects.

192 bytes in 2 objects were allocated during program execution:

----------------------------------------------------------------
160 bytes allocated (5% written) in 1 objects created at:
```

```
    malloc                          malloc.c, line 585
    Booboo                          ex.c, line 10
    main                            ex.c, line 20
    __start                         crt0.s, line 370

Contents:
     0: ..ii....................................

----------------------------------------------------------------
32 bytes allocated (25% written) in 1 objects created at:
    malloc                          malloc.c, line 585
    Booboo                          ex.c, line 10
    main                            ex.c, line 21
    __start                         crt0.s, line 370

Contents:
     0: ..ii....
```

The sample program allocated two objects, for a total of 192 bytes (8*(20+4)). Because each object was allocated from a different call stack, there are two entries in the history. Only one long (8 bytes) in each array was set to a valid value, resulting in the written ratios of 8/160=5% and 8/32=25% shown. The character map, with one character for each 32-bit word in the object, shows that the initialized value was the second long in each of the arrays.

If the sample program was a real application, the fact that so little of the dynamic memory was ever initialized is a warning that it was probably using memory ineffectively.

2.2.3.5 Memory Layout

The memory layout section of the report summarizes the memory used by the program by size and address range. The following excerpt shows a memory layout section. The first two entries give the final (maximum) sizes of the heap and stack at the end of the program. The last two entries give the text and static data areas for the program and any shared libraries.

```
-----------------------------------------------------------------
-----------------------------------------------------------------
  memory layout at program exit
              heap      81920 bytes [0x38000000000-0x38000014000]
             stack       2224 bytes [0x11ffff750-0x120000000]
           ex data      23168 bytes [0x140000000-0x140005a80]
           ex text     262144 bytes [0x120000000-0x120040000]
```

2.3 Interpreting Third Degree Error Messages

Third Degree reports both fatal errors and memory access errors.

Fatal errors include the following:

- Bad parameter

  For example, `malloc(-10)`.

- Failed allocator

  For example, `malloc` returned a zero, indicating that no memory is available.

- Call to the `brk` function with a nonzero argument

  Third Degree does not allow you to call `brk` with a nonzero argument.

A fatal error causes the instrumented application to crash after flushing the log file. If the application crashes, first check the log file and then rerun it under a debugger.

Memory errors include the following (as represented by a three-letter abbreviation):

Name	Error
`ror`	Reading out of range: neither in heap, stack, or static area
`ris`	Reading invalid data in stack: probably an array bound error
`rus`	Reading an uninitialized (but valid) location in stack
`rih`	Reading invalid data in heap: probably an array bound error
`ruh`	Reading an uninitialized (but valid) location in heap
`wor`	Writing out of range: neither in heap, stack, or static area
`wis`	Writing invalid data in stack: probably an array bound error
`wih`	Writing invalid data in heap: probably an array bound error
`for`	Freeing out of range: neither in heap or stack
`fis`	Freeing an address in the stack
`fih`	Freeing an invalid address in the heap: no valid object there
`fof`	Freeing an already freed object
`fon`	Freeing a null pointer (really just a warning)
`mrn`	`malloc` returned null

You can suppress the reporting of specific memory errors by providing a `.third` customization file containing the `ignore` option. This is often useful when the errors occur within library functions for which you do not have the source. Third Degree allows you to suppress specific memory errors in individual procedures and files, and at particular line numbers. See `third`(5) for more details.

2.3.1 Fixing Errors and Retrying an Application

If Third Degree reports many write errors from your instrumented program, you should fix the first few errors and reinstrument the program. Not only can write errors compound, but they can also corrupt Third Degree's internal data structures.

2.3.2 Detecting Uninitialized Values

Third Degree's technique for detecting the use of uninitialized values can cause programs that have worked to fail when instrumented. For example, if a program depends on the fact that the first call to the `malloc` function returns a block initialized to zero, the instrumented version of the program will fail because Third Degree initializes all blocks to a nonzero value.

When it detects a signal, perhaps caused by dereferencing or otherwise using this uninitialized value, Third Degree displays a message of the following form:

```
*** Fatal signal SIGSEGV detected.
*** This can be caused by the use of uninitialized data.
*** Please check all errors reported in app.3log.
```

Using uninitialized data is the most likely reason for an instrumented program to crash. To determine the cause of the problem, first examine the log file for reading-uninitialized-stack and reading-uninitialized heap errors. Very often, one of the last errors in the log file reports the cause of the problem.

If you have trouble pinpointing the source of the error, you can confirm that it is indeed due to reading uninitialized data by supplying a `.third` customization file containing the `uninit_heap no` and `uninit_stack no` options. Using the `uninit_stack no` option disables the initialization of newly allocated stack memory that Third Degree normally performs on each procedure entry. Similarly, the `uninit_heap no` option disables the initialization of heap memory performed on each dynamic memory allocation. By using one or both options, you can alter the behavior of the instrumented program and may likely get it to complete successfully. This will help you determine which type of error is causing the instrumented program to crash and, as a result, help you focus on specific messages in the log file.

Notes

Do not use the `uninit_heap no` and `uninit_stack no` options under normal operation. They hamper Third Degree's ability to detect a program's use of uninitialized data.

If your program establishes signal handlers, there is a small chance that Third Degree's changing of the default signal handler may interfere with it. Third Degree defines signal handlers only for those signals that normally cause program crashes (including `SIGILL`, `SIGTRAP`, `SIGABRT`, `SIGEMT`, `SIGFPE`, `SIGBUS`, `SIGSEGV`, `SIGSYS`, `SIGXCPU`, and `SIGXFSZ`). You can disable Third Degree's signal handling by supplying a `.third` customization file including the `signals no` option.

2.3.3 Locating Source Files

Third Degree prefixes each error message with a file and line number in the style used by compilers. For example:

```
----------------------------------------------------- fof -- 3 --
ex.c: 21: freeing already freed heap at byte 0 of 32-byte block
    free                                   malloc.c
    main                                   ex.c, line 21
    __start                                crt0.s
```

Third Degree tries to point as closely as possible to the source of the error, and it usually gives the file and line number of a procedure near the top of the call stack when the error occurred, as in this example. However, Third Degree may not be able to find this source file, either because it is in a library or because it is not in the current directory. In this case, Third Degree moves down the call stack until it finds a source file to which it can point. Usually, this is the point of call of the library routine.

In order to tag these error messages, Third Degree must determine the location of the program's source files. If you are running Third Degree in the directory containing the source files, Third Degree will locate the source files there. If not, to add directories to Third Degree's search path, supply a `.third` customization file including a `use` option. This allows Third Degree to find the source files contained in other directories. Specifying the `use` option with no arguments clears the search path. The location of each source file is the first directory on the search path in which it is found.

2.4 Examining an Application's Heap Usage

In addition to run-time checks that ensure that only properly allocated memory is accessed and freed, Third Degree provides two ways to understand an application's heap usage:

- It can find and report memory leaks.
- It can list the contents of the heap.

By default, Third Degree checks for leaks when the program exits.

This section discusses how to use the information provided by Third Degree to analyze an application's heap usage.

2.4.1 Detecting Memory Leaks

A memory leak is an object in the heap to which no pointer exists. The object can no longer be accessed and can no longer be used or freed. It is useless and will never go away.

Third Degree finds memory leaks by using a simple trace-and-sweep algorithm. Starting from a set of roots (the currently active stack and static area), Third Degree finds pointers to objects in the heap and marks these objects as visited. It then recursively finds all potential pointers inside these objects and, finally, sweeps the heap and reports all unmarked objects. These unmarked objects are leaks.

The trace-and-sweep algorithm finds all leaks, including circular structures. This algorithm is conservative: in the absence of type information, any 64-bit pattern that is properly aligned and pointing inside a valid object in the heap is treated as a pointer. This assumption can infrequently lead to the following problems:

- Third Degree considers pointers either to the beginning or interior of an object as true pointers. Only objects with no pointers to any address they contain are considered leaks.
- If an instrumented application hides true pointers by storing them in the address space of some other process or by encoding them, Third Degree will report spurious leaks. When instrumenting such an application with Third Degree, create a `.third` configuration file that specifies the `pointer_mask` option. The `pointer_mask` option lets you specify a mask that is applied as an AND operator against every potential pointer. For example, if you use the top 3 bits of pointers as flags, specify a mask of 0x1fffffffffffffff. See `third`(5) for additional information on `.third` configuration files.
- Third Degree can confuse any bit pattern (such as string, integer, floating-point number, and packed struct) that looks like a heap pointer with a true pointer, thereby missing a true leak.
- Third Degree does not notice pointers that optimized code stores only in registers, not in memory. As a result, it may produce false leak reports.

2.4.2 Reading Heap and Leak Reports

You can supply `.third` configuration file options that tell Third Degree to generate heap and leak reports incrementally, listing only new heap objects or leaks since the last report or listing all heap objects or leaks. You can request these reports when the program terminates, or before or after every *n*th call to a user-specified function (see `third`(5) for details).

Third Degree lists memory objects and leaks in the report by decreasing importance, based on the number of bytes involved. It groups together objects allocated with identical call stacks. For example, if the same call sequence allocates a million one-byte objects, Third Degree reports them as a one-megabyte group containing a million allocations.

To tell Third Degree when objects or leaks are the same and should be grouped in the report (or when objects or leaks are different and should not be thus grouped), specify a `.third` configuration file containing the `object_stack_depth` or `leak_stack_depth` option. (See `third`(5) for further description of the `.third` configuration file.) These options set the depth of the call stack that Third Degree uses to differentiate leaks or objects. For example, if you specify a depth of 1 for objects, Third Degree groups valid objects in the heap by the function and line number that allocated them, no matter what function was the caller. Conversely, if you specify a very large depth for leaks, Third Degree groups only leaks allocated at points with identical call stacks from `main` upwards.

In most heap reports, the first few entries account for most of the storage, but there is a very long list of small entries. To limit the length of the report, you can use the `.third` configuration file `object_min_percent` or `leak_min_percent` option. (See `third`(5) for further description of the `.third` configuration file.) These options define a percentage of the total memory leaked or in use by an object as a threshold. When all smaller remaining leaks or objects amount to less than this threshold, Third Degree groups them together under a single final entry.

Notes

Because the `realloc` function always allocates a new object (by involving calls to `malloc`, `copy`, and `free`), its use can make interpretation of a Third Degree report counterintuitive. When an object is allocated, listed, or shrunk through a call to the `realloc` function, it can be listed twice under different identities.

Leaks and objects are mutually exclusive: an object must be reachable from the roots.

2.4.3 Searching for Leaks

It may not always be obvious when to search for memory leaks. By default, Third Degree checks for leaks after program exit, but this may not always be what you want.

Leak detection is best done as near as possible to the end of the program while all used data structures are still in scope. Remember, though, that the roots for leak detection are the contents of the stack and static areas. If your program terminates by returning from `main` and the only pointer to one of its data structures was kept on the stack, this pointer will not be seen as a root during the leak search, leading to false reporting of leaked memory. For example:

```
main (int argc, char* argv[]) {
    char* bytes = (char*) malloc(100);
    exit(0);
}
```

When you instrument a program, providing a `.third` configuration file specifying the `all leaks before exit every 1` option line will result in Third Degree not finding any leaks. When the program calls the `exit` function, all of `main`'s variables are still in scope.

However, consider the following example:

```
main (int argc, char* argv[]) {
    char* bytes = (char*) malloc(100);
}
```

When you instrument this program, providing the same (or no) `.third` configuration file, Third Degree's leak check may report a storage leak because `main` has returned by the time the check happens. Either of these two behaviors may be correct, depending on whether `bytes` was a true leak or simply a data structure still in use when `main` returned.

Rather than reading the program carefully to understand when leak detection should be performed, you can check for new leaks after a specified number of memory allocations. The number of allocations depends on the characteristics of the application being instrumented. Use a `.third` configuration file specifying the following options:

```
no leaks at_exit
new leaks before proc_name every 10000
```

See `third`(5) for further description of the `.third` configuration file.

2.4.4 Interpreting the Heap History

When you instrument this program, providing a `.third` configuration file specifying the `heap_history yes` option line allows Third Degree to generate a heap history for the program. A heap history allows you to see how the program used dynamic memory during its execution. You can use this feature, for instance, to eliminate unused fields in data structures or to pack active fields to use memory more efficiently. The heap history also shows memory blocks that are allocated but never used by the application.

When heap history is enabled, Third Degree collects information about each dynamically allocated object at the time it is freed by the application. When program execution completes, Third Degree assembles this information for every object that is still alive (including memory leaks). For each object, Third Degree looks at the contents of the object and categorizes each word as never written by the application, zero, a valid pointer, or some other value.

Third Degree next merges the information for each object with what it has gathered for all other objects allocated at the same call stack in the program. The result provides you with a cumulative picture of the use of all objects of a given type.

Third Degree provides a summary of all objects allocated during the life of the program and the purposes for which their contents were used. The report shows one entry per allocation point (for example, a call stack where an allocator function such as `malloc` or `new` was called). Entries are sorted by decreasing volume of allocation.

Each entry provides the following:

- Information about all objects that have been allocated at any point up to this point of the program’s execution
- Total number of bytes allocated at this point of the program’s execution
- Total number of objects that have been allocated up to this point of the program’s execution
- Percentage of bytes of the allocated objects that have been written
- The call stack and a cumulative map of the contents of all objects allocated by that call stack

The contents part of each entry describes how the objects allocated at this point were used. If all allocated objects are not the same size, Third Degree considers only the minimum size common to all objects. For very large allocations, it summarizes the contents of only the beginning of the objects, by default, the first kilobyte. You can adjust the maximum size value by specifying the `history_size` option in the `third` configuration file.

In the contents portion of an entry, Third Degree uses one of the following characters to represent each 32-bit longword that it examines:

Character	Description
Dot (.)	Indicates a longword that was never written in any of the objects, a definite sign of wasted memory. Further analysis is generally required to see if it is simply a deficiency of a test that never used this field; if it is a padding problem solved by swapping fields or choosing better types; or if this field is obsolete.
z	Indicates a field whose value was always 0 (zero) in every object.
pp	Indicates a pointer: that is, a 64-bit quantity that was a valid pointer into the stack, the static data area, or the heap; or was zero in every object.
ss	Indicates a sometime pointer. This longword looked like a pointer in at least one of the objects, but not in all objects. It could be a pointer that is not initialized in some instances, or a union. However, it could also be the sign of a serious programming error.
i	Indicates a longword that was written with some nonzero value in at least one object and that never contained a pointer value in any object.

Even if an entry is listed as allocating 100MB, it does not mean that at any point in time 100MB of heap storage were used by the allocated objects. It is a cumulative figure; it indicates that this point has allocated 100MB over the lifetime of the program. This 100MB may have been freed, may have leaked, or may still be in the heap. The figure simply indicates that this allocator has been quite active.

Ideally, the fraction of the bytes actually written should always be close to 100%. If it is much lower, some of what is allocated is never used. The common reasons why a low percentage is given include the following:

- A large buffer was allocated, but only a small fraction was ever used.
- Parts of every object of a given type are never used. They may be forgotten fields or padding between real fields resulting from alignment rules in C structures.
- Some objects have been allocated, but never used at all. Sometimes leak detection will find these objects if their pointers are discarded. If they are kept on a free list, however, they will only be found in the heap history.

2.5 Using Third Degree on Programs with Insufficient Symbolic Information

If the executable you instrumented contains too little symbolic information for Third Degree to pinpoint some program locations, Third Degree prints messages in which procedure names or file names or line numbers are unknown. For example:

```
------------------------------------------------------ rus -- 0 --
reading uninitialized stack at byte 40 of 176 in frame of main
    proc_at_0x1200286f0              libc.so
    pc = 0x12004a268                 libc.so
    main                             app
    __start                          app
```

Third Degree tries to print the procedure name in the stack trace, but if the procedure name is missing (because this is a static procedure), Third Degree prints the program counter in the instrumented program. This information enables you to find the location with a debugger. If the program counter is unavailable, Third Degree prints the address of the unnamed procedure.

More frequently, the file name or line number is unavailable because the program's symbol table is incomplete. In this case, Third Degree prints the name of the object in which the procedure was found. This object may be either the main application or a shared library.

If the lack of symbolic information is hampering your debugging, consider recompiling the program with more symbolic information. For C and C++ programs, recompile with the `–g` flag and link without the `–x` flag.

2.6 Validating Third Degree Error Reports

The following spurious errors may occur in rare instances:

- Modifications to bit fields in optimized code are occasionally reported as uses of uninitialized data. This situation usually occurs in initializations of arrays of items smaller than 32 bits or in initializations of packed structures, as in the following example:

```
void Packed() {
    char c[4];
    struct { int a:6; int b:9; int c:4} x;
    c[0] = c[1] = 1;       /* rus errors here ... */
    x.a = x.c = x.e  = 3;  /* ... maybe here */
}
```

- Third Degree initializes newly allocated memory with a special value to detect references to uninitialized variables (see Section 2.3.2). Programs that explicitly store this special value into memory and subsequently read it may cause spurious "reading uninitialized memory" errors.

- Storing the special uninitialized value into memory and subsequently reading it (though the value is neither a valid pointer, a floating-point number, a remarkable integer, nor ASCII characters).

If you think that you have found a false positive, you can verify it by using the disassembler (`dis`) on the procedure in which the error was reported. All errors reported by Third Degree are detected at loads and stores in the application, and the line numbers shown in the error report match those shown in the disassembly output.

2.7 Undetected Errors

Third Degree can fail to detect real errors, such as the following:

- Errors in logical operations on quantities smaller than 32 bits can go undetected, for example:

  ```
  short Small() {
      short x;
      x &= 1;
      return x;
  }
  ```

  This programming practice may be considered an error if the program depends on the least significant bit of *x*. It may not be considered an error if the program depends only on the most significant bits.

- Third Degree cannot detect a chance access of the wrong object in the heap. It can only detect memory accesses from objects. For example, Third Degree cannot determine that `a[last+100]` is the same address as `b[0]`. You can reduce the chances of this happening by altering the amount of padding added to objects. To do this, supply a `third` customization file that includes the `object_padding` option.

- Third Degree may not be able to detect if the application walks past the end of an array by fewer than 8 bytes. Because Third Degree brackets objects in the heap by "guard words," it will miss small array bounds errors. In the stack, adjacent memory is likely to contain local variables, and Third Degree may fail to detect larger bounds errors. For example, issuing a `sprintf` operation to a local buffer that is much too small may be detected, but if the array bounds are only exceeded by a few words and enough local variables surround the array, the error can go undetected.

- Hiding pointers by encoding them or by keeping pointers only to the inside of a heap object will degrade the effectiveness of Third Degree's leak detection.

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-DIGITAL (800-344-4825) before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-bps modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	—————	Local Digital subsidiary or approved distributor
Internal[a]	—————	SSB Order Processing – NQO/V19 *or* U. S. Software Supply Business Digital Equipment Corporation 10 Cotton Road Nashua, NH 03063-1260

[a] For internal orders, you must submit an Internal Software Order Form (EN-01740-07).