Chapter 5

# Software Development Tools

After you've written the source code for your application, you typically compile, debug, and link it. You may also want to optimize the performance of the resulting executable code. This chapter describes the tools you can use to perform these tasks. It tells you how you can

- create executable modules from your source programs
- use IRIX tools to debug, analyze, and optimize your program
- use CASEVision programming tools to debug and optimize your program
- use CASEVision tools to perform version control and configuration management

Refer to Table 2-4 for a list of the manuals you can read to learn more about the topics discussed in this chapter.

## Creating Executable Files

An executable file (or executable) is a program that is ready for execution. To create an executable version of your application, compile your source code and link it with the appropriate libraries. This section briefly describes how to create executables.

### Compiler Drivers

A compiler driver is a program that prepares your application for execution by calling subsystems that compile the source into object code, and then linking together object files, default libraries, and any other libraries you specify. Here are some of the compiler drivers Silicon Graphics provides and the programming language each supports.

**Table 5-1** Compiler Drivers and Source Languages

| Compiler Driver Name | Source Language |
|---|---|
| *f77* | Fortran |
| *cc* | C |
| *pc* | Pascal |
| *CC* | C++ |

When you invoke a compiler driver, specify the name of one or more source files and, optionally, one or more of the compiler driver options. The compiler drivers share a common set of options, although a few options are unique to a single driver or have different meanings for each driver. For example, the *–c* option suppresses the linker step for all drivers, while the *–C* option has one meaning for C and another for Pascal and Fortran. You can use compiler driver options to control these functions:

- program compilation
- program linking
- production of profiling information for performance tuning (see "Performance Tuning Tools")

Read the *MIPS Compiling and Performance Tuning Guide* for more information about compiler drivers. The reference page for each driver contains detailed information about command options for that driver.

## Object Files and Dynamic Linking

Object files generated by Silicon Graphics compilers are in the Executable and Linking Format (ELF). ELF is the format specified by the System V Release 4 Applications Binary Interface (SVR4 ABI).

ELF provides support for dynamic shared objects (DSOs). A DSO is an object file that's shared by multiple applications as they are executing. The object code of a DSO is position-independent and can be mapped into the virtual address of several processes. DSOs are loaded at runtime (instead of at linking time) by the runtime loader, *rld*.

DSOs replace the static shared libraries provided with releases of IRIX prior to IRIX 5.0. You can, and generally should, use them in place of archive libraries. Using DSOs with your application provides you with several benefits. These include the following:

- Overall memory usage is minimized because code is shared.
- Executables linked with DSOs are smaller (and take less disk space) than those linked with unshared libraries because the shared objects are not part of the executable file image.
- Executables containing DSOs don't have to be relinked if the DSO changes.
- DSOs are easier to use and build than the static shared libraries available in releases of IRIX prior to IRIX 5.0.

Most libraries supplied by Silicon Graphics are available as DSOs. When you invoke a compiler driver to build an executable file from your source program, the driver links to DSOs unless you specify otherwise.

You can build your own DSOs if you have the IRIS Developer's Option installed on your system. You don't have to make any changes to your source code to make it part of a DSO. Just use *ld* with the *–shared* option to build the DSO.

## IRIX Tools for Debugging and Tuning Your Program

IRIX provides a number of standard UNIX tools to aid you in debugging and tuning the performance of your program. It also provides performance tools developed at Silicon Graphics.

### Debugging Tool

IRIX provides *dbx*, a source level debugger that allows you to debug C, C++, Fortran 77, Pascal, and assembler code. You can execute a program under *dbx* control to

- examine source code
- examine and change data
- control program execution
- debug machine language code
- debug multiple processes.

You can also use *dbx* to examine a core file, if a program crash occurs, to determine the point at which the crash occurred.

To use *dbx*

- compile the source using an appropriate compiler driver option, usually *–g*. This produces an executable containing symbol table information that is used by *dbx* during program execution.
- execute the program under *dbx* control.

## Tools for Object File Query and Manipulation

Object files generated by Silicon Graphics compilers are in the Executable and Linking Format (ELF), the format specified by the SVR4 ABI. IRIX provides several tools to query and manipulate object files:

| | |
|---|---|
| *dis* | Dissembles an object file into machine instructions. |
| *elfdump* | Lists the contents of an ELF-format object file. |
| *file* | Provides descriptive information on the properties of a file. |
| *nm* | Lists symbol table information. |
| *size* | Prints the size of each section of an object file. |
| *strip* | Removes symbol table and relocation bits from an object file, which saves space after you've debugged your program. |

## Performance Tuning Tools

IRIX provides several tools that you can use to optimize the performance of your application. Table 5-2 summarizes these tools and the paragraphs following the table describe each tool in greater detail.

**Table 5-2** Summary of Performance Tuning Tools

| Name of Tool | Function |
|---|---|
| *prof* | Measures the amount of time spent in various parts of a program—used in conjunction with the *–p* compiler driver option. |
| *pixie* | Counts the number of times basic blocks in a program are executed—used in conjunction with the *prof* program. |
| *par* | Traces system calls and scheduling activity. |
| *cord* | Rearranges procedures in your program to reduce paging and reduce instruction cache mapping. |
| *xscope* | Monitors connections between an X server and a client. |

### Using *prof* and *pixie*

You can use the profiling tools *prof* and *pixie* to find areas of your program where most of the execution time is spent. With this information, you can concentrate your effort on improving code efficiency in these parts of the program. The profiling tools provide two kinds of information:

- Program counter (pc) sampling, which measures the amount of execution time spent in various parts of a program. It does this by interrupting program execution every 10 milliseconds and recording the value of the program counter.
- Basic-block counting, which counts the number of times each basic block executes. A basic block is a region of the program that can be entered only at the beginning and exited only at the end.

To obtain pc sampling data, specify the *–p* compiler driver option when you prepare your program for execution. A program prepared using this option produces pc sampling data during program execution. To view the pc sampling data, run the *prof* program. *prof* analyzes the data files generated during program execution and produces a formatted listing.

Figure 5-1 shows the steps necessary to get pc sampling information.

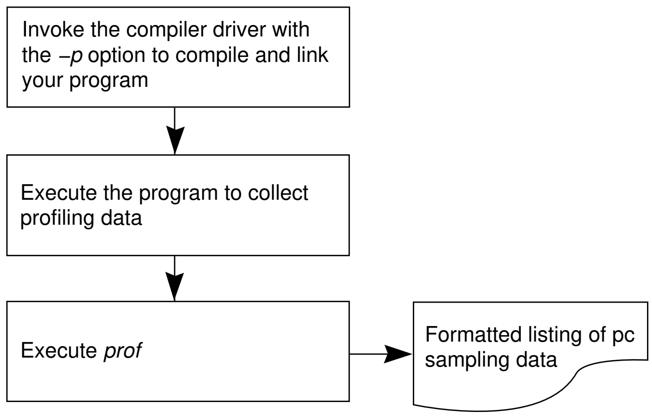


**Figure 5-1** Using *prof* to Obtain pc Sampling Information

To obtain block counting information, use the *pixie* program. *pixie* reads an executable program, partitions it into basic blocks, and writes (instruments) an equivalent program containing additional code that counts the execution of each basic block. You can execute this instrumented program to obtain a file containing basic block counts.

To obtain a formatted listing of the block count information, run the *prof* program using the *–pixie* option. Figure 5-2 shows how you can obtain basic block counts for your program.

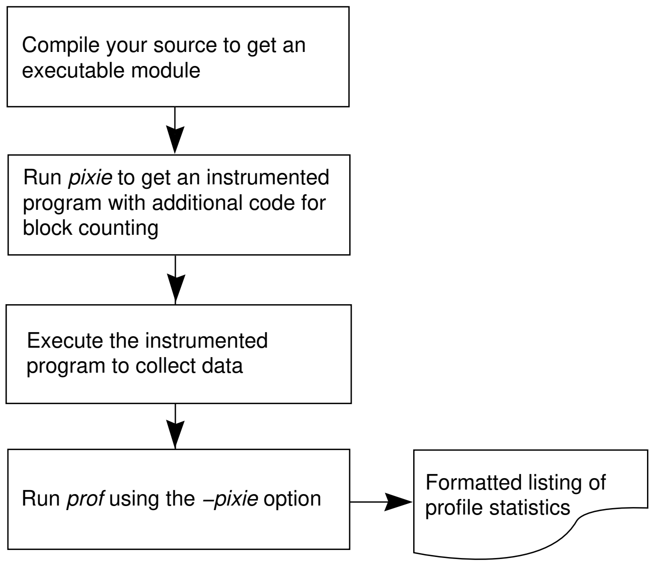


**Figure 5-2** Using *pixie* to Collect Basic Block Counts

### *par*

The *par* system utility program traces system calls and scheduling activity. You can use it to trace the activity of a single process, a related group of processes, or the system as a whole. *par* prints a report showing all system calls made by the specified processes, along with arguments and return values.

### *cord*

The *cord* program rearranges procedures in an executable object to reduce paging and achieve better instruction cache mapping. Typically, the order specified either minimizes paging or maximizes the likelihood that data items are in cache when needed. Figure 5-3 shows how you can use *cord* to rearrange the procedures in your program.

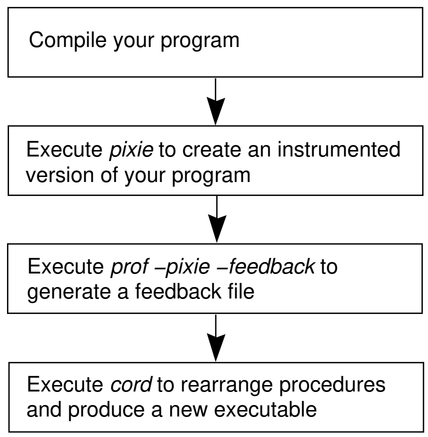


**Figure 5-3** Using *cord* to Rearrange Procedures

### *xscope*

The *xscope* program monitors connections between an X server and a client. It prints the contents of each request, reply, error, or event that is communicated between the server and client. This information can be useful in debugging and performance tuning of X servers and clients.

## The CASEVision Programming Tools

Silicon Graphics provides several interactive, graphical programming tools to aid you in debugging and tuning your application. These computer-aided software engineering (CASE) tools, which comprise the CASEVision product line, operate in a common environment that provides a consistent user interface. "The CASEVision Environment" describes this environment.

The CASEVision tools include:

- CASEVision/WorkShop
- CASEVision/WorkShop MegaDev
- CASEVision/WorkShop Pro MPF

**Note:** The CASEVision/WorkShop tools are not part of IDO—you have to order them separately.

### The CASEVision Environment

All tools in the CASEVision product line share a common environment. Some common facilities and features of this environment are:

- CASEVision tools use standard IRIS IM elements such as the File Browser (used to save and load files) and offer easily set X defaults.
- The CASEVision environment offers a comprehensive online help system with context-sensitive access.
- All CASEVision tools provide access to source code through a common text editor called Source View. Source View provides a window displaying lines of text in a source code file and offers simple text editing features and the ability to fork other text editors such as *vi* or *emacs*.
- Many CASEVision tools provide graphical representations of code, such as function call trees or class hierarchies. These have common features for manipulating the presentation so that you can focus on the data of specific interest or get a larger overview.

### CASEVision/WorkShop

CASEVision/Workshop is a software development environment that helps you visualize your code. It's a set of tools that use an object-oriented application framework and an IRIS IM interface with user-selectable color schemes. The WorkShop toolset includes five graphical tools:

- Debugger
- Static Analyzer
- Performance Analyzer
- Tester
- Build Manager

Figure 5-4 gives you an example of what an interface to a CASEVision tool looks like. This example, output from the Performance Analyzer, shows the total time spent in several routines, and the time each routine actually executed, exclusive of the time spent in called routines.

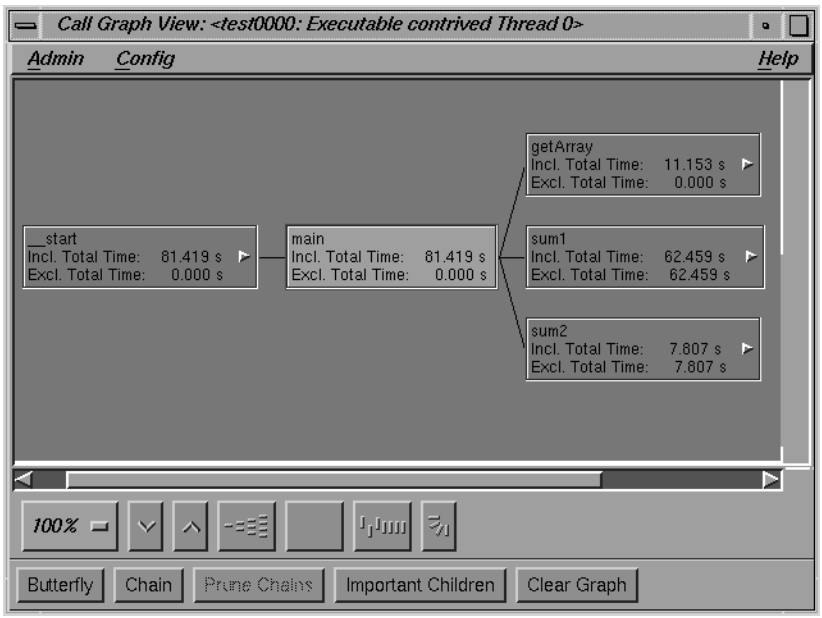


**Figure 5-4** An Example of a CASEVision User Interface

### The WorkShop Debugger

The WorkShop Debugger is a source-level debugging tool that provides special windows (views) for displaying program data and execution status as the program executes. The Debugger lets you set various types of traps (breakpoints) and watch points.

### The WorkShop Static Analyzer

The WorkShop Static Analyzer helps you analyze source code written in C, C++, or Fortran by showing you the code's structure (graphically or in text format). It also shows you how the functions within programs call one another, where and how variables are defined, how files depend on one another, where you can find macros, and many other structural details that help you understand the code.

#### The Performance Analyzer

You can use the Performance Analyzer to define and run experiments that collect performance data. The Performance Analyzer uses this data to produce charts, tables, and annotated code that help you analyze the performance of your program.

#### WorkShop Tester

WorkShop Tester is a software quality assurance toolset for software and test engineers and their managers who are involved in the development, test, and maintenance of long-lived software projects.

#### WorkShop/Build Manager

You can use the WorkShop/Build Manager to compile software without leaving the WorkShop environment. You can look for problems using the WorkShop analysis tools (Static Analyzer, Debugger, and Performance Analyzer), make changes to the source, suspend your testing, and then recompile. The Build Manager has two components:

- Build View—for compiling, viewing compile error lists, and accessing the code containing the errors in Source View (the CASEVision editor) or an editor of your choice. Build View helps you find files containing compile errors so that you can quickly fix them, recompile, and resume testing.
- Build Analyzer—for viewing build dependencies and recompilation requirements and accessing source files.

Build View uses the UNIX *make* facility as its default build software. Although Build Analyzer determines dependencies using *make*, you can use the build software of your choice.

### CASEVision/WorkShop MegaDev

CASEVision/WorkShop MegaDev is a suite of graphical, interactive computer-aided software engineering (CASE) tools designed especially for programmers developing and maintaining C++ libraries and applications.

It contains the C++ Browser and the Fix and Continue utilities. The C++ Browser lets you view the structure of any set of C++ classes. It provides a global, graphical view of interclass relationships such as inheritance, containment, and interaction within a set of classes. The Fix and Continue utilities allow you to redefine functions in your program and then continue execution without recompiling.

### CASEVision/WorkShop Pro MPF

Fortran 77 programmers can use the WorkShop Pro MPF Parallel Analyzer View to view the structure of multiprocessing applications. This tool reads analysis files generated by the Power Fortran Accelerator (PFA) and provides a visual comparison of the original source with the parallelized code.

The function of the Parallel Analyzer View's is integrated with that of CASEVision/WorkShop to allow examination of a program's loops in conjunction with a performance experiment on either a uni- or multiprocessor execution of the program.

## Version Control and Configuration Management

CASEVision provides a tool for creating tracking systems and another for software configuration management.

### CASEVision/Tracker

CASEVision/Tracker is a tool for creating tracking systems for bugs and enhancement requests. It allows you to design and create the tracking database and the interface programs.

The Request Tracking System (RTS) was designed using the CASEVision/Tracker tool. RTS is a system for tracking bugs and requests for enhancements. RTS is designed to:

- meet the basic request tracking needs of most software organizations with only minor modification
- serve as a functioning starter example of a Tracker-based system for those organizations that want to create their own systems

## CASEVision/ClearCase

CASEVision/ClearCase is a software configuration management system that's specifically designed for large-scale, long-lived software projects. ClearCase simultaneously manages multiple versions of evolving software and tracks versions used in software builds. It performs builds and rebuilds of individual programs or entire releases according to user-defined specifications, and enforces project-defined policies.

ClearCase provides enhanced version control of all UNIX (or IRIX) filesystem objects, binary sharing to minimize rebuilds and unnecessary copies or links, and complete build auditing. It also supports parallel builds across the network.