Saturn 4.5 User Guide

Tools > Performance

2007-10-31

Contents

Figures

Overview

After completing the first stages of program development, like design and debugging, you can use performance tools such as Saturn to help optimize your program. Saturn helps you understand your program's function-calling structure and how much time was spent in each function. Saturn consists of two parts: a graphical front-end and a dynamic library back-end. The Saturn back-end library leverages the instrumentation infrastructure in `gcc` to generate an output file that summarizes how much time your program spends in various functions. The Saturn front-end can then read this file and present a representation of the function calling patterns and a function tree view.

UNIX implementations typically support a notion of compiler driven function instrumentation by specifying compiler flags to request instrumentation. The compiler supplied by Apple responds to two different command line options:

- `-pg`— This causes the insertion of call graph profile data generating code into every function prologue during compilation. Each execution of your program produces a `gmon.out` file. The standard tool for reviewing these files is the `gprof` command-line tool, but Saturn's front end can display the same information in a graphical manner on PowerPC-based Macs only.
- `-finstrument-functions`— This causes the insertion of separate user-defined instrumentation routines for each function prologue *and* epilogue. It incurs somewhat more overhead than `-pg`, but allows you to record a selection of custom data items in addition to basic timing. This option works with both PowerPC *and* Intel-based Macs.

Saturn allows you to visualize the data in two ways: a traditional call-tree view and a graphical call-stack timeline. Using this information, you can eliminate expensive calling behavior (for example, deep call stacks which do not last long) as well as understand which functions take up the greatest portion of execution time.

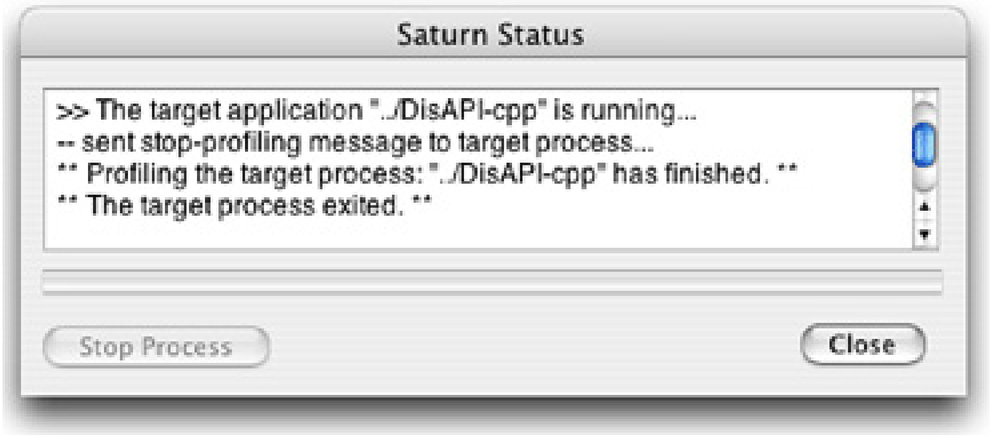
The Saturn Front-end

The Saturn front-end is the application used to visualize the information recorded in an output file containing profiles generated by the Saturn back-end. Because these profiles are deterministic, with *every* function entry and exit recorded, Saturn is able to precisely display the selection of functions . This can be seen in the graphical visualization in the front-end document window(Figure 1-4 (page 14)). You can also use the Saturn front-end to launch your programs and, optionally, to set up what performance events you want to monitor.

The Status Panel

The Saturn front-end features a persistent status panel that provides prompts and other messages indicating what the front-end is doing or about to do, or whether a procedure has failed and why. It is always available (using the *Window StatusPanel* item, or by pressing *Command-1*). If things do not seem to going smoothly, then you should check the status panel for messages from Saturn.

Figure 1-1 Status Panel



```
>> The target application "../DisAPI-cpp" is running...
-- sent stop-profiling message to target process...
** Profiling the target process: "../DisAPI-cpp" has finished. **
** The target process exited. **
```

The Saturn status window has four controls:

1. **Message View**— This area lists messages about the operation of Saturn. Use the scroll bar at the right side to look at older messages, if necessary.

2. **Progress bar**— This displays a colored bar graph that fills with blue from left to right, with progress reflecting the percentage of the task completed, as Saturn performs any command, such as loading a profile.

3. **Stop Process Button**— This button is enabled only while a child process, launched using the *Launch Panel* (see "Launching Processes for Profiling" (page 10)), is running. Pressing it causes the process to finish writing its Saturn data to disk and then exit.

4. **Close Button**— This dismisses the Saturn status panel. You normally will not need to use this, unless you have limited screen area.

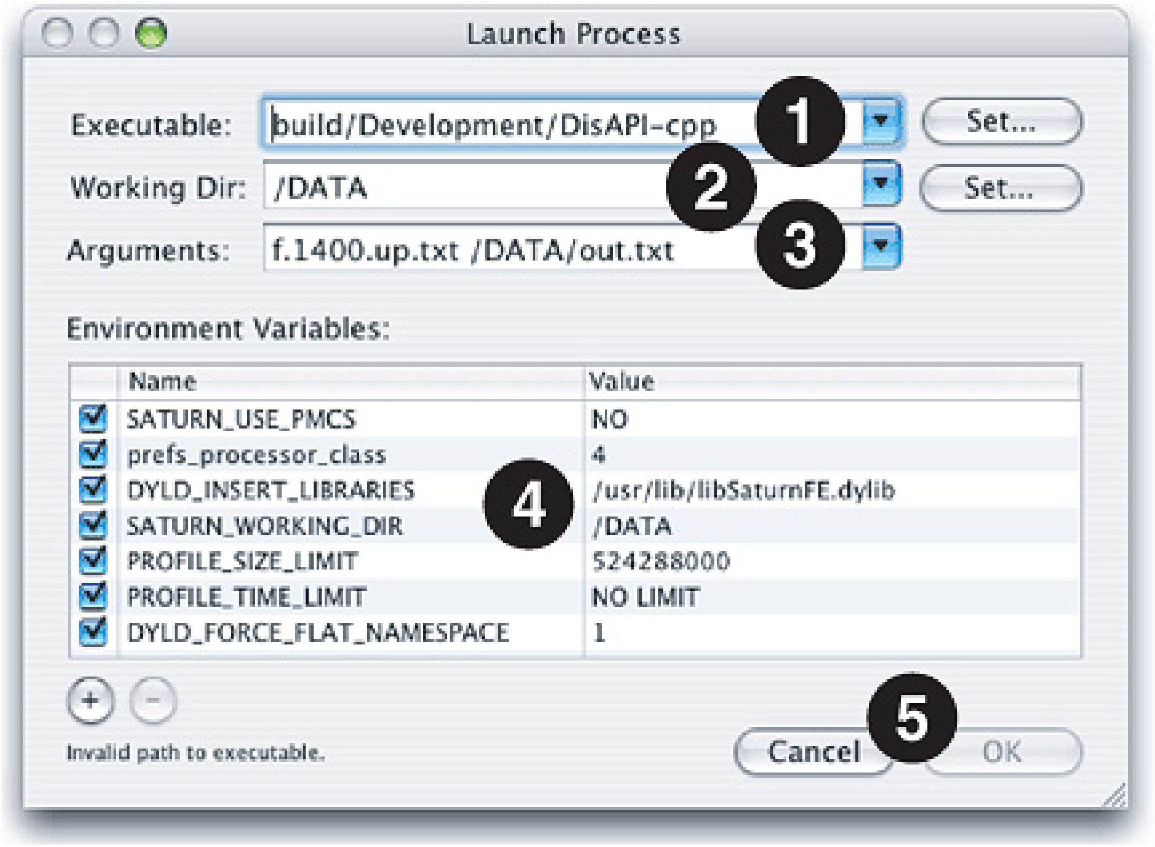
Controlling the Back-End

The Saturn front-end can help you control applications that you are profiling using the Saturn back-end. In particular, it lets you launch applications with all of the correct environment settings already set. After profiling, it can automatically find and open the resulting profiles.

Launching Processes for Profiling

Saturn can help you launch your applications with proper settings for recording profiling data. It does this by providing an interface for supplying parameters and automatically presetting several key environment variables for you. You need to have previously compiled your source code with function profiling by specifying the `-finstrument-functions` option at compile time. Alternatively, on PowerPC Macs only, you may use the `-pg` option, instead. In this case, Saturn *must* be used to launch your process, because it needs to override the profiling functions inserted by `gcc` with its own code. See "The Saturn Back-end" (page 19) for more information.

Figure 1-2 Launch Process Panel



Executable: build/Development/DisAPI-cpp
Working Dir: /DATA
Arguments: f.1400.up.txt /DATA/out.txt

Environment Variables:

	Name	Value
✓	SATURN_USE_PMCS	NO
✓	prefs_processor_class	4
✓	DYLD_INSERT_LIBRARIES	/usr/lib/libSaturnFE.dylib
✓	SATURN_WORKING_DIR	/DATA
✓	PROFILE_SIZE_LIMIT	524288000
✓	PROFILE_TIME_LIMIT	NO LIMIT
✓	DYLD_FORCE_FLAT_NAMESPACE	1

Invalid path to executable.

To launch an application from Saturn, use the *Saturn□Launch Process…* menu item (or *Command-L*). This will bring up the “Launch Process” Panel, illustrated in Figure 1-2, where you need to “fill in the blanks” in order to tell Saturn how to launch your application. Several default environment variables used by Saturn’s profiling library will already be completed for you. Beyond this, you must at least choose an executable file before pressing “OK” and continuing. However, you can also supply many other bits of information to Shark in order to simulate the launching of your application from a command-line shell prompt, and specify a couple of options to help limit capture of spurious samples.

1. **Executable**— The *full* path to the executable. You can either type it here or press the “Set...” button and then find it using a normal Mac “Open File...” dialog box. For Mac applications, you can set this to be either the entire application or the core binary file inside of it.

2. **Working Dir**— The *full* path to the working directory that the application will start using. By default, this is the path where the executable is located, but you may point it anywhere else that you like, either by typing a path or pressing the “Set...” button and using the resulting “Open File...” dialog box to choose a directory. When the application is executed, it will appear to have been started from a shell that had this directory as its working directory (i.e. the output of `pwd`) just before executing the command. Hence, relative paths to data files will be resolved starting from this directory. Folders and files containing Saturn profiling results will also be placed in this directory.

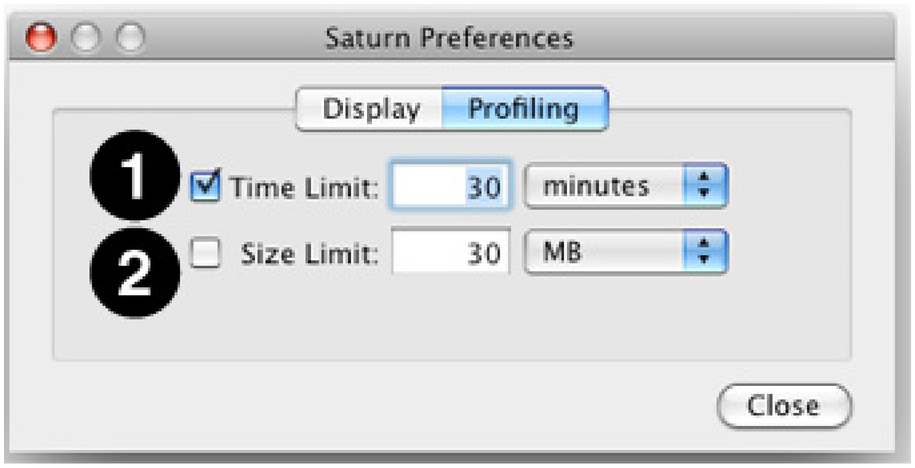
3. **Arguments**— Enter any arguments here that you would have normally entered onto your shell command line after the name of the executable. Saturn will feed them into the application just as if they had come from a normal shell. Note that since Saturn’s “shell” does not have any text I/O, you will need to provide `< stdin.txt` and `> stdout.txt` redirection operations here if your executable expects to use `stdin` and/or `stdout` “files.”

4. **Environment Variables**— Supply *any* environment variables that must be set before your application starts in this table. Otherwise, Saturn will start your application with *no* environment variables set other than the few it uses to control profiling (it even clears out any “defaults” that you may have had before invoking Saturn). Pressing the “+” and “–” buttons below the table allows you to add or remove environment variables, respectively. Once added, you can freely edit the names of the variables and their value in the appropriate table cells. In general, you should not touch any of the default variables that Saturn supplies for you. However, it is possible to control the `PROFILE_SIZE_LIMIT` and `PROFILE_TIME_LIMIT` variables using the Saturn “Profiling Preferences” (page 12). These settings help prevent runaway profiling from filling up your hard disk space with a gigantic Saturn data file. By default, the file size is limited to 30 MB and the profiling time to 30 minutes, whichever comes first.

5. **Cancel & OK**— Saturn will start your application and sample it *immediately* after you hit OK. If you change your mind, Cancel will leave without starting any child processes.

Profiling Preferences

Figure 1-3 Preference Panel – Profiling Tab



[✓] Time Limit: 30 minutes

[ ] Size Limit: 30 MB

Saturn's preference panel, opened using the menu item *Saturn▸Preferences…* menu item (or *Command-,*), contains two tabs. The second, "Profiling" (illustrated in Figure 1-3) lets you control the `PROFILE_SIZE_LIMIT` and `PROFILE_TIME_LIMIT` environment variables that Saturn passes to profiled applications when executing them. Each variable is controlled independently:

1. **Time Limit**— The first row of controls sets the `PROFILE_TIME_LIMIT` environment variable. If the checkbox is not set, then a value of `NO LIMIT` is used, and no time limit will be imposed on profiling. Otherwise, Saturn passes the time value expressed in the center edit box and pop-up menu on the right. You can specify the time in seconds, minutes, or hours. Because the `PROFILE_TIME_LIMIT` variable is expressed in seconds, Saturn will automatically convert the values you provide into seconds before setting the variable.

2. **Size Limit**— The second row of controls sets the `PROFILE_SIZE_LIMIT` environment variable. If the checkbox is not set, then a value of `NO LIMIT` is used, and the profile output file size will only be limited by the size of your hard disk. If you are writing the output to your startup disk, this can cause serious problems for your Mac. Otherwise, Saturn passes the size value entered into the center edit box and pop-up menu on the right. You can specify the size in kilobytes (KB), megabytes (MB), or gigabytes (GB). Because the `PROFILE_SIZE_LIMIT` variable is expressed in bytes, Saturn will automatically multiply the values you provide by the appropriate conversion factors before setting the variable.

Note regarding file size: A file size between 500MB and 1GB might not open in Saturn, because opening it may cause Saturn to exceed the available virtual memory allowed for one 32-bit program in OS X (currently 2.5 GB). Whether the expanded data will exceed the virtual memory limit is totally dependent on the number of unique functions accessed. A very large file generated by a program with only a handful of functions will open just fine, while a large file generated by a program with hundreds (or thousands) of functions will often be problematic.

Displaying Results

After you record a profile, Saturn allows you to examine it using an interactive, graphical profile browser by choosing the *File > Open…* menu item (or *Command-O*) and then selecting the profile data file produced during your sample execution with the resulting standard "Open File..." dialog box. You may also see this result in a more automated fashion after executing your program from within Saturn. Saturn's graphical interface lets you see the same results that are presented textually in `gprof` output, but in a dynamic way that can be flexibly sorted and rearranged in order to allow you to find the most interesting parts of the profile more easily.

What Saturn Shows You

Saturn is most useful as a tool for understanding the function call behavior of your code. Using Saturn it is possible to understand four types of calling characteristics:

1. **Call Count**— Call count is useful as a sanity check for your program's behavior. You should ensure that the profile call count matches your expectations. Each function call incurs a fixed calling overhead, so making a large number of calls to a short function that performs very little useful work per call is inefficient. Hence, functions with large call counts and short execution times are good candidates for inlining.

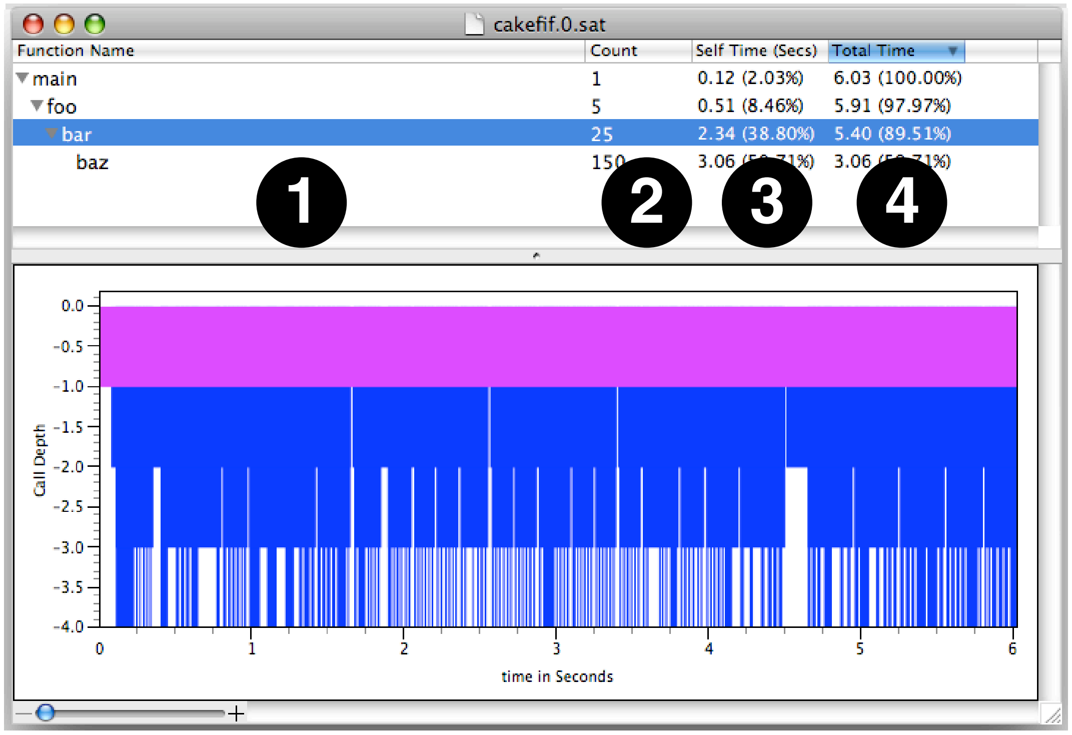
2. **Call Time**— In general, an application spends a large fraction of its time in a small fraction of its code. Therefore, the most efficient way to utilize limited programmer time is to concentrate on optimizing those functions that take up the largest portion of execution time.

3. **Call Depth**— Call depth is directly proportional to the amount of calling overhead your program incurs in order to get to the function(s) that do the actual work. Modern programming principles such as layering, abstraction and polymorphism lighten the burden of programming and code maintenance, but often at the expense of calling overhead and obfuscated calling behavior. Deep call stacks that terminate in functions with short call tenure suffer from an inordinate amount of calling overhead. In these cases, it is often beneficial to inline all or some of the functions in the callstack in order to reduce this overhead.

The Profile Window

When you open a profile data file, Saturn presents the profiled information in a single, unified window. After a delay that can be up to several minutes for profiles with a high function call count, a window summarizing the profile results will appear (see Figure 1-4 for a sample). Since Saturn profile data files can be very large, you will probably want to keep an eye on the status panel progress bar and message view during the process of loading the file, in order to make sure that no problems occur.

Figure 1-4 Profile Window



cakefif.0.sat

Function Name	Count	Self Time (Secs)	Total Time
main	1	0.12 (2.03%)	6.03 (100.00%)
foo	5	0.51 (8.46%)	5.91 (97.97%)
bar	25	2.34 (38.80%)	5.40 (89.51%)
baz	150	3.06 (50.71%)	3.06 (50.71%)

The upper half of a Saturn profile window shows the top of *Call Tree* view of profiled functions. Each row of the table represents one function in the call tree, and presents key profiling information about that function. This view is useful for understanding the average behavior of your program, such as which functions were executing most often and how often they were called. This information is often useful because you can use it to quickly focus in on the functions in your program that are executing most often or being called most frequently.

The columns of the call tree view from left to right are:

1. **Function Name**— The name of the function, as listed in the process' symbol table, associated with the data on this row of the table. Except for leaf functions (ones that do not call others), there is a disclosure triangle next to each function name. Click the triangle next to a function of interest in order to reveal all child functions called by it on one or more new rows. Simply click it again to hide the child functions again.

2. **Count**— The number of times this function was called.

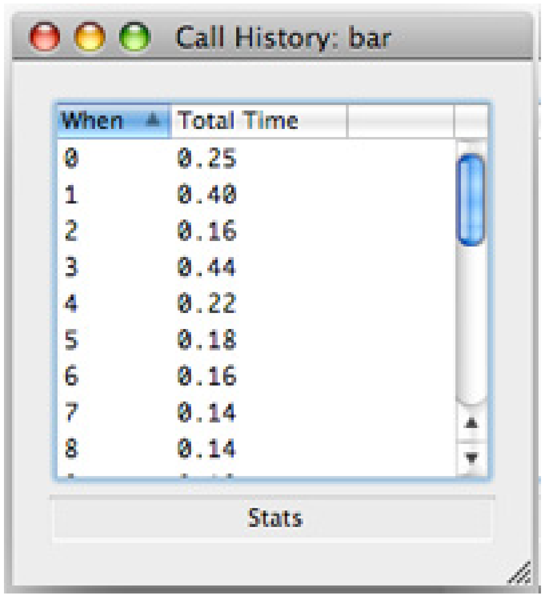
3. **Self Time**— The amount of time spent executing this function's own code (but not including time spent in its children).

4. **Total Time**— The total amount of time spent executing this function *and* its children (other functions called by this function).

You may click on the any of the column headers to have Saturn sort the profile results of *sibling* functions by the information in that column. The order of functions at different levels or on different branches of the call tree will not be affected by this, because children are always listed directly below their parents and the group of sibling functions at each branch/level of the tree is only sorted within that group. Clicking on the small arrow at the right end of the selected column header allows you to choose an ascending or descending sort order. Usually, sorting by "total" or "self" time is most useful, but sometimes it can be helpful to sort based on call counts, if you are trying to find small functions that are called an extraordinarily large number of times.

Double-clicking on any row of the table opens up an inspector window for that function. The inspector window lists all of the individual call times (see Figure 1-5) for the given function. For non-leaf functions, both self and total values are displayed.

Figure 1-5 Function Inspector Window



When	Total Time
0	0.25
1	0.40
2	0.16
3	0.44
4	0.22
5	0.18
6	0.16
7	0.14
8	0.14

The lower half of the window is a *Chart* view that shows the depth of your program's callstack (plotted vertically) over time (plotted horizontally). Callstack levels are indicated by differing shades of blue, which cycles vertically. Each function's tenure, or time that it was executing, is depicted by the width (horizontal size) of its area in the callstack graph. Look at this view to explore your profile data chronologically. This can help you understand the sequence of calls used in your program, as opposed to the overall summary calling behavior shown in the *Call Tree View*. Using this chart, you can see at a glance if your program rarely/often calls functions and if there are any recurring patterns in the way your program calls functions. Based on this, you can often visually see different *phases* of execution — areas where your program is executing different pieces of its code in different ways. This information is useful, because each phase of execution will usually need to be optimized in a different way.

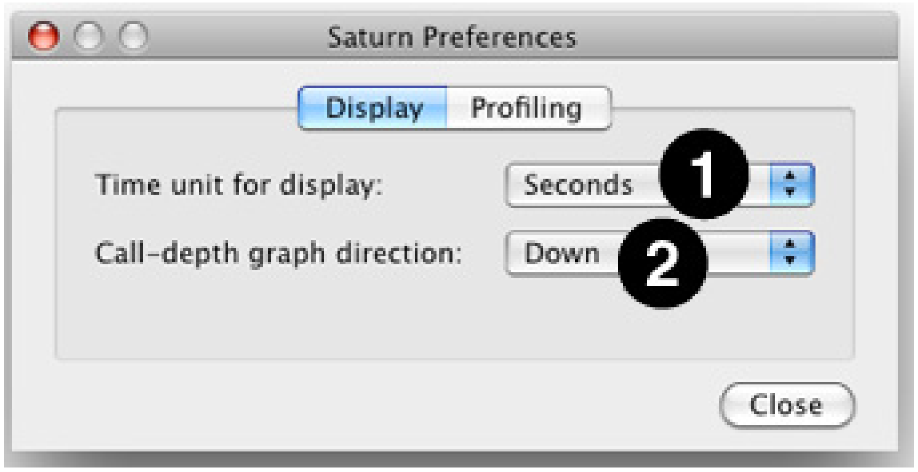
The *Chart* view is very dynamic. You can select a function's execution by clicking on it in the graph. This action also selects the function in the *Call Tree* view. Similarly, selecting a function in the *Call Tree* view highlights all of the tenures of that function in the *Chart* view. If the area of interest on the graph is too small for you to see, then you can simply drag your mouse (click and hold) over the desired rectangular area to have Saturn zoom into and magnify that area to fill the whole view. When you wish to return back to seeing the entire chart, zooming out is accomplished by *Option-Clicking* anywhere in the chart area.

Note on early-exit functions: If Saturn encounters function call enter without a corresponding exit, usually as a premature exit from a function such as a C++ exception, the function timeline entries for that call sequence will be colored lavender instead of blue, and will highlight in yellow instead of red.

Note on generic gmon.out data files: Opening generic `gmon.out` files created when you start binaries compiled with `-pg` by any means other than the Saturn program launcher will usually cause unexpected results. Opening `gmon.out` files produced on an Intel Mac will also fail. Instead, this feature only works reliably when you launch your application from within Saturn on a PowerPC-based Mac. Even in this case, however, the *Chart* view is not displayed, because it requires information that is not captured during `-pg` profiling. If you need to see the *Chart* view or are running on an Intel Mac, then you will need to use the `-finstrument-functions` option.

Display Preferences

Figure 1-6 Preference Panel – Display Tab



Time unit for display: Seconds

Call-depth graph direction: Down

Saturn's preference panel, opened using the menu item *Saturn□Preferences…* menu item (or *Command-,*), contains two tabs. The first, "Display" (illustrated in Figure 1-6) lets you control two options regarding the display in each profile window. Both options can also be controlled using menu items in Saturn, if desired. The options are:

1. **Time unit for display**— The first pop-up menu lets you choose the time units for displaying results in the self/total time columns of the *Call Tree* view here. You can choose between seconds, milliseconds, microseconds, and nanoseconds. This can also be set using the various commands in Saturn's *Time* menu.

2. **Call-depth graph direction**— The second pop-up menu lets you choose to have the graph on Saturn's *Chart* view grow down from the top (the default) or up from the bottom. This is purely a matter of personal preference, as the same information is presented either way. These options may also be chosen using Saturn's *Graph□ Stack Grows□* submenu.

Path Menu

This menu allows you to control how much information Saturn displays, by allowing you to focus its display on particular parts of your trace. In very long or complex traces, this can be very helpful. The menu contains three options:

1. **Follow Longest Time Path**— This reveals the callstack pattern that has the most time attributed to it, either through one or many calls. Because the function executing most often will often be the first place you will want to consider optimizing your programs, this is a good command to use when first examining a profile.

2. **Prune Path**— This will hide all functions outside of the function you are currently examining and any child functions that it calls. All parent functions in the callstack and functions that this function or its children never call are simply eliminated from the display. This allows you to focus on only the effects of this function and anything that it calls.

3. **Restore Root**— This restores any pruned-off paths, returning you back to the view of the entire Saturn trace.

The Saturn Back-end

The Saturn back-end is a dynamic library that is *linked* with the application under study. It is invoked by your application during function prologues and epilogues in order to produce the profile traces used by the Saturn front-end application.

After the back-end is linked with your application, it will automatically produce one or more Saturn profile files in the active each time your application is executed. The output file name format is: "{app name}`.`{thread #}`.sat`". Because each execution of your application can create many different profile files, one for each thread, Saturn creates a new directory in your working directory and names it "`Saturn_profile_`{app name}_{run number}" to collect all of the separate profiles together. In this name, the run number starts at 0 and is incremented each time you run your program, if you take several profiles in succession.

Most of the time, this library will work for you with very little intervention. However, in case of problems or if you desire more precise control over your profiling then you may need to control it more directly. This chapter discusses some issues that may occur when linking the back-end to your application and some ways that you can control the back-end to get more control over how it profiles your application.

Linking the Back-end to Your Application

As mentioned in the "Overview" (page 7), there are two ways to use the back-end to generate the data files for the front-end:

- `-finstrument-functions`— Saturn typically takes advantage of the infrastructure that GCC provides through the `-finstrument-functions` option. This option works with both PowerPC *and* Intel-based Macs. You can type this option right into your GCC command lines. With Xcode, you will need to enter it as a string into the "Other C Flags" line in your target's build settings, because Xcode does not have pre-made setting that you can check for this option.

  When program source is compiled by GCC with this flag, GCC adds a **prologue** function call at each routine entry point, in each module compiled. GCC also adds an **epilogue** function call to each routine in every module, before the routine returns. The prologue and epilogue functions that Saturn attaches to your application emit an entry for a Saturn document file (`*.sat` format). These two functions are included in the Saturn back-end dynamic library called `libSaturn.dylib`. To link to the library, either add the file `/usr/lib/libSaturn.dylib` to your Xcode project (when adding this "framework," you will need to use *Command-Shift-G* in the *Open file...* dialog in order to access the `/usr/lib` folder, which is otherwise hidden) or add `-lSaturn` to the link command in a makefile.

- `-pg`— This profiling option, which works on PowerPC-based Macs only, causes `gcc` to use its *gprof* profiling technique. Saturn can read the resulting `gmon.out` files and display the profiling results in a manner similar to the command-line `gprof` tool.

  When using this method, you do not have to make any other modifications to your source code or compiler/linker options. The whole program will be instrumented with just the `-pg` option. You also do not have to link against any special libraries.

Important: While you do not need to specify additional compiler or linker options, you *must* launch your program using the Saturn front-end. The Saturn front-end will overload the profiling functions inserted by `gcc` to collect the necessary function call graph data. If you open a `gmon.out` file created without starting the application from within Saturn, unpredictable results may occur.

Apple supplies libraries already compiled with the `-pg` option enabled, so they may be profiled when linked with your `-pg` binary. However, by default libraries without `-pg` are used. To select the profiling versions of the libraries, set the environment variable `DYLD_IMAGE_SUFFIX` to "`_profile`" before running your application.

So which option should you use? In general, the `-finstrument-functions` option is the better choice. While it can incur somewhat more overhead than the `-pg` one, it allows Saturn to record more types of information. To be precise, it captures more callstack timing information, which allows Saturn to display its full callstack graph. Without this information, Saturn can present only summary statistics. Moreover, if you are on an Intel-based Mac, this is the only option that you can use.

Important: It is very important that the program to be profiled has been compiled and linked with **symbol information**. Otherwise, the *Call Tree* view will contain only the hex addresses of the functions that were called; the Saturn front end will not be able to resolve those addresses with the function names that it expects to find in the executable image. The common compiler option `-g` is sufficient to provide the necessary symbol information in an executable. However, even if you specify this option, you must be careful with linker options, because it is common practice to have the linker strip out symbol information during the link phase. Check for enabling of any of these linker options that might strip out symbol information if you have added `-g` and still see no function names in the Saturn *Call Tree* view.

Note on alloca(): When using the `-finstrument-functions` instrumentation technique, Saturn profiling is currently unable to profile applications that use `alloca()` with *runtime calculated* allocation size, although ones with sizes fixed at compile time do work. This is a known problem. The current workaround is to compile your code with the `-mdynamic-nopic` flag.

Environment Variable Controls

If you link with the Saturn dynamic library, you can set the following environment variables in your shell, before starting your application, in order to easily control several options:

- `SATURN_WORKING_DIR`— Set this to the path of the directory where you want to store the Saturn data files. Absolute paths are generally best; relative paths will be resolved relative to the current working directory of the shell that invokes the application being profiled.
- `PROFILE_SIZE_LIMIT`— This parameter, when supplied, forces the back-end to limit the size of the data files produced to a fixed limit, expressed here in bytes. If this limit is reached, profiling stops immediately. It is a good way to make sure that your application does not accidentally fill up your entire disk with profile data, a factor that is especially important if you are writing profile data to your Mac OS X startup disk.
- `PROFILE_TIME_LIMIT`— This parameter, when supplied, forces the back-end to limit the time that it records profiling information to a fixed time limit, expressed here in seconds. If this limit is reached, profiling stops immediately. This option is an easy way to terminate profiling early for long-running

applications. (For more precise termination control, however, see "Programmatic Start/Stop Control" (page 21), instead.) In addition, like the previous `PROFILE_SIZE_LIMIT` variable, it can be used to prevent your application from filling up your entire disk with profile data. However, because it provides no file size guarantees, `PROFILE_SIZE_LIMIT` is generally recommended over this.

Programmatic Start/Stop Control

By default, Saturn profiles your entire application, from beginning to end, in all threads of execution. However, in real programs you often only want to profile a subset of your application's execution at any one time, in order to focus your analysis on one part of the application or another. To allow you to exercise this level of control, the Saturn back-end contains three other functions that you can use directly in your programs: `initSaturn()`, `startSaturn()`, and `stopSaturn()`. These functions control generation of output data in the back-end dynamic library, enabling profiling for a thread at each `startSaturn()` call and disabling it at each `stopSaturn()` call.

Normally, these functions are used in a fairly simple pattern. To explicitly initialize the Saturn back-end for tracing your program, add the call: `initSaturn(char *path)`. This acts much like the automatic initialization, but allows you to explicitly specify the name of your Saturn profile. Once tracing has been initialized, you can then start and stop tracing using pairs of `startSaturn` and `stopSaturn` calls.

Note: Even if you have let the initialization of Saturn occur automatically, you can still add a call to `stopSaturn()` in your code to end the profiling before the natural end of the program. This is the only way that you should ever use mismatched `startSaturn` or `stopSaturn` calls.

If your program creates any child threads over the course of its execution, they inherit the profiling status of their parent, by default. Hence these explicit start and stop calls are also a good way to limit profiling to a limited number of threads of interest in a heavily multithreaded program.

API Reference

This section gives a brief reference for the three routines you can use to control Saturn directly.

- **initSaturn**—

  ```
  void initSaturn(char *path);
  ```

  This must be called before any `startSaturn` or `stopSaturn` calls are made. It initializes Saturn with the following parameter:

  path— The path of the directory to write output files. This can be a full path or a path relative to the current working directory. A `NULL` or an empty string argument will cause the files to be written in the current working directory.

- **startSaturn**—

  ```
  void startSaturn(void);
  ```

  Begins Saturn profiling. Data for every function that is called after this will be stored in the appropriate data file for the thread that is executing the call.

- **stopSaturn**—

```
void stopSaturn(void);
```

Ends Saturn profiling. Data collection ceases for this thread until `startSaturn` is called again. There can be many `startSaturn/stopSaturn` function call pairs throughout an instrumented application, if you want to see many small segments of execution scattered in different areas.

Note on Headers: Include the header file: "`<Saturn.h>`" in source code modules that will be making calls to any of these API routines.

A Programmatic Example

Below is a basic example using these functions. It shows how you can use a single pair of `startSaturn` and `stopSaturn` calls to exclude initialization and shutdown code from your profile. This is a common practice with programs that spend minimal amounts of time in these routines, in order to avoid polluting the profile with the large number of relatively unimportant function calls often found there. Please note that there can be many `startSaturn/stopSaturn` function call pairs throughout an instrumented application, if you want to focus even further on small segments of execution scattered in different areas.

```
#include <Saturn.h>
int main()
{
 // Initialize Saturn:
 // The output file will put in the current working directory.
  initSaturn ("");

 // Do work that you don't want to measure, like initialization
  . . . init code here . . .

  // Start the back-end.
  startSaturn ();

  // This portion of the program is measured
  . . . program core here . . .

  // Stop the back-end.
  stopSaturn ();

  // This isn't measured, again
  . . . shutdown core here . . .
}
```

Document Revision History

This table describes the changes to *Saturn 4.5 User Guide*.

Date	Notes
2007-10-31	New document that explains how to analyze a program's function-calling structure.