Thread Analyzer User's Guide

SunSoft
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No.: 801-6691-10
Revision A, August 1994

© 1994 Sun Microsystems, Inc.
2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

All rights reserved. This product and related documentation are protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or related documentation may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Portions of this product may be derived from the UNIX® and Berkeley 4.3 BSD systems, licensed from UNIX System Laboratories, Inc., a wholly owned subsidiary of Novell, Inc., and the University of California, respectively. Third-party font software in this product is protected by copyright and licensed from Sun's font suppliers.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the United States Government is subject to the restrictions set forth in DFARS 252.227-7013 (c)(1)(ii) and FAR 52.227-19.

The product described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

TRADEMARKS
Sun, the Sun logo, Sun Microsystems, Sun Microsystems Computer Corporation, Solaris, are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and certain other countries. UNIX is a registered trademark of Novell, Inc., in the United States and other countries; X/Open Company, Ltd., is the exclusive licensor of such trademark. OPEN LOOK® is a registered trademark of Novell, Inc. PostScript and Display PostScript are trademarks of Adobe Systems, Inc. All other product names mentioned herein are the trademarks of their respective owners.

All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. SPARCstation, SPARCserver, SPARCengine, SPARCstorage, SPARCware, SPARCcenter, SPARCclassic, SPARCcluster, SPARCdesign, SPARC811, SPARCprinter, UltraSPARC, microSPARC, SPARCworks, and SPARCompiler are licensed exclusively to Sun Microsystems, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOKand Sun™ Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

X Window System is a product of the Massachusetts Institute of Technology.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME

Please
Recycle

Adobe PostScript

Contents

1. `tha`**—Thread Analyzer** **1**

Getting Started 1

Compiling and Instrumenting the Source 1

Running an Experiment 2

Running `tha` 2

Starting `tha` 2

Loading a Trace Directory 5

`tha` Glyph Hierarchy 6

Navigating through the `tha` Glyph Hierarchy 6

Interactive Error Messages 7

Exiting from `tha` 8

Metrics Collected by `tha` 8

`tha` Menus 9

Main Menu Buttons 10

Load Button 10

View Button 10
2. Using tha 15
Scenario 1: Initial Investigation — CPU Time Tabular Data. . . 15
Step 1: Display the Thread Level Nodes. 15
Step 2: Request a `prof` Table for All Threads 16
Step 3: Analyze the `prof` Table Data 17
Step 4: Focus on a Particular Thread. 17
Step 5: Request a `gprof` Table for a Particular Thread. . . . 18
Step 6: Request a Metric Table for a Particular Function . . 19
Step 7: Request a Sorted Metric ProfileTable for All Threads 19
Scenario 2: Initial Bottleneck Identification via Graphical Data 21
Step 1: Select the Node of Interest 21
Step 2: Designate the Metric. 21
Scenario 3: Focused Bottleneck Identification via Graphical Data 23
Step 1: Select the Node of Interest 23
Step 2: Designate the Metric. 23
Scenario 4: CPU Time Filter 25
Step 1: Display the Thread Level Nodes. 25
Step 2: Specify the Filter Value. 25

Preface

The *Thread Analyzer User's Guide* describes how to instrument multithreaded C, C++ and FORTRAN programs and how to use the thread analyzer to gather performance data about the instrumented program.

Who Should Use This Book

This manual is intended for C, C++ and Fortran MP programmers who are developing or tuning multithreaded programs.

How This Book Is Organized

The *Thread Analyzer User's Guide* has the following organization:

Chapter 1, "tha—Thread Analyzer," describes how to instrument your program, the metrics collected by thread analyzer, and how to navigate through the graphical user interface.

Chapter 2, "Using tha," presents examples of how to use the thread analyzer to determine what your program is doing and see where bottlenecks are present.

What Typographic Changes and Symbols Mean

The following table describes the type changes and symbols used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `system% You have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`system% su` `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.
Code samples are included in boxes and may display the following:		
`%`	UNIX C shell prompt	`system%`
`$`	UNIX Bourne and Korn shell prompt	`system$`
`#`	Superuser prompt, all shells	`system#`

`tha`—*Thread Analyzer* 1

The thread analyzer, `tha`, is a graphical user interface for viewing trace information. It displays `gprof` and `prof` tables on a per-thread basis.

`tha` displays:

- tables of metrics
- graphs of these metrics.

This chapter describes what you need to do to prepare your program for analysis, how to navigate through `tha` glyph hierarchy, and how to use its menu-driven interface.

Getting Started
Running tha
Metrics Collected by tha
tha Menus

Getting Started

Compiling and Instrumenting the Source

Compile your program with the `-Ztha` option. The `-Ztha` option instruments your program; that is, it inserts instrumentation points at the beginning and end of each function.

1

Note – If an instrument program calls `fork`, the forked process does not contribute trace data until it calls `exec`.

The `-Ztha` option instruments C++ and FORTRAN programs as well as C programs. For C, the syntax looks like:

```
cc -Ztha -o prog1 prog1.c
```

Running an Experiment

Run the object file to write trace data to files in a `tha`.*pid* directory, where *pid* is the unique process id of the particular invocation of the object.

Caution – Do not use ctrl-C to stop a running instrumented program. ctrl-C leaves the trace directory in an inconsistent state. A subsequent invocation of `tha` on that trace directory cannot process its trace data.

Running `tha`

Starting `tha`

To analyze the trace data collected for your program, run `tha` from the command line.

```
tha &
```

Optionally, you can direct `tha` to load a `tha`.*pid* trace data directory via the command line.

```
tha tha.1234 &
```

Note – `tha` expects to find the executable in the same directory as the trace directory. There are known problems related to correctly finding the executable associated with a trace directory. If you encounter problems in this area, use the `-exec` command line option to specify the pathname to the executable.

You can use the `-exec` option to specify a relative pathname or full pathname to the executable to be used in conjunction with a particular trace data directory.

The next examples show two variations of specifying a full pathname for the executable:

```
tha -exec /bin/ls tha.1234 &
```

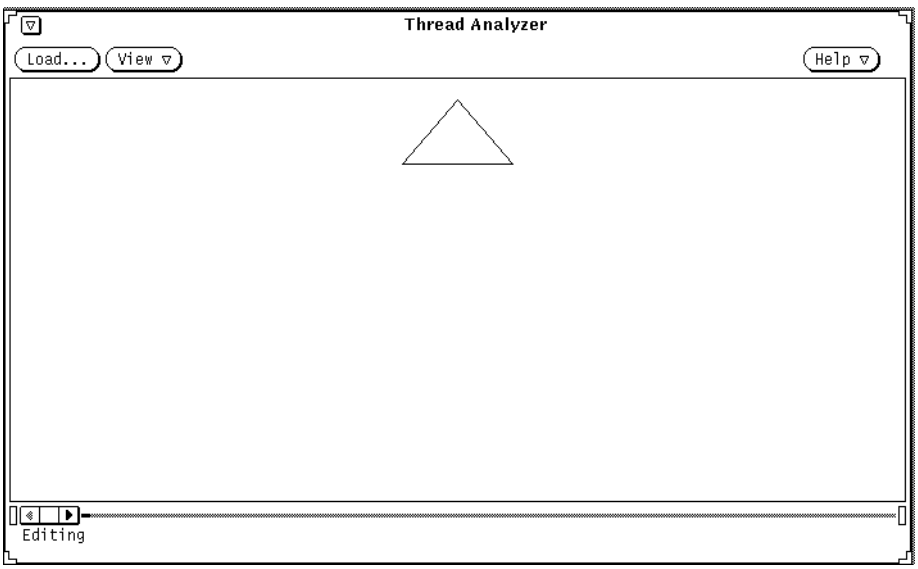
or

```
tha tha.1234 -exec /bin/ls &
```

The following example shows how to specify a relative pathname:

```
tha -exec ls tha.1234 &
```

tha displays its initial window:



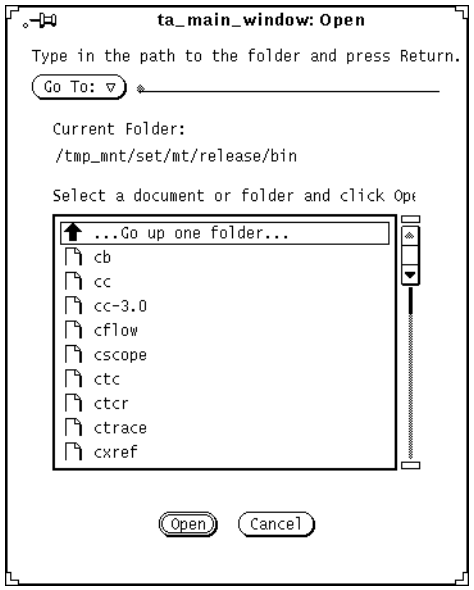
This window draws a hierarchy of glyphs representing the threads and functions that make up the program. The glyph hierarchy is described later in this chapter.

The Load button loads a trace directory. The View menu specifies the particular metric you are interested in and whether you wish to see the information as a table or graph. The Load button and View menu are described later in this chapter.

Loading a Trace Directory

If you did not specify the `tha.`*pid* trace data file in the command line, use the Load button to specify the directory that contains your trace data file.

Note – If you loaded a trace file via the command line, `tha` makes the Load button unavailable; that is, `tha` loads only one trace file per session.



Type the pathname of the trace directory generated by the instrumented program or click left on the icon of the desired trace directory from the displayed list. Click Open.

If `tha` does not recognize the pathname you typed, it emits an audible beep and clears the pathname. Check for typographic errors and retype the pathname.

1

`tha` *Glyph Hierarchy*

`tha` glyphs are arranged in the following hierarchy:

1. program
2. thread
3. functions

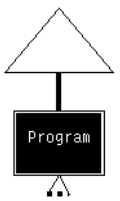
Navigating through the `tha` *Glyph Hierarchy*

To select a particular glyph, position the mouse to that glyph and click the left mouse button.

To expand (show a node's children) or collapse the glyph hierarchy (hide a node's children), position the mouse to the desired glyph and click the right mouse button. The right mouse button acts as a toggle to alternate between an expand or collapse operation.

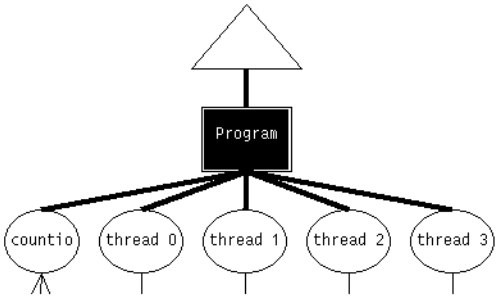
To hide an individual node, position the mouse to the desired glyph and click the middle mouse button.

The program glyph is the root of the hierarchy. Stub lines extending from the program node represent a further level of hidden nodes—the threads.



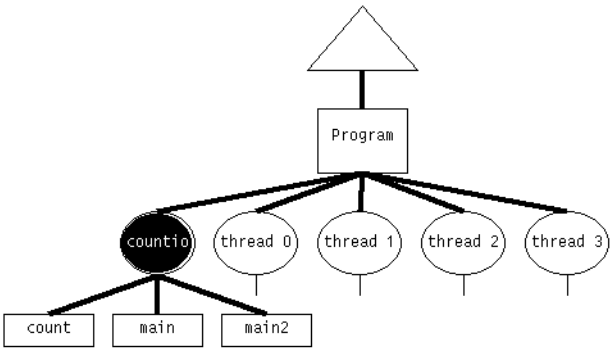
The thread level is next in the hierarchy. All threads are displayed across the width of the main window. To see obscured threads, move the slider along the horizontal scrollbar located at the bottom of the window.

The thread that contains `main()` is named with the program name; all other threads are named numerically in order of creation.



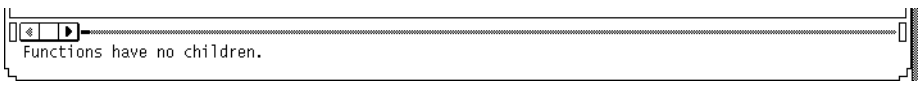
The stub lines extending from the thread level nodes represent a further level of hidden nodes—the functions.

The function level is the bottom of the tree. tha displays the function (thread children) glyphs in alphabetic order, from left to right. The children of a given thread are the procedures called within that thread. Use the scroll bar to scan function glyphs hidden from view.



Interactive Error Messages

`tha` displays error messages in the footer of the main canvas.



1

Exiting from `tha`

Use Window => Quit to exit the `tha` session.

Metrics Collected by `tha`

Metrics apply to objects. An object can be the entire program, a single thread, or a single function. If a metric refers to a single function, it refers to the total number of times that function was called by a particular thread. Function calls are implicitly divided up according to the thread that made the call.

Note – `tha` does not support metrics for forked programs.

The following discussion provides a brief description of each of the metrics collected by `tha`.

`CPU Time`
: This metric measures the amount of time an object was scheduled by the operating system, and was running on a CPU.

`Wall Clock Time`
: This metric measures the amount of time between when an object was created and when it was destroyed. The obvious difference from CPU Time is that when a thread is suspended (for instance waiting for a mutex), the thread still 'exists' even though it isn't taking up CPU time.

`Mutex Wait Time`
: This metric measures the wall-clock time an object spends suspended waiting to acquire a mutex lock.

`Join Wait Time`
: This metric measures the wall-clock time an object spends suspended in the thr_join function, waiting for another thread to terminate.

`Semaphore Wait Time`
: This metric measures the wall-clock time an object spends suspended waiting to acquire a semaphore.

`Condition Variable Wait Time`
: This metric measures the wall-clock time an object spends suspended waiting to be signalled on a condition variable.

`RW Read Lock Wait`
This metric measures the wall-clock time an object spends suspended waiting to acquire a read lock on a Reader/Writer style lock.

`RW Write Lock Wait`
This metric measures the wall-clock time an object spends suspended waiting to acquire a write lock on a Reader/Writer style lock.

`Total Sync Wait Time`
This metric measures the total wall-clock time spent waiting on any of the six forms of threads synchronization.

`Read Wait Time`
This metric measures the amount of time spent blocked on read system calls.

`File Write (bytes)`
This metric measures the number of bytes written per second by write system calls.

`File Writes (ops)`
This metric measures the number of write system calls made per second by a certain object.

`File Reads (bytes)`
This metric measures the number of bytes read per second by read system calls.

`File Reads (ops)`
This metric measures the number of read system calls made per second by a certain object.

`File IO (bytes)`
This metric measures the number of bytes read or written per second by an object via the read and write system calls.

`File IO (ops)`
This metric measures the combined number of read and write system calls made by an object per second

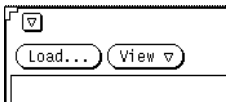
`tha` *Menus*

`tha` uses the following conventions for manipulating menus:

- ▼ in a button indicates there are more entries to be revealed. Pull right to reveal the entries.
- ❒ checkboxes in property sheets allow you to select items from a list of properties. Click left to select/unselect an item.
- click left on a ⊂⊃ button to specify an action; such as Apply or Cancel.
- menus are pinnable; that is, they stick around if you pin them to the canvas.

Main Menu Buttons

The Load and View buttons are the main menu buttons.



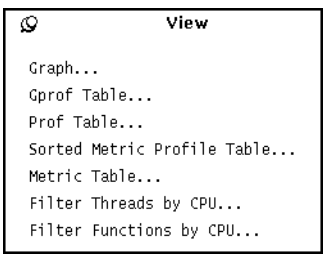
Load Button

The Load button allows you to specify the directory that contains the trace data file. See the discussion of "Loading a Trace Directory" earlier in this chapter.

View Button

Use the View menu to specify the particular metric you are interested in and whether you wish to see it as a table or graph. You can apply a CPU time filter to the glyph display to show only those threads or functions that exceed or equal a particular threshold.

To bring up the View Menu, pull right on the View button. Optionally, you can use the pushpin to pin the menu to the canvas.

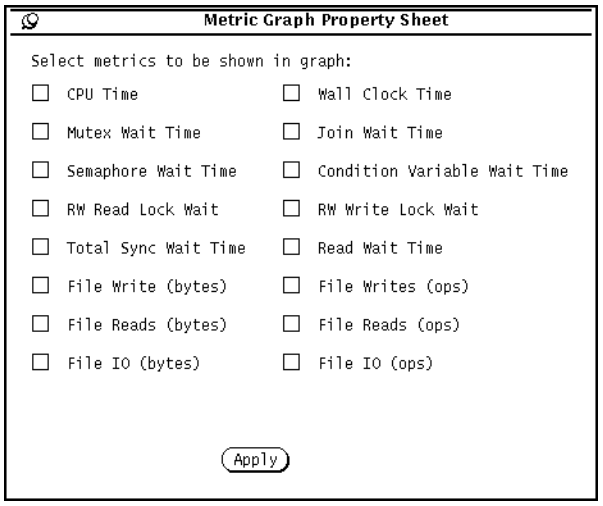


Metric Graph Property Sheet

Graphs plot the value of a metric against wallclock time. To bring up the Metric Graph Property Sheet, pull right on the View button and release on Graph.

Click left on the checkbox for the particular metrics you wish to graph. Click Apply to generate the graph.

Note – `tha` allows you to display a maximum of ten graphs at a time. `tha` emits an audible beep if you attempt to simultaneously display more than the ten graph maximum.



See “Scenario 2: Initial Bottleneck Identification via Graphical Data”, for an example of how to display CPU time versus wall clock time.

Gprof Table

Gprof tables display call-graph profile data for threads and functions. More specifically, the call-graph profile displays a list of functions called by threads .

To generate a `gprof` table for a particular level in the hierarchy, select a glyph at that particular level. Pull right on the View button and release on Gprof Table. See “Scenario 1: Initial Investigation — CPU Time Tabular Data”, for an example of how to display a `gprof` table.

Prof Table

Prof tables display the profile data for the program, threads, and functions. The program level `prof` table shows the total time for the entire program. The thread level `prof` table has an entry for each thread (with a total of all function calls made by the thread). The function level `prof` table has an entry for each thread/function combination.

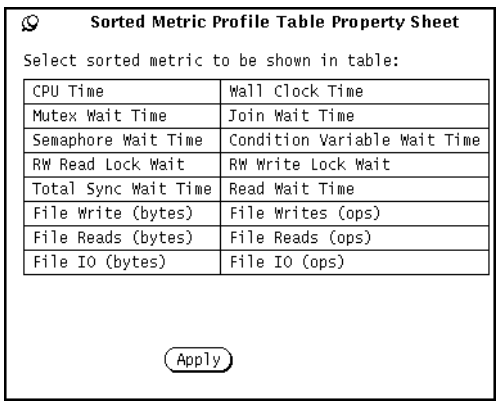
To generate a `prof` table for a particular level in the hierarchy, select a glyph at that particular level. Pull right on the View button and release on Prof Table.

See “Scenario 1: Initial Investigation — CPU Time Tabular Data”, for an example of how to display a `prof` table.

Sorted Metric Profile Table Property Sheet

A Sorted Metric Table shows a designated metric for all nodes for a particular glyph level.

To bring up the Sorted Metric Profile Table Property Sheet, select a glyph from the desired level in the hierarchy. Pull right on the View button and release on Sorted Metric Profile Table.



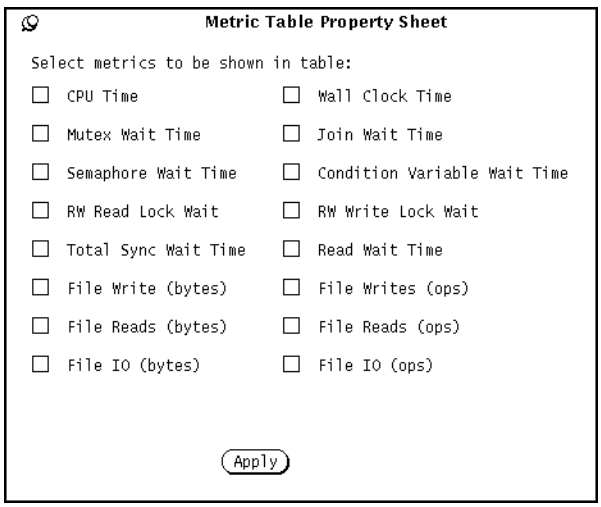
Click left on the particular metric you wish to graph—you may select only one from this list. Grayed out entries signify the metric is not available. Click Apply to generate the table.

See “Scenario 1: Initial Investigation — CPU Time Tabular Data”, for an example of how to generate a sorted metric table.

Metric Table

A Metric Table shows multiple metrics for a particular thread or function. For a function, the metrics are shown for calls of that function made by the thread that is that function’s parent.

To bring up the Metric Table Property Sheet, select a glyph from the desired level in the hierarchy. Pull right on the View button and release on Metric Table.

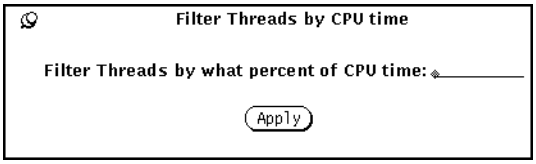


Click left on the checkbox for the particular metrics you wish to graph. Click Apply to generate the table.

See “Scenario 1: Initial Investigation — CPU Time Tabular Data”, for an example of how to generate a metric table.

Filter Threads by CPU ...

This filter displays the thread glyphs whose percent of CPU time is equal to or greater than a designated threshold.

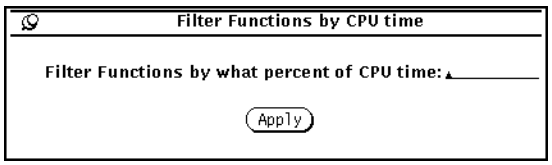


Type the threshold value and click Apply.

See “Scenario 4: CPU Time Filter”, for an example of how to filter threads by CPU time.

Filter Functions by CPU ...

This filter displays the function glyphs whose percent of CPU time is equal to or greater than a designated threshold.



Type the threshold value and click Apply.

Using `tha` 2

This chapter presents examples for how to use `tha` to determine what your program is doing and to see where bottlenecks are present.

"Scenario 1: Initial Investigation — CPU Time Tabular Data"
"Scenario 2: Initial Bottleneck Identification via Graphical Data"
"Scenario 3: Focused Bottleneck Identification via Graphical Data"
"Scenario 4: CPU Time Filter"

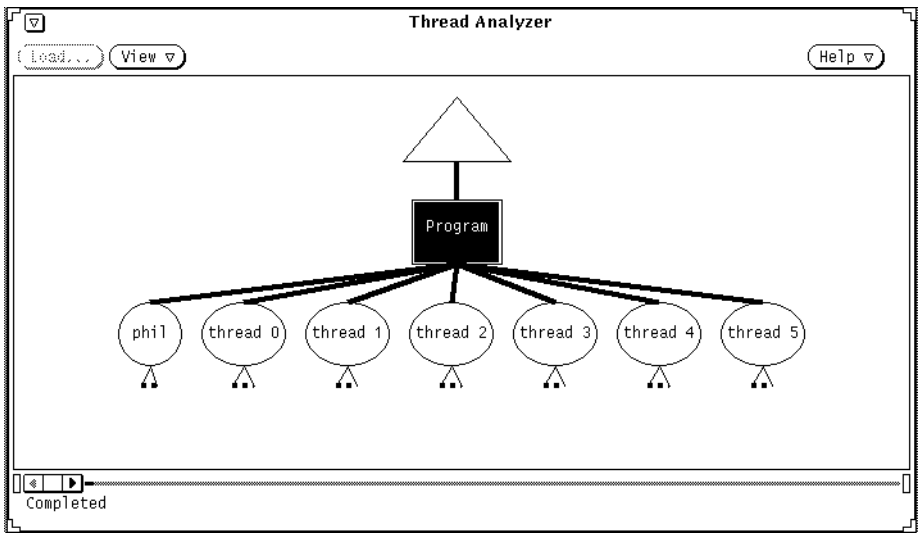
Scenario 1: Initial Investigation — CPU Time Tabular Data

The initial investigation gathers an overview of what your program is doing.

Looking at CPU time is a good starting point to discover where your program is spending its time.

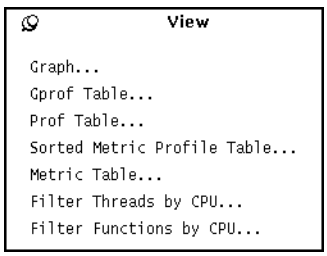
Step 1: Display the Thread Level Nodes

If the thread-level nodes are not already displayed, move the mouse to the "program" node and click right to expand the glyph display to the thread level nodes.



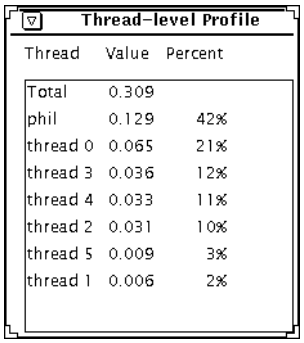
Step 2: Request a `prof` *Table for All Threads*

Select any thread glyph. Pull right on the View button and release on “Prof Table” to display prof information for all the threads in your program.



Step 3: Analyze the `prof` *Table Data*

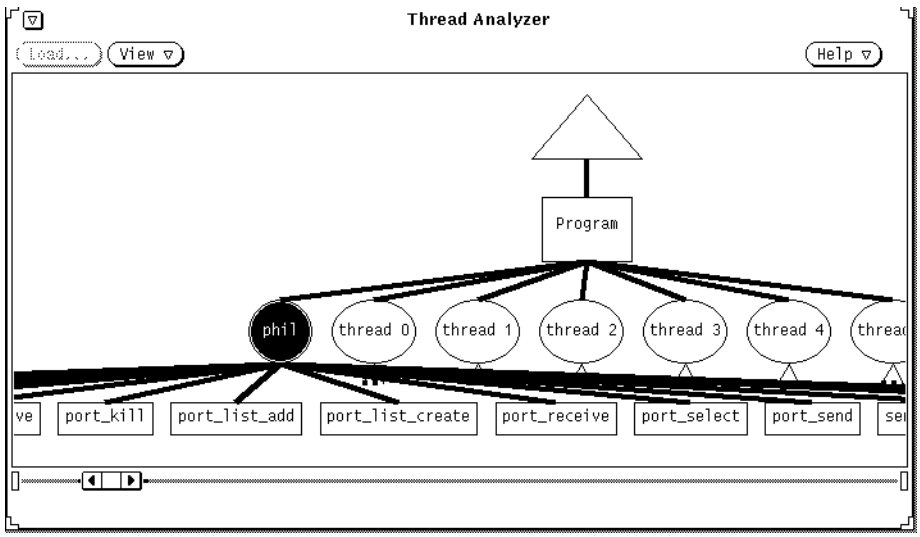
The prof table shows the percent of CPU time spent by all threads of the program. Table elements are arranged in descending order of CPU time.



See if any threads consume a disproportionate amount of CPU time. In our example the “phil” thread accounts for 42% of the CPU time. This is a likely candidate for further investigation.

Step 4: Focus on a Particular Thread

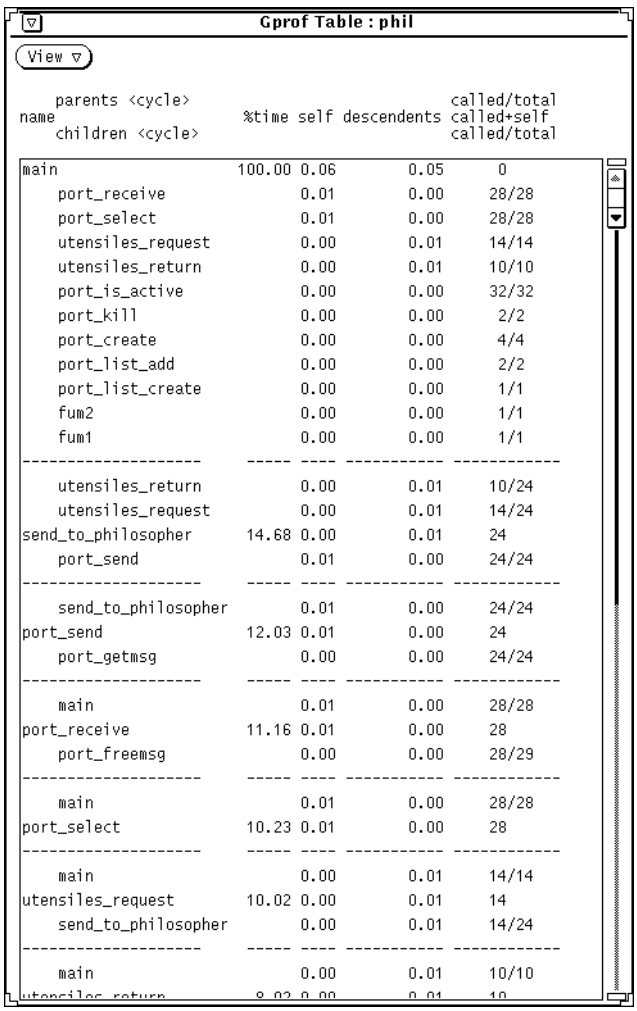
Move the mouse to the “phil” node and click right to expand the glyph display to show the “phil” node’s children (function nodes).



Step 5: Request a `gprof` *Table for a Particular Thread*

To generate a `gprof` table for a particular thread — "phil" is still selected — pull right on the View button and release on Gprof Table...

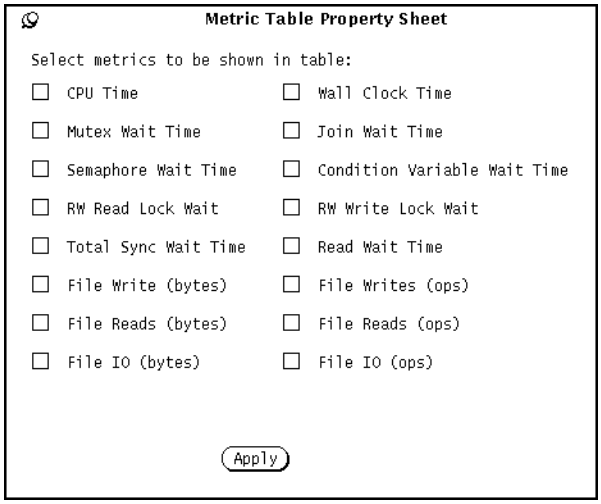
`tha` displays a thread-specific `gprof` table for the "phil" thread. See the `gprof` man page for more information.



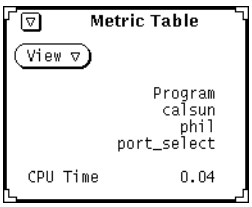
Note – called+self shows direct recursion; <cycle> shows indirect recursion.

Step 6: Request a Metric Table for a Particular Function

Select the glyph for the particular function of interest. Pull right on the View button and release on Metric Table... Click on the checkbox for CPU Time and click Apply. (Multiple metrics can be selected from this menu).



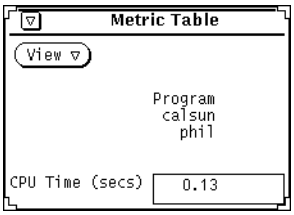
The resultant table shows the CPU Time used by the port_select function in the phil thread.



To display a Metric Table for a thread, select its glyph in the hierarchy. Repeat the request for a Metric Table for CPU Time. The resultant table shows the CPU Time used by the phil thread

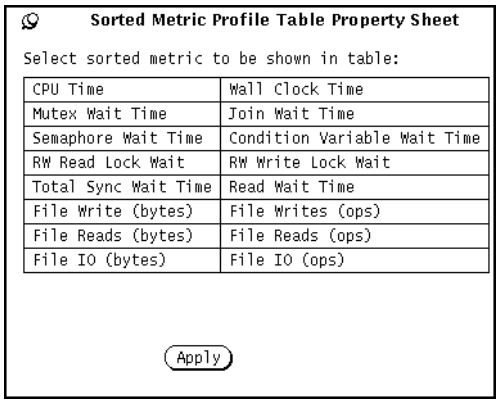
Step 7: Request a Sorted Metric ProfileTable for All Threads

`tha` displays a sorted profile table for a single metric for all functions in all threads.



Select any function in the glyph hierarchy. Pull right on the View button and release on Sorted Metric Profile Table...

Select the CPU Time entry in the property sheet and click Apply.



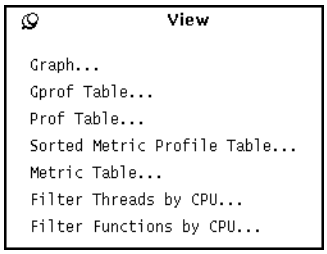
tha displays CPU time for all functions in all threads. The table is sorted in descending order of percent of time spent.

Scenario 2: Initial Bottleneck Identification via Graphical Data

`tha` can plot graphs of the metrics (graphed over time) to help discern the performance of your program. In particular, this feature helps discover performance bottlenecks. A common metric to view is the consumption of CPU time versus wall clock time.

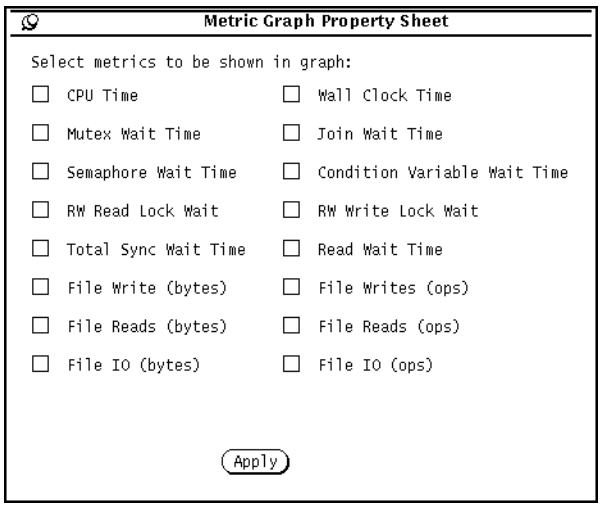
Step 1: Select the Node of Interest

Select the glyph for the thread to be analysed. Pull right on the View button and release on Graph..

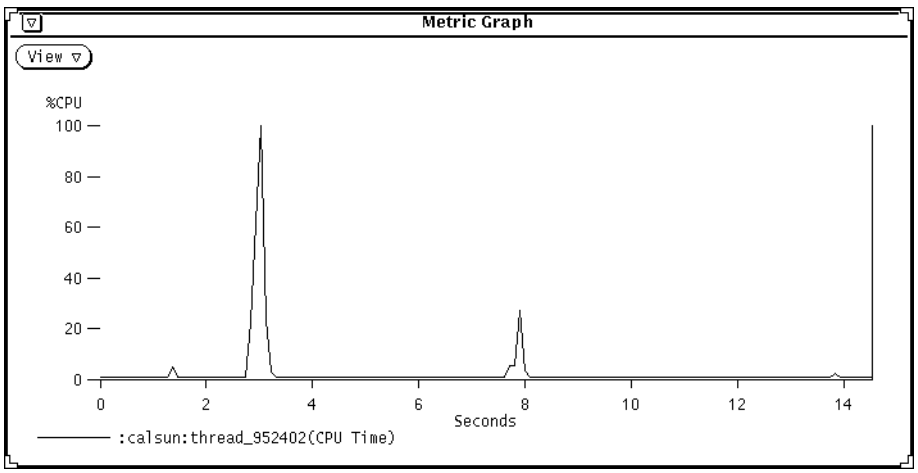


Step 2: Designate the Metric

Click on the checkbox for CPU Time and click Apply.



`tha` plots percent of CPU Time (y-axis) versus wallclock time (x-axis).



Except for a couple of “spikes”, a small amount of CPU time is being consumed. This suggests a bottleneck in I/O or thread synchronization (see “Scenario 3: Focused Bottleneck Identification via Graphical Data”).

Scenario 3: Focused Bottleneck Identification via Graphical Data

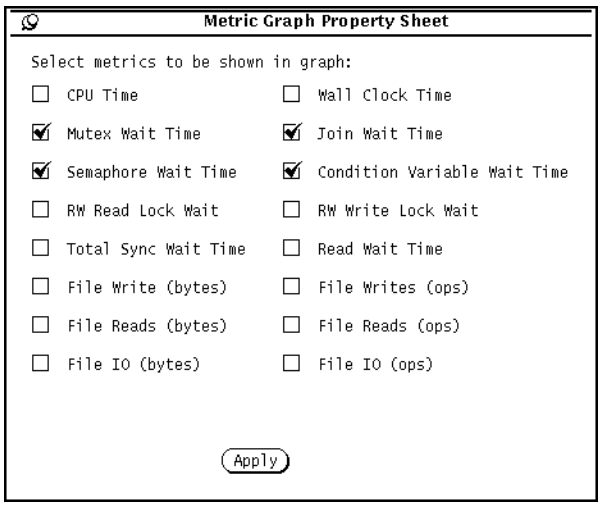
tha graphs can plot the time spent waiting on a resource. This helps you diagnose performance bottlenecks caused by blocking on I/O or thread synchronization.

Step 1: Select the Node of Interest

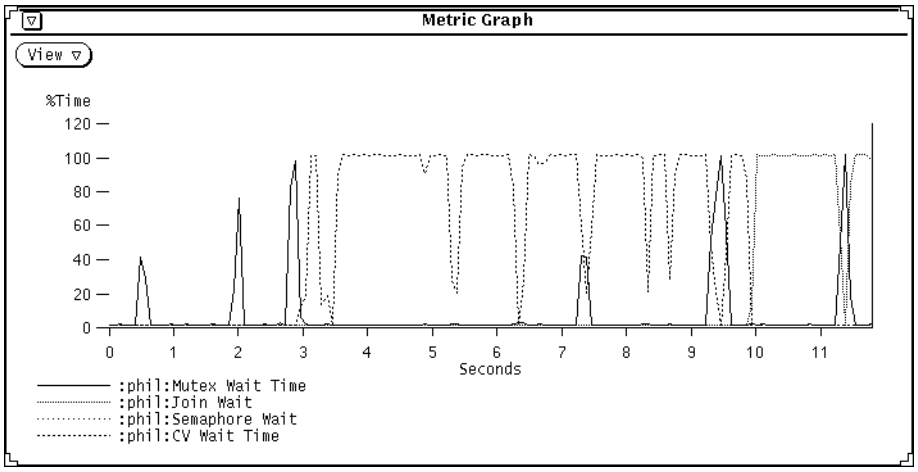
Select the phil glyph in the hierarchy. Pull right on the View button and release on Graph...

Step 2: Designate the Metric

Click the checkbox for the metrics you wish to graph and click Apply. In this case, the synchronization-related metrics are selected.



`tha` plots percent of time waiting for a resource (y-axis) versus wallclock time (x-axis). tha shows each metric with a different line representation (see the legend on the bottom of the graph).



This graph shows that most of the time wating for a resource was spent in CV (Condition Variable) Wait Time

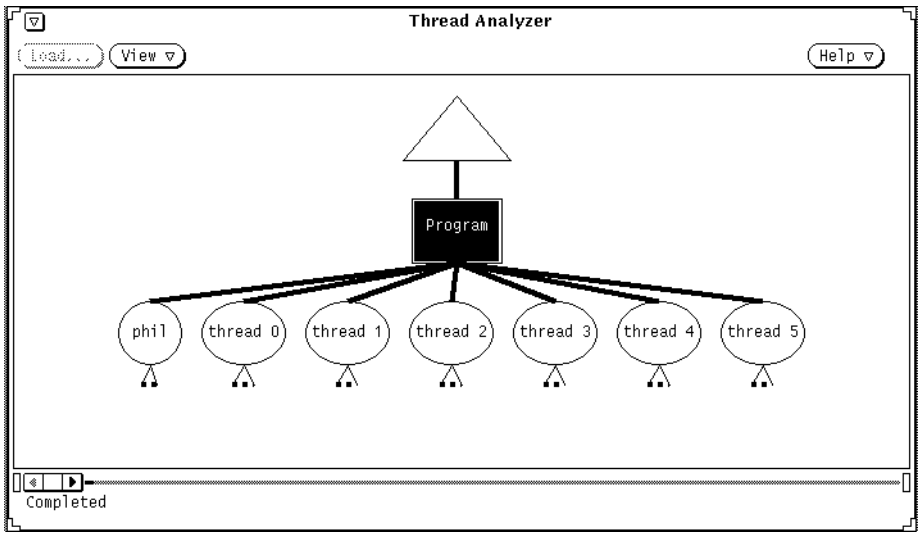
Scenario 4: CPU Time Filter

The Filter Threads by CPU operation modifies the glyph display to show only those threads that spend the designated percent of total CPU time.

Type a number (representing the threshold CPU Time percentage). The canvas collapses to show only those threads that account for at least the percent of CPU Time specified.

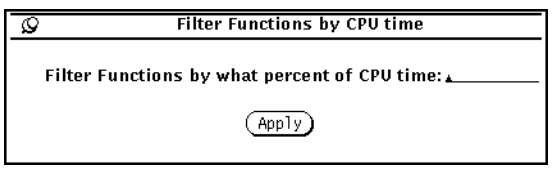
Step 1: Display the Thread Level Nodes

Expand the glyph hierarchy to show all threads. Select a thread icon.



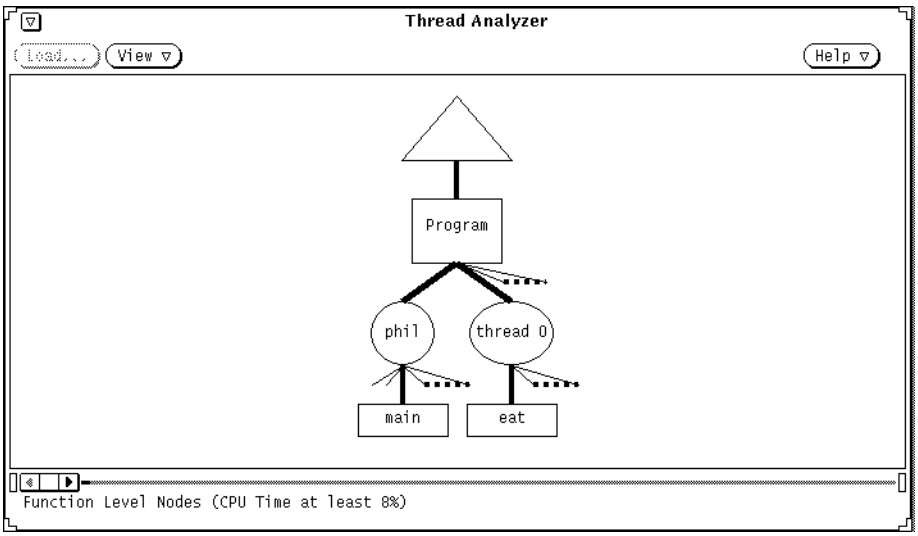
Step 2: Specify the Filter Value

Pull right on the View button and release on the Filter Threads by CPU ... entry.



Type the threshold value (8 for our example) and click Apply.

The canvas now collapses to show only those threads that account for at least 8% of the CPU Time of the program run.



Index

B

bottleneck, narrowing focus
    blocking on I/O or thread synchronization, 23
    graph condition variable wait time, 24

C

call-graph profile
    functions, 11
    threads, 11
collapse glyph hierarchy, 6
collecting metrics
    run object file, 2
condition variable
    time spent wating for signal, 8

E

error messages
    interactive, 7
exit `tha` session, 8
expand glyph hierarchy, 6

F

filter
    threshold CPU time for function, 14
    threshold CPU time for thread, 14
filter threads by percent of total CPU time, 25
function glyph, 6, 7

G

glyph
    function, 6
    program, 6
    thread, 6
glyph hierarchy
    collapse, 6
    expand, 6
gprof table
    call-graph profile for functions, 11
    call-graph profile for threads, 11

H

hierarchy
    glyphs, 6
    navigation, 6
horizontal scrollbar, 6

I

initial investigation

focus on particular thread, 17
gprof table for particular thread, 18
metric table for particular function, 19
prof table for all threads, 16
program CPU time, 15
sorted metric profile table for all threads, 19
instrument program
-Ztha, 1
interactive error messages, 7

L

Load button
load trace directory, 10
loading a trace directory
via command line, 2
via load button, 5

M

manipulating menus
conventions, 9
metric
condition variable wait time, 8
CPU time, 8
file I/O (bytes), 9
file I/O (ops), 9
file reads (bytes), 9
file reads (ops), 9
file write (bytes), 9
file writes (ops), 9
join wait time, 8
mutex wait time, 8
read wait time, 9
reader/writer read lock wait, 9
reader/writer write lock, 9
semaphore wait time, 8
total sync wait time, 9
wall clock time, 8
metric graph property sheet, 11
metric scope
entire program, 8
single function, 8
single thread, 8
metric table
multiple metrics for particular thread or function, 13
mouse gestures, 6
multiple metrics
particular function, 13
particular thread, 13
mutex lock
time spent wating to acquire, 8

N

navigating through glyph hierarchy, 6

P

performance bottlenecks
identified via graphical data, 21
plot CPU time versus wallclock time, 21
prof table
profile data for functions, 12
profile data for program, 12
profile data for threads, 12
profile data
functions, 12
program, 12
threads, 12
program glyph, 6
property sheet
metric graph, 11
sorted metric profile table, 12

R

read and write system calls
number of bytes read or written, 9
number of callls, 9
read system call
number of bytes read, 9
number per second, 9
time spent blocked on, 9
reader/writer read lock
time spent wating to acquire, 9

reader/writer write lock
time spent waiting to acquire, 9

S

semaphore
time spent waiting to acquire, 8
sorted metric profile table property sheet
particular glyph level, 12
specifying pathname to executable
via command line, 3
start tha
via comand line, 2

T

thread glyph, 6
thread naming convention, 7
thread synchronization
time spent on, 9
thread terminate
wait in `thr_join`, 8
trace data file, 5

V

View menu
display graph, 10
display table, 10
select metric, 10

W

window quit
exit tha, 8
write system call
number of bytes written, 9
number per second, 9
write trace data files
tha.pid directory, 2

Z

-Ztha
compiler option, 1

instrument program, 1
C, 2
C++, 2
FORTRAN, 2