CONVEX Consultant Programmer's Reference

Document No. 710-004130-000

First Edition
December 1989

CONVEX Computer Corporation
Richardson, Texas USA

CONVEX Consultant Programmer's Reference
Order No. DSW-029
First Edition

Printed in the United States of America

Consultant 8.0

Optional Product

CONVEX Consultant is comprised of the following utilities: the CONVEX symbolic debugger (*csd*), the *prof*, *bprof*, and *gprof* profilers, and a post-mortem dump utility (*pmd*). The CONVEX Consultant provides tools for debugging and optimizing C and FORTRAN programs on the CONVEX C100 and C200 Series supercomputers. It is addressed to experienced FORTRAN and C programmers. CONVEX Ada has its own debugger, which is not part of the Consultant package.

CONVEX Consultant documentation:

CONVEX Consultant User's Guide

List of Entries:

bprof(1)
csd(1)
gprof(1)
pmd(1)
prof(1)

NAME

bprof – display source-level execution counts

SYNOPSIS

bprof *options* [*a.out* [*bmon.out* ...]]

DESCRIPTION

The usual way of measuring program performance is by timing its execution. While this can provide the analyst with much useful information, the data obtained may have several inconsistencies. For instance, it is possible for the time for one routine to get attributed to another routine. In addition, many thousands of instructions may be executed during each sampling interval, making it impossible to get reliable counts for smaller subroutines. Finally, if the program is somehow dependent on the hardware clock, e.g. communications applications, the profiling sampling is not random.

As an alternative, *bprof* produces source-level execution count listings of C or FORTRAN programs. Knowing how many times a statement is executed enables the analyst to easily determine which parts of the program are executed most frequently. This in turn helps in identification of processing bottlenecks and indicates which parts of the routine have not been executed.

The profile data is taken from the basic block profile file *(bmon.out* default) which is created by programs compiled with the **–pb** option of *cc* and *fc*. The symbol table in the named object file (*a.out* default) is read and correlated with the basic block profile file. If more than one profile file is specified, the *bprof* output shows the sum of the information in the given files.

The following options are available:

–m Lists counting information by module name. This listing includes the number of routines in the module, the total number of calls made to them, a count of the number of source lines executed, the sum of all the lines executed, and the number of lines not executed.

–f Lists counting information by routine name. This listing includes the routine name, a count of the number of source lines executed in it, the sum of the lines executed, the number of source lines not executed, and the number of times it was called.

–l Produces a source listing containing the number of times each source line was executed.

–s Produces a file named *bmon.sum* which contains the sum of all the specified profile data files. This is only useful when more than one profile file is specified.

–I*directory*

Adds *directory* to the list of directories which are searched when looking for a source file. Normally, *bprof* looks for source files in the current directory and in the directory where *a.out* is located.

FILES

a.out	the namelist and text space.
bmon.out	basic block execution count data file

SEE ALSO

cc(1), prof(1), fc(1), gprof(1)
CONVEX Consultant User's Guide

BUGS

The profiled program must call *exit*(2) or return normally for the profiling information to be saved in the *bmon.out* file.

Source lines containing multiple statements are treated as a single statement. *indent*(1) can be used to split them onto individual lines if so desired.

A profiled version of *libc* is not available at this time.

NAME

csd – CONVEX symbolic debugger for C and FORTRAN programs

SYNOPSIS

csd [**-f**] [**-r**] [**-I** *dir*] [...] [*objfile* [*corefile*]]

DESCRIPTION

csd is a tool for source-level debugging and execution of C and FORTRAN programs under UNIX. The *objfile* is an executable file produced by a compiler with the *-g* or *-db* options specified to produce symbol information in the object file. It is possible to use *csd* on parallel and optimized code produced by the *fc*(1F), *vc*(1), and *cc*(1) compilers.

If no *objfile* is specified, *csd* prompts you for one. The object file contains symbol information, which includes the name of all the source files translated by the compiler to create it. You may view these files while using the debugger. Each time *csd* opens a source file, it prints a warning message if the source file is newer than *objfile*.

If a *corefile* file is specified, *csd* can examine the state of the program when it faulted. Unlike *adb*(1), a *corefile* is not read by default; its name must be explicitly entered.

If the file *.csdinit* exists in the current directory, the debugger commands in it are executed immediately after the symbolic information is read. *csd* also checks for a *.csdinit* in the user's home directory if there isn't one in the current directory.

The command line options and their meanings are as follows:

–f Execute *objfile* with fixed scheduling, reserving all processors for its use. Fixed scheduling should be used to debug programs which include parallel execution. Without fixed scheduling, whether threads are created depends on the system load.

–r Execute *objfile* immediately. If it terminates successfully, *csd* exits. Otherwise, the reason for termination is reported and the user offered the option of entering debugger commands or letting the program take the fault. *csd* reads from **/dev/tty** when *-r* is specified and standard input is not a terminal.

–I *dir* Add *dir* to the list of directories that are searched when looking for a source file. Normally, *csd* looks for source files in the current directory and in the directory where *objfile* is located. The directory search path can also be set with the *use* command. (See **Accessing Source Files.**) The space between the option and the directory name is optional. Several directories can be specified by repeating the option for each directory.

Unless *-r* is specified, *csd* displays a prompt and waits for a command. A command line may consist of multiple commands separated by a semicolon ";".

USE

Refer to *csd* sections in the *Consultant User's Guide.*

The most useful basic commands to know are *run* to run the program being debugged, *stop* for setting breakpoints, *print* for displaying variables, and *where* to obtain a list of the active routines. When *csd* prints a source line, it follows the line number with an "*" if execution can stop on that line as the result of a *stop, step,* or *next* command.

FORTRAN programmers note the following when issuing *csd* commands.

* The name of the main program is "MAIN_". If you use a PROGRAM statement, the name of the main program is still "MAIN_", not the name in the PROGRAM statement.
* The array notation uses parentheses "( )" for subscripted items.
* Type the names of variables, functions, and subroutines in lower case.
* Avoid using the *csd* reserved words **if, true, false, hex, decimal, chained, sequential,**

native, **ieee**, and **auto** as symbolic names.

* Do not use the logical operators **.not.**, **.and.**, **.or.**; instead use **not**, **and**, **or**, respectively. The logical operators **.eqv.**, **.neqv.**, and **.xor.** are not defined.
* The logical constants **.TRUE.** and **.FALSE.** are predefined; use **.true.** and **.false.**, respectively.
* Do not use the relational operators **.eq.**, **.ne.**, **.gt.**, **.ge.**, **.lt.**, **.le.**; instead use ==, !=, >, >=, <, <=, respectively.
* A statement function cannot be called because the compiler doesn't generate closed routines.
* An assignment of a label number to an integer variable is invalid as a *csd* command.
* Prepend an underscore to the name of a common block to refer to the block name.

C programmers note the following when issuing *csd* commands.

* The name of the main program is "main"
* The array notation uses brackets "[ ]" for subscripted items.
* Names of variables and routines are case sensitive; type them as they appear in the source.
* Avoid using the *csd* reserved words **true**, **false**, **hex**, **decimal**, **chained**, **sequential**, **native**, **ieee**, and **auto** as names.
* The enumeration constants **true** and **false** are predefined.
* Use the relational operators ==, !=, >, >=, <, <=.

Scope Rules

Command references to nonunique symbols must be qualified with the environmental pathname of the symbol:

[*module_name.*][*routine.*][*block_name.*]*symbol_name*

A partial pathname may be used if it is unique. The current environmental pathname is automatically maintained by *csd* as routines are entered and exited during execution. References to nonqualified symbols default to using the current environmental pathname. The current environmental pathname is changed when the current function is changed by the *func* command. The *block_name* is of the form **$b***n*; the *block_name* for global variables is **$b0**.

Multiple Language Programs

If you use *csd* on an application containing a mixture of C and FORTRAN routines, you must append an underscore to C function names in all command references.

Pathnames

When specifying the name of a file (not in the current directory) or a directory, you must use the full pathname. The characters '.' and '~' may not occur at the beginning of a pathname. This rule applies to the commands *run*, *rerun*, *status*, *where*, *dump*, *edit*, *file*, *source*, and *use*.

Execution and Tracing Commands

run [*args*] [< *filename*] [>[&] *filename*]
: Start executing *objfile*, passing *args* as command line arguments; use < to redirect standard input, > to redirect standard output, and >& to redirect both standard output and standard error.

rerun [*args*] [< *filename*] [>[&] *filename*]
: Like the *run* command, except that when *args* are omitted, *csd* uses the argument list from the previous *run* command.

trace [**in** *routine*] [**if** *condition*]
trace *source_line_number* [**if** *condition*]

trace *routine* [**if** *condition*]
trace *expression* **at** *source_line_number* [**if** *condition*]
trace *variable* [**in** *routine*] [**if** *condition*]
trace threads [**in** *routine*]

Print tracing information when the program is executed. The index associated with the command is used to turn the tracing off (see the *delete* command).

The first argument describes what is to be traced. If it is a *source_line_number*, *csd* prints the line immediately prior to execution. Source line numbers in a file other than the current source file must be preceded by the name of the file in quotes and a colon, e.g. *"foo.c":17*.

If the argument is a routine name *csd* prints information showing what routine called it, from what source line it was called, and what parameters were passed to it each time the routine is called. Also, the routine's return is noted and any return values are displayed.

If the argument is an *expression* with an **at** clause, *csd* prints the value of the expression whenever the identified source line is reached.

If the argument is a *variable*, *csd* prints the name and value of the variable whenever it changes. This form of tracing makes program execution substantially slower.

If the argument is *threads*, *csd* prints the thread number and source line for that thread when a thread is created, and the thread number and source line for the current thread when a thread terminates.

If no argument is specified, *csd* prints all source lines before they are executed. This form of tracing is substantially slower.

The clause "**in** *routine*" restricts tracing information to be printed only while executing inside the given routine.

Condition is a logical expression and is evaluated prior to printing the tracing information; if it is false, the information is not printed. A logical expression may be a relational expression or a parenthesized relational or logical expression connected by the logical operators **and** and **or**. The logical operator **not** is not included because it is unnecessary; the sense of a relational expression can always be reversed.

stop at *source_line_number* [**if** *condition*]
stop in *routine* [**if** *condition*]
stop *variable* [**if** *condition*]
stop if *condition*
stop threads [**in** *routine*]

Stop execution when the *source_line_number* is reached, *routine* is called, *variable* is changed, *condition* is met, or a thread is created or terminated.

status [> *filename*]

Print out the currently active *trace*, *stop* and *when* commands, along with their unique event index.

delete all
delete *index ...*

The *trace*, *stop*, or *when* command corresponding to the event *index* is removed, or all active *trace*, *stop* and *when* commands are removed. The event index associated with a

trace, *stop*, or *when* command is displayed by the *status* command.

when at *source_line_number* **{** *command*; [*command*; ...] **}**
when in *routine* **{** *command*; [*command*; ...] **}**
when *condition* **{** *command*; [*command*; ...] **}**
Execute the *csd command*(s) when the *source_line_number* is reached, the *routine* is called, or the *condition* is true. Execution continues after the last command is executed, unless it is the special command **STOP** which stops program execution.

catch [*signum* | *signame*]
Catch the signal identified by *signum* or *signame*. The signal numbers and signal names are described in */usr/include/signal.h* and in the *signal*(3c) manual page. When specifying a *signame*, omit the SIG prefix (so, refer to the SIGINT signal with INT). Initially all signals are trapped except SIGCONT, SIGCHLD, SIGALRM, SIGTSTP, SIGHUP, and SIGKILL.

With no arguments, *catch* lists all signals to be caught.

ignore [*signum* | *signame*]
Ignore the signal identified by *signum* or *signame*. The signal numers and signal names are described in */usr/include/signal.h* and in the *signal*(3c) manual page. When specifying a *signame*, omit the SIG prefix (so, refer to the SIGINT signal with INT). Initially, SIGCONT, SIGCHLD, SIGALRM, SIGTSTP, SIGHUP, and SIGKILL are ignored.

With no arguments, *ignore* lists all signals to be ignored.

cont [**all**] [*signum* | *signame*]
Continue execution of the current thread or all threads from a stop, or as if the signal *signum* or *signame* had occurred. Execution cannot be continued if the process has "finished," that is, called the standard procedure "exit". *csd* does not allow the process to exit, thereby letting the user examine the program state when the program terminates. When specifying a *signame*, omit the SIG prefix (so, refer to the SIGINT signal with INT).

goto [*lineNumber*
Changes the program counter so that execution will continue with the line specified. The line specified by *lineNumber* must lie in the current function. If the environment (e.g. sp, fp, PSW) is incorrect at that point, user beware.

mode [**chained** | **sequential**]
Display or set the default execution mode of the program being debugged. The default mode is sequential. This is only useful when debugging parallelized and vectorized applications.

step [**all**] [*count*]
Execute the current thread or all threads for *count* source lines. If omitted, *count* is one. Steps into routines, i.e., stops at the beginning of the routine.

next [**all**] [*count*]
Execute the current thread or all threads for *count* source lines. If omitted, *count* is one. Steps past routines, counting them as single steps.

thread [*number*]
Make thread *number* the current thread. If *number* is omitted, report what the number of the current thread is.

threads [*number*]
Limit execution to *number* threads for the next *run* or *rerun* command. If *number* is omitted, report the current thread limits and show the program counter value and

current line for each active thread.

return [*routine*]
Pop one or more stack frames from the top of the runtime stack. The child begins execution at the current program counter, and ends at the next statement after it returns. If the routine argument is used, stack frames are executed until the named *routine* is on the top of the stack.

quit Exit *csd*.

Displaying and Naming Data

Array elements are subscripted by brackets ([ ]) for C, and by parenthesis (( )) for FORTRAN.

print *expression* [, *expression* ...]
print *type*(*expression*)
print *expression* \ *type*
print *expression* \ *variable*
Print the values of the *expressions*. Variables having the same identifier as one in the current block may be referenced as "*block-name* . *variable*". The field reference operator . can be used with pointers as well as records, making the C operator -> unnecessary (although it is supported). To display subranges of arrays, specify as an index a lower and upper bound separated by a colon (:). If one dimension of a multi-dimensional array element reference specifies a subrange, all other dimensions must specify subranges, even if the desired subrange is only one element long. Alternate forms support type transfer, which permits the type of an expression to be temporarily overridden. Using a type transfer, the value of an *expression* may be printed as though it were of type *type* (C only), or the value of *expression* may be printed as though it were of the same type as *variable*. A valid *type* includes the simple C types (int, float, char, double), user defined typedefs, and structure or union types. For the latter (as in a declared structure or union), use $$*tag* where *tag* is the structure or union tag, or just the *tag* if its name is unique.

format [**hex** | **decimal**]
Display or set the default print format in which integers are printed. The default is decimal.

fpmode [**native** | **ieee** | **auto**]
Display or set the floating-point format used by the *dump*, *print* and *assign* commands. If no argument is given, the floating-point mode of *csd* and the program being debugged is displayed. If one of the three valid arguments is issued, the floating-point mode of *csd* changes accordingly. If *auto* is specified, the floating-point mode of *csd* changes as needed to match that of the program. Initially, the floating-point mode of *csd* is *ieee* or *native* for single-mode IEEE or native mode programs, respectively. If the program is a dual-mode program, *csd* begins in *auto* mode.

whatis *type*
whatis *variable*
Print the declaration of the given *type* (C only) or *variable*. A valid *type* includes the simple C types (int, float, char, double), user defined typedefs, and structure or union types. For the latter (as in a declared structure or union), use $$*tag* where *tag* is the structure or union tag, or just the *tag* if its name is unique.

which *identifier*
Print the full qualification of the function or variable whose name is *identifier* (for example, the blocks in which the *identifier* is nested or the source module in which the function resides). References to the identifier are resolved to this pathname by default. You

can change the identifier with the *func* command.

whereis *identifier*
Print the full qualification of all the functions and variables whose name matches the given *identifier*. The order in which the symbols are printed is not meaningful.

assign *variable* = *expression*
Assign the value of the *expression* to the *variable*. There is no type conversion for operands in the *expression*. Also, the type of the *expression* must match that of the *variable*.

call *routine*([*parameters*])
Execute the object code associated with the named user defined *routine*. The routine must be compiled in with the program. Currently, calls to a routine with a variable number of arguments are not possible.

up [*count*]
down [*count*]
Move up (toward "main") or down the call stack one or *count* levels. The current routine, which is used for resolving names, is automatically reset.

where [*integer*][> *filename*]
Print all active routines, or the top *integer* of routines, on the runtime stack.

dump [> *filename*]
Print post-mortem dump information.

set [**deref_aregs** | **dump_lfmt** | **dumpsrc** | **dumpvregs** = true | **false**]
set num_elements = *number* [, *number* ...]
set precision = *number*
Display or enable/disable the parameters used by the *dump* command when formatting listing output. *deref_aregs* causes the contents of all address registers to be dereferenced when output. *dump_lfmt* generates a complete detailed listing. *dumpsrc* causes the approximate location at which the program is stopped at to be displayed. *dumpvregs* results in the *vector* register contents to be displayed. *num_elements* sets the default indexes used for dumping array contents. It is used to specify regions of arrays to display and is useful for examining very large arrays. *precision* sets the precision in which floating-point numbers are displayed to *number* (the default precision is 6). Setting the precision to zero restores the default value.

Accessing Source Files

/regular expression[/]
?regular expression[?]
Search forward or backward in the current source file for the given pattern.

/[/]

?[?] Search forward (backward) in the current source file for the next occurrence of the last pattern specified from the next (previous) line to the end (top). The matching line becomes the new current line.

edit [*filename*]
edit *routine*
Invoke an editor on *filename* (default is the current source file). If a *routine* name is specified, the editor is invoked on the file that contains it. Which editor is invoked by default depends on the installation. The default can be overridden by setting the environment variable EDITOR to the name of the desired editor.

file [*filename*]

Change the current source file to *filename*. If it is not specified, *csd* prints the name of the current source file.

func [*routine*]
Change the current routine to *routine*. If no argument is specified, *csd* prints the current routine. Changing the current routine implicitly changes the current source file to the one that contains the *routine*; it also changes the current environmental pathname used for name resolution.

list [**eventlines**] [*startline* [[,] *endline*]]
list *routine*
List the lines in the current source file from the *startline* to the *endline*, inclusive. If no lines are specified, the next 10 lines are listed. If *eventlines* is specified, only lines for which events (e.g. *stop at*) can be specified are listed. If the name of a routine is given, lines *n-k* to *n+k* are listed, where *n* is the first statement in the routine and *k* is small.

use [*directory ...*]
Display or set the list of directories to be searched when looking for source files.

Machine Level Commands

tracei [*address*] [**if** *condition*] Trace execution of all instructions or a specific machine instruction at *address*.

tracei *variable* [**at** *address*] [**if** *condition*]
Trace changes to the *variable* at *address*.

stopi at *address* [**if** *condition*]
Set a breakpoint at machine instruction *address*.

stopi *variable* [**if** *condition*]
Stop if *variable* changes.

nexti [**all**] [*count*]
stepi [**all**] [*count*]
Single step as in **step** or **next**, but do a single or *count* machine instructions rather than source lines.

regs [**all**]
List all control, scalar, and address register contents for the current thread or all active threads.

cregs [*index*]
List all of the user communication registers or the specified register.

vregs [**all**] [*index*]
List the contents of all vector registers or of the specified vector register for the current thread or all active threads. If the vector registers are empty, *vregs* prints all zeroes.

address,*count*?*mode*
.,*count*?*mode*
Print the contents of memory, starting at the first *address* and continuing until *count* items are printed. If a dot is specified, the address following the one printed most recently is used. The *mode* specifies how memory is to be printed; if it is omitted, the

previous mode specified is used. The initial mode is **X**. The following modes are supported:

b print a byte in octal (8 bits)
c print a byte as a character (8 bits)
d print a word in decimal (32 bits)
D print a long word in decimal (64 bits)
f print a single-precision real number (32 bits)
F print a double-precision real number (64 bits)
g print a single-precision real number using the C printf %g format (32 bits)
G print a double-precision real number using the C printf %g format (64 bits)
i print the machine instruction
o print a word in octal (32 bits)
O print a long word in octal (64 bits)
s print a string of characters terminated by a null byte
x print a word in hexadecimal (32 bits)
X print a long word in hexadecimal (64 bits)

Symbolic addresses are specified by preceding the name with an &. Registers are denoted by $r, where *r* is the name of the register. Valid register names are as follows:

a0-a7	address registers
s0-s7	scalar registers
sp(a0)	stack pointer
ap(a6)	argument pointer
fp(a7)	frame pointer
psw	processor status word
pc	program counter

Addresses may be expressions made up of other addresses and the operators +, -, and indirection (unary *).

Miscellaneous Commands

sh *command-line*
Pass the command line to the shell for execution. The SHELL environment variable determines which shell is used.

alias

alias *newname*
alias *newname command*
alias *newname "string"*
alias *newname (parameters) "string"*
When commands are processed, *csd* first checks to see if the command name is an alias for either a *command* or a command *string*. If it is an alias, *csd* treats the input as though the corresponding *command* or *string* (with values substituted for any parameters) had been entered. For example, to define an alias "rr" for the command "rerun", type

alias rr rerun

To define an alias called "b" that sets a breakpoint at a particular line, type

alias b(x) "stop at x"

Subsequently, the command "b(12)" expands to "stop at 12".

If no parameters are specified, a list of all active user defined aliases and their values is printed. If just *newname* is specified, its alias is printed.

Alias commands may be placed in the *.csdinit* file in the current or home directory, and are defined when *csd* is invoked.

unalias *name*
Remove the alias with the given name.

help [*command*]
Display a summary of *csd* commands, or print a short synopsis explaining *command.*

source *filename*
Read and execute *csd* commands from the given *filename.*

quit Exit *csd.*

Standard Command Aliases

a	assign
b	stop
c	cont
d	delete
e	edit
f	func
h	help
l	list
n	next
p	print
q	quit
r	run
s	step
t	trace
w	where
st	status
wi	whereis
W	which

RESTRICTIONS

When setting conditional breakpoints, such as *stop if var1 < var2*, and the scope var1 and var2 is only in the routine *foo*; the breakpoint should be further qualified as: *stop in foo if var1 < var2.*

FILES

a.out	default object file
.csdinit	initial commands

SEE ALSO

cc(1), fc(1F) (Optional Product), vc(1) (Optional Product), pmd(1) (Optional Product), *CONVEX Consultant User's Guide*

NAME

gprof – display call graph profile data

SYNOPSIS

gprof [*options*] [**a.out** [**gmon.out...**]]

DESCRIPTION

Gprof produces an execution profile of C or FORTRAN programs. The effect of called routines is incorporated in the profile of each caller. The profile data is taken from the call graph profile file (*gmon.out* default) which is created by programs which are compiled with the **–pg** option of *cc* and *fc*. That option also links in versions of the library routines which are compiled for profiling. The symbol table in the named object file (*a.out* default) is read and correlated with the call graph profile file. If more than one profile file is specified, the *gprof* output shows the sum of the profile information in the given profile files.

First, a flat profile is given, similar to that provided by *prof*(1). This listing gives the total execution times and call counts for each of the functions in the program, sorted by decreasing time.

Next, these times are propagated along the edges of the call graph. Cycles are discovered, and calls into a cycle are made to share the time of the cycle. A second listing shows the functions sorted according to the time they represent, including the time of their call graph descendents. Below each function entry is shown its (direct) call graph children, and how their times are propagated to this function. A similar display above the function shows how this function's time and the time of its descendants is propagated to its (direct) call graph parents.

Cycles are also shown, with an entry for the cycle as a whole and a listing of the members of the cycle and their contributions to the time and call counts of the cycle.

The following options are available:

–a Suppresses the printing of statically declared functions. If this option is given, all relevant information about the static function (*e.g.*, time samples, calls to other functions, calls from other functions) belongs to the function loaded just before the static function in the *a.out* file.

–b Suppresses the printing of a description of each field in the profile.

–c The static call graph of the program is discovered by a heuristic which examines the text space of the object file. Static-only parents or children are indicated with call counts of 0.

–e *name*
Suppresses the printing of the graph profile entry for routine *name* and all its descendants (unless they have other ancestors that aren't suppressed). More than one **–e** option may be given. Only one *name* may be given with each **–e** option.

–E *name*
Suppresses the printing of the graph profile entry for routine *name* (and its descendants) as **–e**, above, and also excludes the time spent in *name* (and its descendants) from the total and percentage time computations. (For example, **–E** *mcount* **–E** *mcleanup* is the default.)

–f *name*
Prints the graph profile entry of only the specified routine *name* and its descendants. More than one **–f** option may be given. Only one *name* may be given with each **–f** option.

–F *name*
Prints the graph profile entry of only the routine *name* and its descendants (as **–f**, above) and also uses only the times of the printed routines in total time and percentage computations. More than one **–F** option may be given. Only one *name* may be given with each

-F option. The **-F** option overrides the **-E** option.

-s A profile file *gmon.sum* is produced which represents the sum of the profile information in all the specified profile files. This summary profile file may be given to subsequent executions of *gprof* (probably also with a **-s**) to accumulate profile data across several runs of an *a.out* file.

-z Displays routines which have zero usage (as indicated by call counts and accumulated time). This is useful in conjunction with the **-c** option for discovering which routines were never called.

FILES

a.out	the namelist and text space.
gmon.out	dynamic call graph and profile.
gmon.sum	summarized dynamic call graph and profile.
/vmunix	Kernel name list
/dev/mem	Kernel data values
/lib/kernsyms/symdata_*	Kernel symbol addresses

SEE ALSO

monitor(3), profil(2), cc(1), prof(1)
"gprof: A Call Graph Execution Profiler", by Graham, S.L., Kessler, P.B., McKusick, M.K.; *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, SIGPLAN Notices, Vol. 17, No. 6, pp. 120-126, June 1982.

BUGS

Beware of quantization errors. The granularity of the sampling is shown, but remains statistical at best. We assume that the time for each execution of a function can be expressed by the total time for the function divided by the number of times the function is called. Thus the time propagated along the call graph arcs to parents of that function is directly proportional to the number of times that arc is traversed.

Parents which are not themselves profiled will have the time of their profiled children propagated to them, but they will appear to be spontaneously invoked in the call graph listing, and will not have their time propagated further. Similarly, signal catchers, even though profiled, will appear to be spontaneous (although for more obscure reasons). Any profiled children of signal catchers should have their times propagated properly, unless the signal catcher was invoked during the execution of the profiling routine, in which case all is lost.

The profiled program must call *exit*(2) or return normally for the profiling information to be saved in the gmon.out file.

NOTES

Gprof is an optional product; for more information, contact your CONVEX sales representative.

NAME

pmd – display post-mortem dump if command aborts

SYNOPSIS

pmd [**–alsvS**][**–d num:num: ...**][**–t cpu_limit**] command [arguments]

DESCRIPTION

Pmd executes a *command* with the *arguments* supplied on the command line. If *command* aborts and creates a core file, *pmd* will display information about the aborted process. The information will be written to *stderr*. The post-mortem information includes:

1. The signal that caused *command* to abort
2. A runtime stack backtrace and the approximate source line location of where *command* took the exception
3. The contents of the machine registers
4. A dump of active local variables in each routine on the runtime stack
5. A dump of global, or common, variables
6. The region of disassembled object code where the exception took place
7. A summary of resources used by *command*: execution time, elapsed time, percent of time in CPU, size of shared memory and unshared memory, page faults, and swaps.

The options for *pmd* and their meanings are:

–a Dereference the address registers and print their contents in hex, decimal, and floating point formats.

–l Display a post-mortem dump in long format. The long format for *pmd* includes items 1. through 7. described above.

–s Display a post-mortem dump in short format. The short format includes: the signal that caused *command* to abort, a runtime stack backtrace, and the approximate source code location of the exception. The short format is the default format for *pmd*.

–v Include the contents of the machine's vector registers in the post-mortem dump listing.

–S Exclude the approximate source code location of where *command* took the exception from the post-mortem dump listing.

–d Print up to *num* elements of arrays in the post-mortem dump listing. Up to seven dimensions for arrays can be specified with this option. The default is 100:10:1 — print up to 100 elements of the first dimension of arrays, up to 10 elements of the second dimension of arrays, up to 1 element of the third dimension of arrays, and zero elements for dimensions four through seven.

–t Limit the cpu time for *command* to *cpu_limit* seconds.

If it becomes necessary to re-examine the post-mortem dump information, *csd* can be invoked with the appropriate commands to reproduce the post-mortem dump listing.

RESTRICTIONS

For *pmd* to obtain information about the runtime stack backtrace, source line locations, and active local variables; *command* must be compiled with the appropriate option. For *fc*(1f) and *vc*(1), the appropriate compiler option is **–db**. For *cc*(1), the the appropriate compiler option is **–g**.

The higher the optimization level specified when compiling *command*, the greater the uncertainty of variables and source line location accuracy. Several assumptions can be made, however:

1. Regardless of optimization level, common block variables' memory locations are guaranteed to be accurate at the call to a subprogram.

2. The validity of memory location contents of active local variables **cannot** be guaranteed. In most code, however, the memory locations for local variables are likely to be accurate.

3. The mapping of object code to source code is granular to the basic block. A basic block is defined as a sequence of statements with no branches.

4. The values of subprogram arguments at entry points to subprograms are accurate.

The core file must be created in order for *pmd* to produce post-mortem dump information. This necessitates that the user have write permission in the working directory of *command*. In addition, the maximum core file size must not be exceeded. Maximum core file size is set in the user's shell.

SEE ALSO

cc(1), csd(1), csh(1), fc(1f), vc(1), signal(3C)

NAME

prof – display profile data

SYNOPSIS

prof [**–a**] [**–l**] [**–n**] [**–z**] [**–s**] [a.out [mon.out ...]]

DESCRIPTION

Prof interprets the file produced by the *monitor* subroutine. Under default modes, the symbol table in the named object file *(a.out* default) is read and correlated with the profile file *(mon.out* default). For each external symbol, the percentage of time spent executing between that symbol and the next is printed (in decreasing order), together with the number of times that routine was called and the number of milliseconds per call. If more than one profile file is specified, the output represents the sum of the profiles.

In order for the number of calls to a routine to be tallied, the **–p** option of *cc, fc,* or *pc* must have been given when the file containing the routine was compiled. This option also arranges for the profile file to be produced automatically.

Options are:

–a All symbols are reported rather than just external symbols, so that static functions appear in the output.

–l The output is sorted by symbol value. The rightmost of the -n and -l options prevails.

–n The output is sorted by number of calls. The rightmost of the -n and -l options prevails.

–s A summary profile file is produced in *mon.sum*. This is really only useful when more than one profile file is specified.

–z Routines which have zero usage (as indicated by call counts and accumulated time) are nevertheless printed in the output.

FILES

mon.out for profile
a.out for namelist
mon.sum for summary profile

SEE ALSO

monitor(3), profil(2), cc(1), gprof(1), plot(1G)

BUGS

Beware of quantization errors. The granularity of the sampling is shown, but remains statistical at best. We assume that the time for each execution of a function can be expressed by the total time for the function divided by the number of times the function is called.

NOTES

Prof is an optional product; for more information, contact your CONVEX sales representative.