Michael Palmer
Jeffrey M. Russo

Parasight: Debugging and Analyzing Real-time Applications under Digital UNIX

Conventional UNIX debug and analysis tools, with their static debugging model and low-resolution-sampling profiling techniques, are not effective in dealing with real-time applications. Encore Computer Corporation has developed Parasight, a set of debug and analysis tools for real-time applications. The Parasight tool set can debug running programs, debug multiple programs, constantly monitor local and global variables, and perform on-the-fly execution analysis. Thus, Parasight provides much improved debug and analysis capabilities, which application developers can use on both static and dynamic applications. Parasight can be used on any of Digital's Alpha platforms running under the Digital UNIX operating system.

Because of their time-critical nature, real-time applications do not respond well to the perturbations that conventional UNIX debug and analysis tools cause. For instance, the static debugging model of the dbx debugger requires that a program be stopped before it can be debugged. Also, execution analysis using the profiling techniques of the prof profiler often provide erroneous results for real-time applications because of the low-resolution sampling employed.

This paper describes the critical aspects of debugging real-time applications, the deficiencies found in conventional UNIX tools, and the methodology Encore Computer Corporation used to develop Parasight, a set of easy-to-use graphical user interface tools that debug and perform execution analysis on real-time programs while they are running. Parasight can be used on any of Digital's Alpha platforms that operate under the Digital UNIX operating system.

Real-time Applications

Real-time applications perform a wide variety of functions, from flying state-of-the-art military aircraft to controlling nuclear power plants. All real-time applications have one common denominator: They must complete their calculations before a deadline expires. Taking too long to calculate the correct answer can have just as detrimental an effect as arriving at an incorrect answer; either result could cause an aircraft to crash or a nuclear power plant to experience a meltdown.

Most real-time applications consist of one or more programs that are scheduled to run in response to an event. The triggering event is usually transmitted in the form of an interrupt and can be generated randomly by an external event or regularly by a interval timer running at a fixed rate, such as 60 times per second. Once the interrupt is received, the application must perform the allotted task before the next interrupt occurs.

The elements of a real-time application communicate with each other dynamically; that is, the results of the calculations of one element are used immediately for the calculations of another element. Real-time applications are often referred to as dynamic applications, since they react dynamically to changes in their environment and often refer to elapsed time in their calculations. In contrast, static applications have results that rarely depend on changes in their environment or on elapsed time.

The Problems Associated with Debugging and Analyzing Real-time Applications Using Conventional UNIX Tools

Debugging a real-time application during execution, debugging and analyzing multiple programs, constantly monitoring variables, and analyzing program execution are all activities that debug and analysis tools have to deal with. This section discusses the capabilities and limitations of conventional UNIX tools and describes the features required of effective real-time debug and analysis tools.

Running Programs

Debugging a static program typically involves controlling the execution flow and examining the values of variables within the program. Stopping a real-time program or even delaying it by single stepping, however, is usually not possible without adversely affecting the application. Debugging real-time applications is therefore limited to examining the values of program variables while the program is still running.

Conventional UNIX debuggers are not able to examine variables during program execution and therefore cannot be employed on running real-time applications. Consequently, these debuggers are useful only in the early stages of real-time program development, essentially while the program is still static.

The traditional methods of debugging real-time applications involve placing all the critical data into one or more global, shared memory regions. A data-monitoring tool, usually written by the user, runs as a normal UNIX process and attaches to the global region. The tool can then be used to inspect and/or change the values of the global variables. This technique is nonintrusive in that it does not affect the real-time application programs in any way. Unfortunately, the debugging is restricted to global data, and, unless the programs are designed with this in mind, this restriction can be a severe limitation. Modifying existing programs to change local data into global data for debugging purposes can result in a whole new set of problems in managing the separation of data.

An effective real-time debugging tool must be able to attach to a running program without stopping it and then be able to nonintrusively inspect and/or change the global data.

Debugging and Analyzing Multiple Programs

Real-time applications typically consist of several programs working together. Invoking multiple copies of the `dbx` debugger to debug each program individually is cumbersome and precludes studying the interaction between programs.

A real-time debugger must be able to work with one or more programs at the same time, providing the user with an integrated and cohesive debugging environment.

Monitoring Variables

The one-shot variable evaluation capability of conventional UNIX debuggers is of limited use for programs that are running. These debugging tools provide the user with only one previous value of a variable, not necessarily the current value.

A real-time debugger must be able to constantly monitor the values of global variables. The minimum and maximum values that each variable attained should optionally be available as a record of transient conditions.

Execution Analysis (Profiling)

Since performance is important in real-time applications, program execution analysis is often needed to locate areas of a program where the performance could be improved. A real-time application may have a strict execution order requirement, whereby one routine must run prior to the execution of another routine. This requirement may be accomplished easily if the routines are in the same program; however, often the routines are in different programs or are executing on different CPUs in a symmetric multiprocessing (SMP) environment.

The Digital UNIX profiling tools provide two kinds of execution analysis:

1. PC sampling, which involves interrupting the program periodically to record the value of the program counter.
2. Block counting, which inserts profiling code at key points in the program to count the number of times each basic block of code executes. (A basic block is a region of the program that can be entered only at the beginning and exited only at the end.)

Both techniques involve the following basic steps:

1. Preprocess the program to produce the desired profiling information.
2. Execute the program to produce a profiling data file.
3. Postprocess the program with the profiling tools to view the data collected.

The normal sampling period employed by the PC-sampling method is based on the hard clock (CLOCK_REALTIME) of the Digital UNIX operating system. This method results in 1,024 samples being taken per second, which provides a timing resolution of 976 microseconds, or approximately 1 millisecond.

The routines that make up a real-time application typically take from a few microseconds to several milliseconds to execute. Attempting to measure the execution time of routines that take less than 1 millisecond to execute with a clock resolution of 1 millisecond can lead to erroneous results. A test on a 150-megahertz (MHz) Alpha 21064 CPU showed that the prof tool, using the normal PC-sampling rate, reported the execution time of a routine to be 4 milliseconds when the true execution time was 20 microseconds. (The true execution time was measured using the Parasight tool set.)

It is possible to increase the sampling rate using the uprofile utility, but doing so also proportionally increases the number of interrupts per second that the system must handle. For instance, to obtain even 10-microsecond resolution would require the system to handle 100,000 interrupts per second. This amount of interrupt activity would rapidly swamp the system, leaving little or no CPU time to execute the program being instrumented. The PC-sampling method of execution analysis is therefore not suitable for the short execution times typical in real-time application routines.

The block-counting method, although capable of high-resolution measurement, suffers from the inability of the pixie utility to work with programs that receive signals. Most real-time applications use signals for program scheduling and are therefore disqualified from using the block-counting method.

In addition to the problems just discussed, the traditional UNIX profiling tools are unsuitable for real-time program execution analysis for the following reasons:

- A program must be preprocessed for profiling prior to execution. Adding or removing profiling requires stopping, processing, and restarting the program. This assumes that the problem area is known before the application starts to run. If a problem suddenly develops after an uninstrumented program has been running for 24 hours, the user will have lost the opportunity to determine which routine is causing the problem.
- A program must be profiled as a whole, unless source code modifications are made to the program to control the profiling. This can cause excessive overhead, which real-time programs usually cannot tolerate.
- The profiling results cannot be seen until the program terminates, unless source code modifications are made to the program to permit the results to be dumped on command. The user needs to see the results while the program is running and often needs to repeat a test several times to get the desired results. Stopping and restarting the application once for each test could be laborious.
- Only the average and cumulative times for each routine are available. That is, the individual execution times for each call to a routine are not available. This also precludes the examination of the calling sequence.
- The results cannot be cross-correlated between programs to provide information about the relative calling sequences between programs or across processors.

A real-time execution analysis tool must operate with sufficient resolution to measure the execution time of a routine that may take 10 microseconds to execute. The instrumentation should be dynamically insertable into the current areas of interest and should be able to move to new areas of interest as required—all without stopping and restarting the real-time application.

Parasight: A Solution for Real-time Debugging and Program Analysis

Parasight is an integrated set of real-time debugging and analysis tools with a graphical user interface. The tool set consists of a debugger (Debug), a data monitor (DataMon), and a program analysis tool (Paragraph). The Parasight tool set solves the real-time deficiencies found in dbx, prof, and the other conventional UNIX debug and analysis tools used under the Digital UNIX operating system. Parasight is able to debug applications in either a dynamic (running) or a static (stopped) state; it can perform debugging and program execution analysis on several programs simultaneously, without adversely affecting the dynamics of time-critical applications.

Parasight's Foundation

The Parasight tool set features the use of a symbol table, the /prof file system, global memory, and scanpoints.

The Symbol Table, .pg File, and /proc File System Parasight's source of knowledge about the target application is derived from the symbol table and the .pg file. Both are generated at compile time as a result of the -para special compiler option.

Parasight manipulates target applications by using the /proc file system services available under the Digital UNIX operating system. The /proc file system enables Parasight to control the program flow and to read and write any memory address in the target application.

Global Data Just as the traditional means of debugging real-time applications depends on global memory regions, Parasight uses the global memory access concept as the basis for accomplishing most of its advanced capabilities. Parasight either accesses the target program data directly, through the use of /proc, or uses global memory to access data gathered for Parasight by one of its scanpoints.

Scanpoints The Parasight tool set uses global memory access whenever possible to provide nonintrusive access to the target application. Certain functions, however, require access to data that is local to a program. Parasight accesses this data through small segments of code called scanpoints.

A scanpoint is a function that is dynamically inserted into the target program by Parasight. The scanpoint function then runs in the same context as the target program and thus has access to all the local data of the program. The scanpoint function works as an agent for Parasight, gathering data that Parasight does not have direct access to. The Parasight tool set uses two principal types of scanpoints: datamon-scanpoints, which are used by DataMon to perform local data monitoring, and sensor-scanpoints, which are used by Paragraph to perform program execution analysis.

Inserting the scanpoints does not require modifying the application's source code or preprocessing the application's object code. The only requirement is to link each program with the special -para option. This adds a memory buffer to the target program for use by Parasight. The buffer is benign until used by Parasight.

Parasight dynamically inserts scanpoints by using the /proc service to build a scanpoint template in the special buffer of the target program. This can occur even while the program is running. The template code contains the functionality to

- Save the register state that existed when the program counter was at the scanpoint insertion location
- Set up the arguments to the scanpoint function, including the register state
- Call the scanpoint function
- Restore the register state
- Execute the instruction that was originally at the insertion location
- Branch back to the instruction following the insertion location

Parasight then dynamically alters the template code according to the insertion location and the instruction contained therein. If the instruction was a branch control instruction, Parasight alters the instruction's displacement so that the location corresponds to the instruction's new displaced location within the template. All other instructions, including jump control instructions, do not require altering and are simply copied to the new displaced location.

Once this code is constructed in the buffer, Parasight completes the scanpoint insertion process by overwriting the instruction at the insertion location with a branch to the newly generated scanpoint template. The fixed instruction length of Digital's Alpha microprocessors simplifies this step enormously.

It is important to note that the scanpoint is built by Parasight, not the target program. The target program is affected only by the final step of the scanpoint insertion, when Parasight overwrites the instruction at the insertion location. This design prevents excessive interference of the target program. Scanpoints are written in highly optimized code to minimize the impact on the target application when they are executed.

Parasight dynamically deletes scanpoints by writing back the original instruction at the insertion location. This design allows Parasight to disable a scanpoint even if the scanpoint function has not completed.

Meeting Requirements

Parasight has the capabilities required of effective real-time debugging and analysis tools.

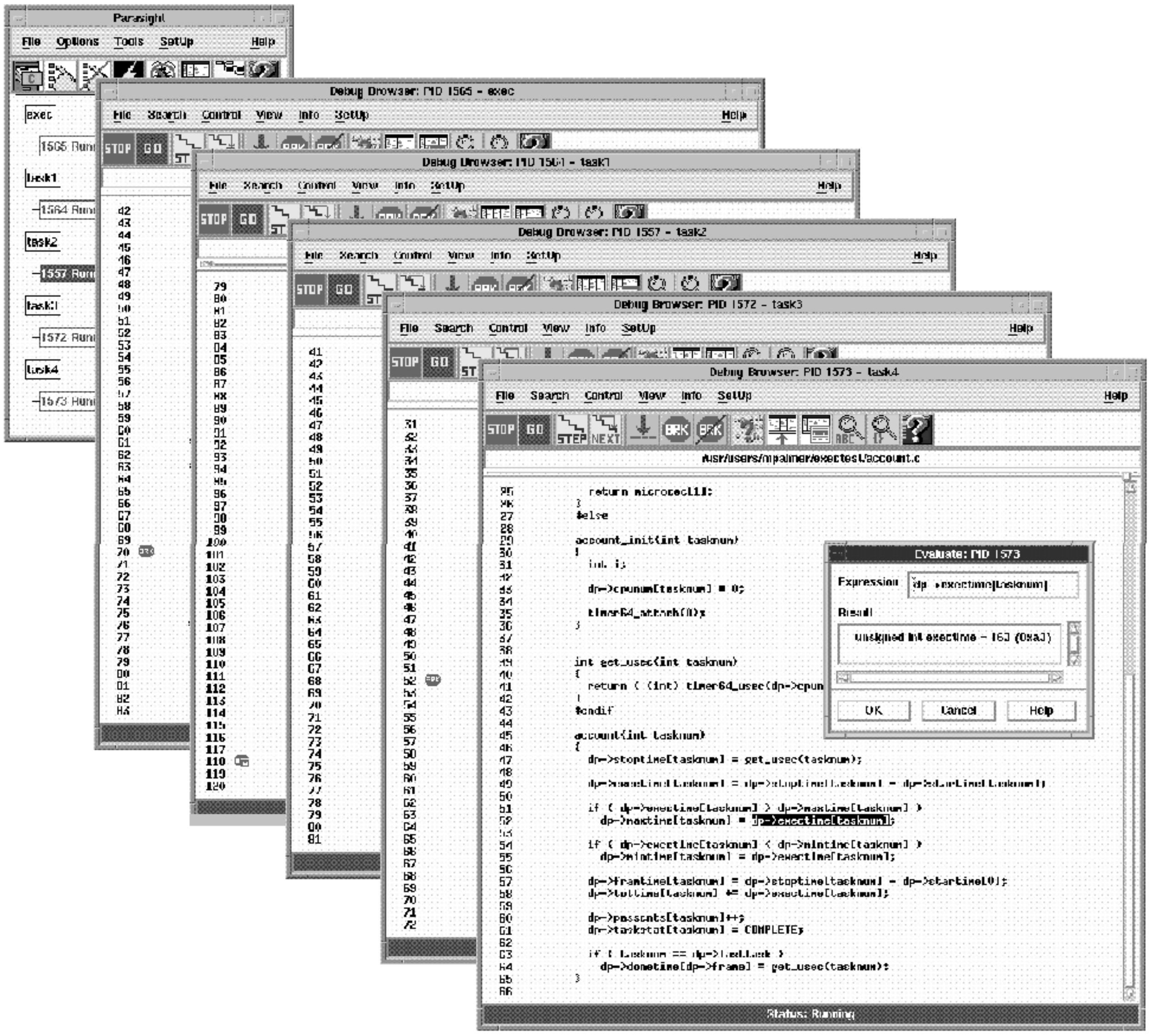
Debugging Running Programs Conventional UNIX debuggers deliberately stop a program when attaching to it, because these tools do not operate on running programs. When Parasight's debugger, Debug, attaches to a program, there is no impact on the program.

Conventional UNIX debuggers refuse to access any data while a program is running, even though global data resides at fixed memory locations that are accessible at all times through the /proc service. The reason for this limitation of the conventional UNIX tools is unclear. Parasight's debugger is able to examine and to change the value of any global data while the program is running or stopped.

Conventional UNIX tools also refuse to set any breakpoints in a program while the program is running. Again, the reason for this constraint is unknown. Parasight's debugger is able to insert breakpoints into running programs, a feature that is valuable in debugging error conditions in real-time applications.

Debugging Multiple Programs Parasight's Debug, DataMon, and Paragraph components constitute an integrated set of tools capable of working on one or more applications simultaneously, as shown in Figure 1. The Parasight main window displays the programs (and any children they create) attached to Parasight. The window also provides an easy mechanism to access the Parasight tool for each specific program.

Monitoring Variables Constantly Parasight's DataMon tool allows the user to simultaneously monitor the values of any local or global variables in one or more stopped or running programs. Parasight constantly monitors the values and shows any change on the DataMon display screen. DataMon is also capable of displaying the minimum, maximum, and average


Parasight
File Options Tools SetUp Help
exec
1565 Runi
task1
1564 Runi
task2
1557 Runi
task3
1572 Runi
task4
1573 Runi
Debug Browser: PID 1565 - exec
Debug Browser: PID 1564 - task1
Debug Browser: PID 1557 - task2
Debug Browser: PID 1572 - task3
Debug Browser: PID 1573 - task4
File Search Control View Info SetUp Help
STOP GO STEP NEXT BRK BRK ABC
/usr/users/mpalmer/exectest/account.c
Status: Running
Evaluate: PID 1573
OK Cancel Help

```
    return microsec[1];
}
#else

account_init(int tasknum)
{
  int i;

  dp->cpunum[tasknum] = 0;

  timer64_attach(0);
}


int get_usec(int tasknum)
{
  return ( (int) timer64_usec(dp->cpun
}
#endif

account(int tasknum)
{
  dp->stoptime[tasknum] = get_usec(tasknum);

  dp->exectime[tasknum] = dp->stoptime[tasknum] - dp->starttime[tasknum];

  if ( dp->exectime[tasknum] > dp->maxtime[tasknum] )
    dp->maxtime[tasknum] = dp->exectime[tasknum];

  if ( dp->exectime[tasknum] < dp->mintime[tasknum] )
    dp->mintime[tasknum] = dp->exectime[tasknum];

  dp->frametime[tasknum] = dp->stoptime[tasknum] - dp->starttime[0];
  dp->tottime[tasknum] += dp->exectime[tasknum];

  dp->passcnts[tasknum]++;
  dp->taskstat[tasknum] = COMPLETE;

  if ( tasknum == dp->lasttask )
    dp->donetime[dp->frame] = get_usec(tasknum);
}
```

Expression: dp->exectime[tasknum]

Result: unsigned int exectime = 163 (0xa3)

Figure 1
Parasight's Debugger Working with Five Tasks Simultaneously

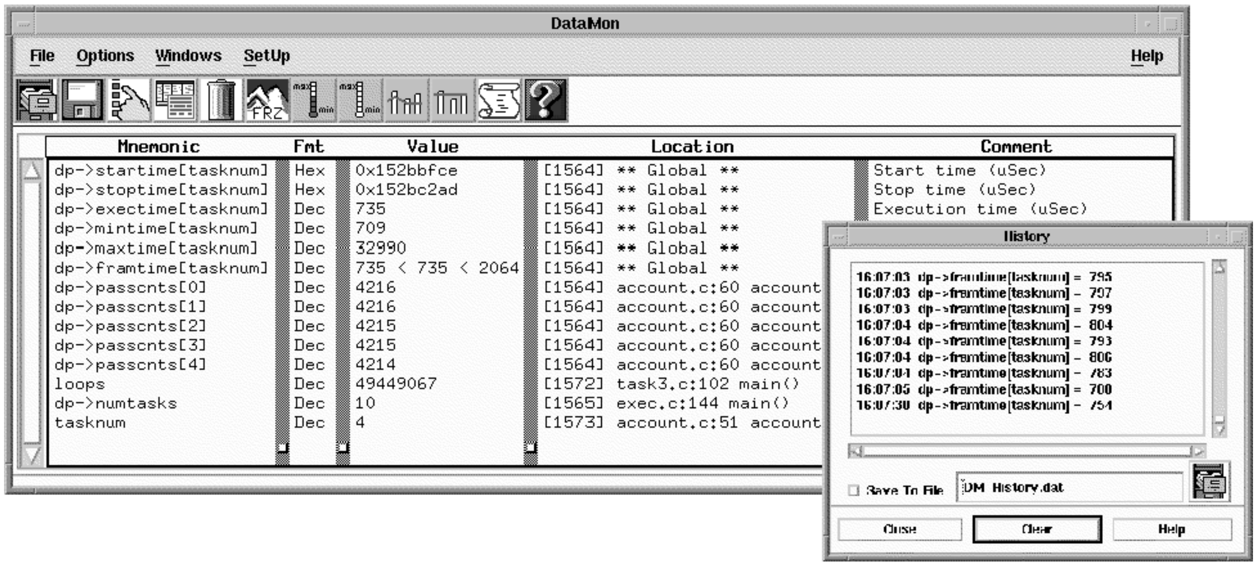
values attained for any variable. A scrolling history display along with a time stamp is also available for solving transient problems.

The variables to be monitored can be selected using the mouse on the Debug browser or entered into a dialog box using the keyboard. The DataMon graphical user interface has a point-and-edit capability, which allows the user to edit the mnemonic data (i.e., name, display format, value, location, or comment) directly on the screen. The user can store mnemonic lists on disk for fast retrieval when required. Figure 2 shows a DataMon display screen.

DataMon is able to monitor global data completely and nonintrusively using the /proc service and uses a datamon-scanpoint to implement local data monitoring. The datamon-scanpoint is attached to the DataMon database, which is a shared memory region connecting all the scanpoints and the DataMon display program. The datamon-scanpoints deposit the values of local data into the database for the display program to show on the screen. Datamon-scanpoints are also used to change the values of local data, depositing the value from the database into the specified variable.

DataMon uses the Debug tool's expression evaluator to parse the required mnemonic to derive the location of the value to be displayed. This may include register access for local variables saved on the stack. Multiple mnemonics can be monitored locally at the same location since a datamon-scanpoint function can traverse a list of mnemonics to be monitored.

Note that DataMon monitors data asynchronously; therefore, DataMon cannot guarantee to display every



```
Mnemonic                  Fmt   Value                Location                       Comment
dp->startime[tasknum]     Hex   0x152bbfce           [1564] ** Global **            Start time (uSec)
dp->stoptime[tasknum]     Hex   0x152bc2ad           [1564] ** Global **            Stop time (uSec)
dp->exectime[tasknum]     Dec   735                  [1564] ** Global **            Execution time (uSec)
dp->mintime[tasknum]      Dec   709                  [1564] ** Global **
dp->maxtime[tasknum]      Dec   32990                [1564] ** Global **
dp->framtime[tasknum]     Dec   735 < 735 < 2064     [1564] ** Global **
dp->passcnts[0]           Dec   4216                 [1564] account.c:60 account
dp->passcnts[1]           Dec   4216                 [1564] account.c:60 account
dp->passcnts[2]           Dec   4215                 [1564] account.c:60 account
dp->passcnts[3]           Dec   4215                 [1564] account.c:60 account
dp->passcnts[4]           Dec   4214                 [1564] account.c:60 account
loops                     Dec   49449067             [1572] task3.c:102 main()
dp->numtasks              Dec   10                   [1565] exec.c:144 main()
tasknum                   Dec   4                    [1573] account.c:51 account
```

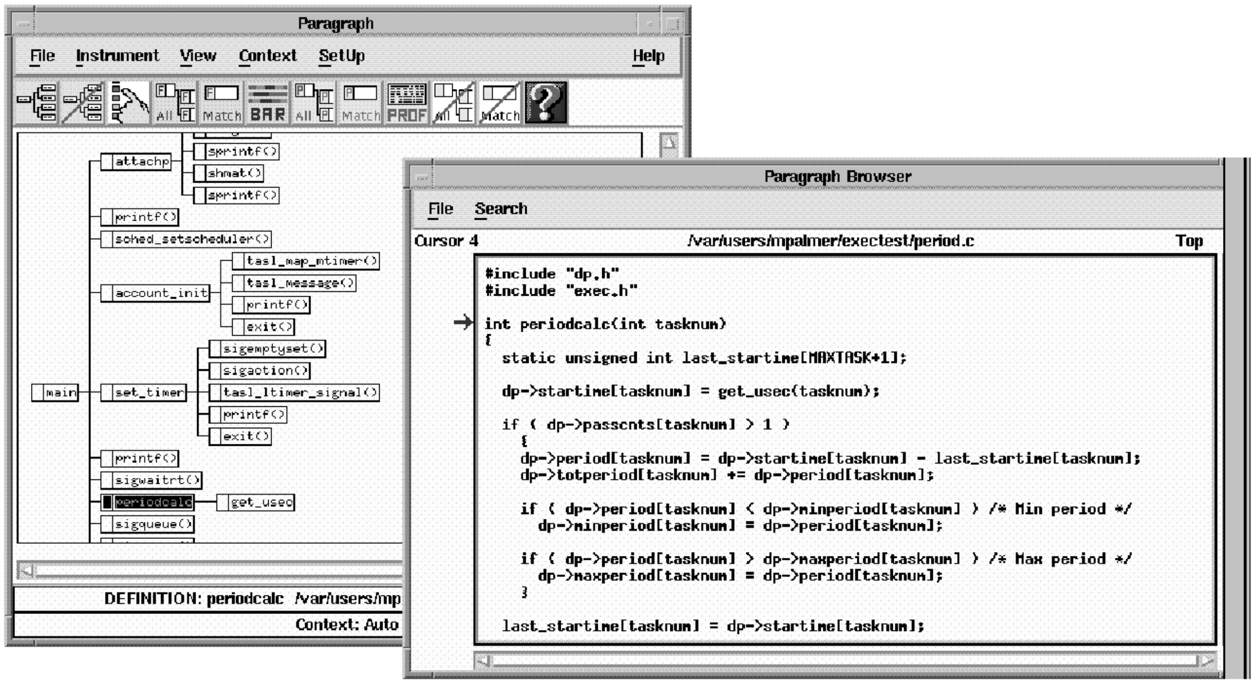
```
16:07:03 dp->framtime[tasknum] = 795
16:07:03 dp->framtime[tasknum] - 797
16:07:03 dp->framtime[tasknum] = 799
16:07:04 dp->framtime[tasknum] - 804
16:07:04 dp->framtime[tasknum] = 793
16:07:04 dp->framtime[tasknum] - 806
16:07:04 dp->framtime[tasknum] - 783
16:07:05 dp->framtime[tasknum] = 700
16:07:30 dp->framtime[tasknum] - 754
```

DM History.dat

Figure 2
The DataMon Display Screen with History Window

value that the variables reach. For global data, Parasight records only the minimum and maximum values that DataMon sees. For local data, however, the scanpoint keeps track of the minimum, maximum, and average values, so these can be guaranteed. Parasight can also monitor global data by using a datamon-scanpoint to monitor the value at a particular line of code.

On-the-Fly Execution Analysis Paragraph displays static source-code call graphs of the application's programs, illustrating the hierarchy of function calls, system calls, and statement-level control flow. Point-and-click operations allow the user to quickly view the source code for any program or function, thus simplifying the task of analyzing source code. Figure 3 shows a Paragraph call graph and browser.



DEFINITION: periodcalc /var/users/mp
Context: Auto

/var/users/mpalmer/exectest/period.c

```
#include "dp.h"
#include "exec.h"

int periodcalc(int tasknum)
{
  static unsigned int last_startime[MAXTASK+1];

  dp->startime[tasknum] = get_usec(tasknum);

  if ( dp->passcnts[tasknum] > 1 )
    {
    dp->period[tasknum] = dp->startime[tasknum] - last_startime[tasknum];
    dp->totperiod[tasknum] += dp->period[tasknum];

    if ( dp->period[tasknum] < dp->minperiod[tasknum] ) /* Min period */
      dp->minperiod[tasknum] = dp->period[tasknum];

    if ( dp->period[tasknum] > dp->maxperiod[tasknum] ) /* Max period */
      dp->maxperiod[tasknum] = dp->period[tasknum];
    }

  last_startime[tasknum] = dp->startime[tasknum];
```

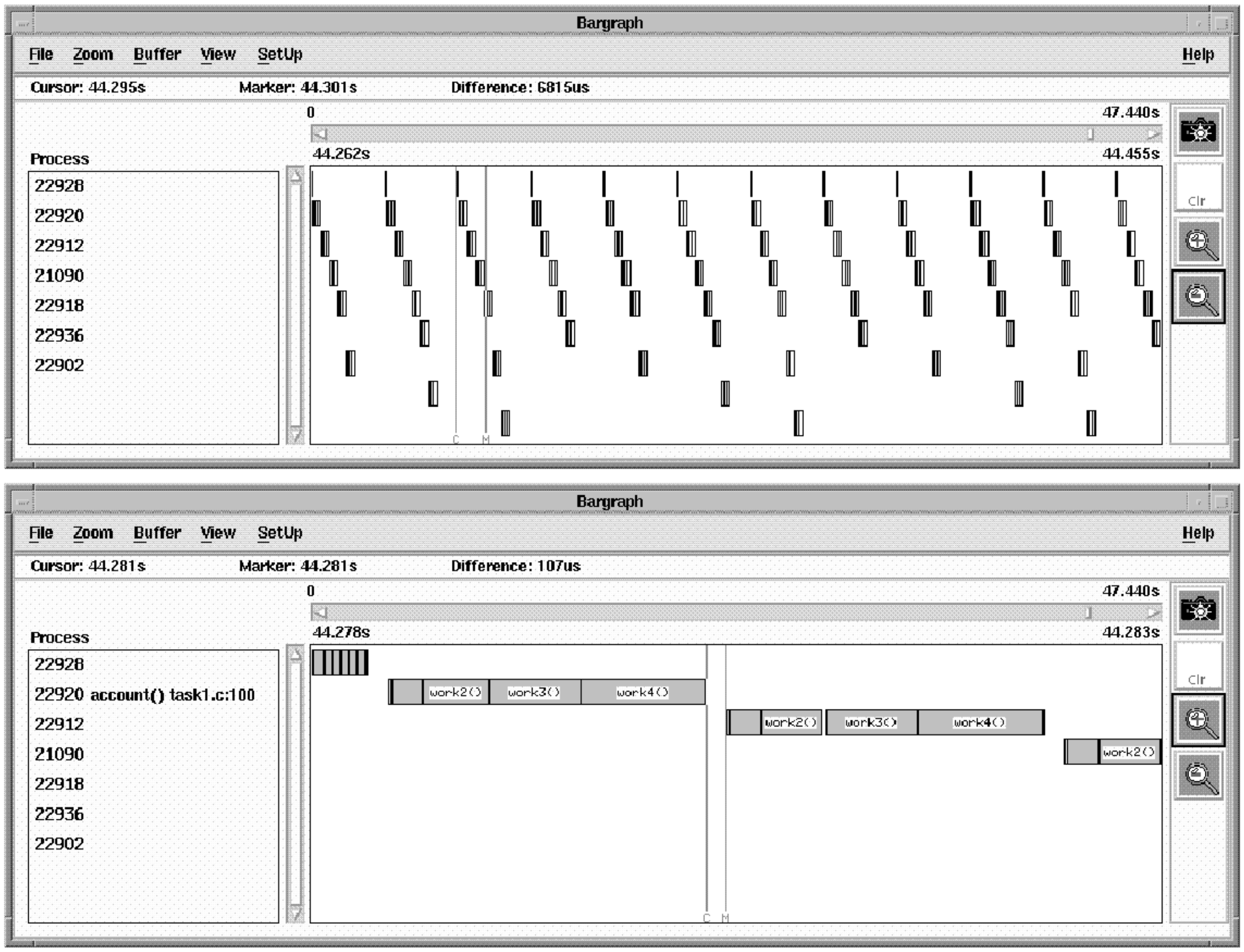
Figure 3
Paragraph Call Graph and Browser

Call graphs are also used to define where to insert instrumentation in an application. The instrumentation is used to perform execution timing analysis on a part or the whole of one or more of an application's programs. The instrumentation is inserted dynamically into a running program, without the need for source-level changes or object code preprocessing and without significantly affecting the dynamics of a running application. The inserted instrumentation may be deleted or added to at any time.

Paragraph uses sensor-scanpoints to measure how long a function takes to execute. The sensor-scanpoint function is placed at a branch-to-subroutine instruction. The function takes a time stamp from a nanosecond-resolution timer before and after the instruction to note the exact time the function started and ended. The sensor-scanpoints are attached to the Paragraph database, a shared region accessible to the sensor-scanpoints and Paragraph. Data is written into the database each time an instrumented function is executed. The results of the instrumentation may be viewed immediately, even while the program is running. The graphical view shows each function call as it occurred in time. Each program has a different bar, so the user can determine the relative time between functions called in different programs or even across multiple processors in an SMP environment. The zoom capability may be used to measure time periods down to a single microsecond. Figure 4 shows the Paragraph graphical display, called Bargraph, and the zoom capability.

Data gathering is continuous until the instrumentation is removed, so new data can be added onto the previous snapshot's view at any time. Multiple Bargraph windows can be used to recall previously saved timing data to easily compare current results with past results.

The nanosecond-resolution timer used by Paragraph is derived from the process control counter (PCC) register available on all Alpha microprocessors. This 32-bit, free-running timer operates at the same rate as the microprocessor and therefore provides a



Cursor: 44.295s Marker: 44.301s Difference: 6815us

22928
22920
22912
21090
22918
22936
22902

Cursor: 44.281s Marker: 44.281s Difference: 107us

22928
22920 account() task1.c:100
22912
21090
22918
22936
22902

Figure 4
The Paragraph Graphical Display, Bargraph, Showing Zoom Capability

3.6-nanosecond-resolution timer on a 275-MHz Alpha CPU. Unfortunately, since it is only a 32-bit timer, it wraps every 15.6 seconds. Parasight keeps track of the wrap count to create a 64-bit timer that allows problem-free timing for more than 2,000 years!

Adverse Effects

Although, ideally, the Parasight tool set should be completely nonintrusive and thus not affect the application in any way, such operation is not completely achievable for all functions. Capabilities such as inspecting (Debug) and monitoring (DataMon) global variables require no intrusion to implement; however, monitoring local variables and analyzing program execution do require a small amount of intrusion.

While most real-time applications cannot tolerate exceeding the time available for the completion of the task, they do have some spare time available after completing the task. Without this spare time, the risk of exceeding the deadline before program completion would be too great. This spare time can be used judiciously for the mildly intrusive functions of Parasight.

Summary

This paper discusses several capabilities required to effectively debug and analyze real-time applications. These capabilities include debugging of running programs, constant monitoring of variables, and on-the-fly execution analysis. The paper also details some of the problems associated with conventional UNIX tools, such as the inability to debug running programs, the adverse effects on target programs, the erroneous profiling results, and the cumbersome operation. Encore Computer Corporation's Parasight tool set offers a solution to these difficult problems. The paper describes the methodology behind the product and the capabilities that make Parasight an invaluable tool for debugging and analyzing real-time applications.

Acknowledgments

The authors would like to acknowledge the efforts of the following Parasight team members for their contributions to the product: Raghuveer Chakravarthi, Dileep Katta, Carlos Gonzalez, Deborah Grimstead, and Ken Shaffer.

General References

Z. Aral, I. Gertner, and G. Shaffer, "Efficient Debugging Primitives for Multiprocessors" (Fort Lauderdale, Fla.: Encore Computer Corporation, 1989).

DEC OSF/1 Programmer's Guide, Section 6 (Maynard, Mass.: Digital Equipment Corporation, August 1994).

Biographies

Michael Palmer
Michael Palmer is a principal member of Encore Computer Corporation's technical staff and has led the Parasight team for the past three years. Prior to joining Encore in 1991, Mike worked for several major flight simulation vendors throughout the world, advancing from computer systems engineer to lead software engineer for a $50 million, dual-dome tactical fighter simulator. He has used his real-time simulation background to mold Parasight into a leading tool set for real-time development. Mike holds a B.Sc. (Honors) in electronics from Newcastle Polytechnic, Newcastle upon Tyne, England.

Jeffrey M. Russo
Jeff Russo has been employed by IBM since June 1995. He is an Advisory Programmer working as a team leader for the OS/2 operating system. Prior to joining IBM, Jeff worked at Encore Computer Corporation for 10 years, advancing from the position of software engineer to that of Senior Section Manager responsible for several real-time software groups. He has significant experience with real-time, microkernel-based operating systems, as well as with the accompanying critical, real-time tool set. Jeff earned a B.S. in computer engineering from the University of Florida in 1985.