DCPI for OpenVMS
a Technical Introduction to a "System Microscope "

By Anders Johansson

Anders is a principal software engineer in the kernel tools team of the OpenVMS development engineering group and the project leader for DCPI on OpenVMS. Other projects currently include development work for OpenVMS I64 in the areas of LIBRTL and SDA. Anders, who has been with the company for 16 years and is based in Stockholm, Sweden, is also an OpenVMS Ambassador for Sweden.

Introduction

How many times have you wondered how your application executes, which parts of the system are used most often, or where you might have bottlenecks in the code? Several products are available to measure performance on OpenVMS systems. Most of these products use software performance counters, which have certain limitations.

HP (Digital) Continuous Profiling Infrastructure, DCPI for OpenVMS, uses the hardware performance counters of the Alpha chip to overcome these limitations. DCPI provides a "fine-grained" view of the system. During the analysis of data, DCPI produces information ranging from the time spent in individual executable images to which instructions are executed within an executable image. It also provides insight into where stalls and instruction or data cache misses occur, and so on. A normal sampling frequency for DCPI on a 600MHz Alpha is 10000 samples per second on every CPU in the system. DCPI does this with a minimum CPU overhead -- usually below 5% of the total available CPU time.

DCPI for OpenVMS, Some Background Information

DCPI began as a research project to find out how programs could be optimized to run faster on the Alpha processor. This project, called "Where have all the cycles gone?" resulted in the first version of DCPI (available on Tru64 Unix), and "SRC Technical Note 1997-016A," which is available at the following web site:

http://gatekeeper.research.compaq.com/pub/DEC/SRC/technical-notes/SRC-1997-016a-html/

An investigation into the feasibility of porting DCPI to OpenVMS started in late 1999; most of the porting work was completed during 2000 and early 2001. DCPI was then used within OpenVMS engineering to pinpoint performance problems. Early in 2002, a version of DCPI for OpenVMS became available externally; it is downloadable from the following OpenVMS web site as an "advanced development kit" under field test license terms:

http://h71000.www7.hp.com/openvms/products/dcpi/

DCPI provides the fundamentals for instruction-level system profiling. In general, DCPI does not require any modifications to the code being profiled, because it is driven by the hardware performance counters on the Alpha chip itself. Data is collected on the entire system, including user and third-party applications, runtime libraries, device drivers, the VMS executive itself, and so on. The collected data can then be used for program, routine, and instruction-level profiling of the environment.

DCPI for OpenVMS. How Does It Work?

DCPI consists of the following major components:

1. A data collection subsystem, which includes a device driver, a "daemon program," and a daemon control program (dcpictl) for user intervention with the DCPI daemon.

2. Data analysis tools that are used to break down the collected data into image/routine/code-line/instruction profiles.

3. A special version of the OpenVMS debugger shareable image.

4. A shareable image containing the API for data collection of dynamic code.

The role of the DCPI device driver (DCPI$DRIVER) is to interface with the Alpha chip registers that control the hardware performance monitoring of the chip and to handle the performance monitor interrupts generated when the Alpha chip performance counters overflow. The interrupt handler stores the data acquired at each interrupt into resident memory. The data stored by the driver consists of type of event, PC, and PID.

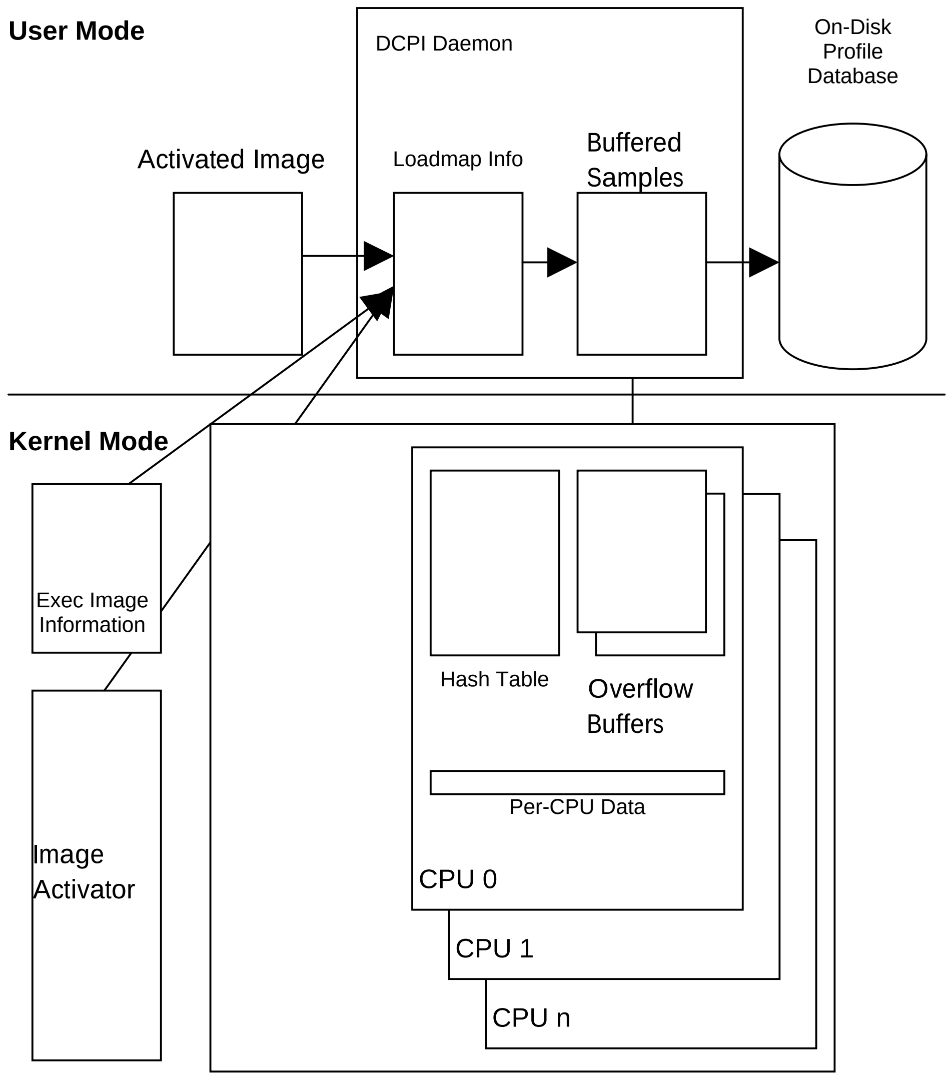
The DCPI Daemon, which runs as an interactive process on OpenVMS, controls the kind of monitoring that is performed and also starts and stops the data collection. However, the main task of the DCPI daemon during the data collection is to:

- Read the data out of the driver buffers
- Map the event/PC/PID into an Imagename/Image-offset pair
- Store it into on-disk profiles for later analysis

To do this mapping, the DCPI daemon must have an in-memory map of the activated images within every process on the system and also a map of the exec loaded image list. The DCPI daemon builds a "loadmap" during its initialization, including the exec loaded image list and the activated images in all the processes currently running on the system. Furthermore, to do the mapping on a running system correctly, the DCPI daemon must track all the subsequent image activations in all the active processes on the system. This image activation tracking is done using "image activator hooks" that are available in the OpenVMS operating system starting with OpenVMS V7.3. This tracking is implemented by means of a mailbox interface between the OpenVMS image activator and the DCPI daemon. In this interface, the image activator provides detailed information to the DCPI daemon about all image activations on the system.

The data analysis tools provide various views of the collected data. For these tools to provide routine names and source code correlations to the profile data, DCPI uses its own version of the OpenVMS debugger shareable image (DCPI$DBGSHR.EXE). To perform routine/source correlations, DCPI also needs images with debug information (LINK/DEBUG) or debug symbol files (.DSF files) for the running images (LINK/DSF). Therefore, while the data collection subsystem collects data on ALL images running on the system, the analysis tools require debug symbol information to perform in-depth analysis of the collected data.

The following figure illustrates DCPI data collection principles.



DCPI Data Collection Principles

Alpha Chip Performance Monitoring

Two different methods exist for collecting performance data using the hardware performance counters of the Alpha chip:

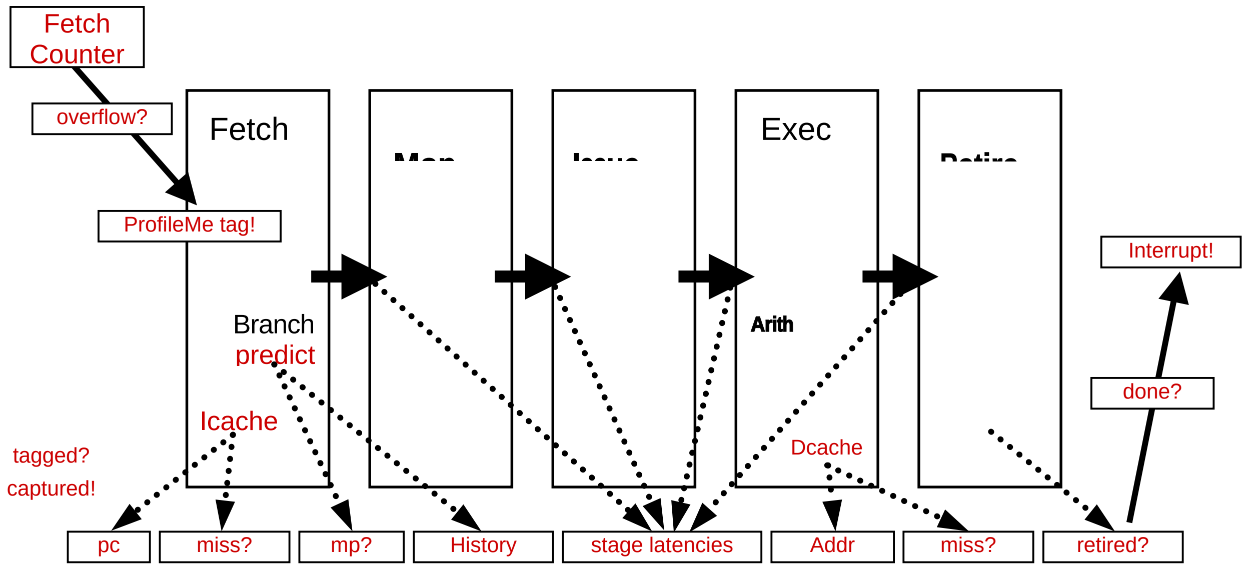
- Aggregate events
  This method is available to a varying degree on all existing Alpha chips. Using this method, DCPI sets a value (sampling period) in the performance monitor control register on the Alpha processor and also specifies which event to sample. Then, each time the event occurs, for example, a “CPU cycle has executed,” this event will be counted. When the number of events has reached the specified sampling period, an interrupt is generated, and DCPI stores the PC, PID and event type. For example: If DCPI is sampling the CPU cycles event with a period of 63488, the Alpha chip generates an interrupt every 63488 cycles.

  Although collecting aggregate events is generally a far better method for obtaining reliable profiling data than collecting software performance counters, collecting aggregate events has certain disadvantages This method relies on the instruction that is active at the time of the interrupt being, in fact, the instruction that generated it -- in other words, it counts on the performance monitor interrupts being precise. This is, however, not always the case. Only a few of the performance monitor events produce a precise interrupt. Also, on recent processors -- EV6 and later -- none of these interrupts are precise. This might appear to be an important problem. However, even though the interrupts are imprecise, they are fairly predictable, which DCPI takes into account.

  Another problem with this method is that it allows for “blind spots” -- for example, any code executing at or above IPL29 will not be profiled, because the performance monitor interrupts are at IPL29. Such blind spots also include all the PAL code on Alpha, since the PAL code runs with interrupts turned off. In those cases, the performance monitor interrupt takes place on the first instruction after the PAL call, or when IPL drops below 29.

- ProfileMe
  This method, which is available on EV67 and upward, is in many ways superior to “aggregate events” ProfileMe uses some specific ProfileMe registers on the Alpha chip. When DCPI sets the period for ProfileMe, the CPU counts instruction fetches. When the period has passed (that is, when the instruction fetch counter overflows), the fetched instruction is tagged as an instruction to profile. Information about this instruction is then recorded into the ProfileMe registers throughout the execution phases of the instruction. When the instruction retires, the interrupt is generated. At this point, DCPI reads all the information out of the on-chip ProfileMe registers. This method of collecting performance data is much more reliable, and also provides a much more complete picture of how the different instructions perform.

The following figure shows the flow of events during ProfileMe sampling.



Internal processor registers

Data Analysis Tools on DCPI for OpenVMS

For the analysis tools to work, they require access to the exact images that were used when the data was collected. The reason is that the tools read the instructions directly from the image files, and the analysis becomes meaningless if the instructions read are not the instructions that were executed. A test is performed that verifies that the image analyzed is the same image as the one being profiled. This verification is performed by checking various fields in the image header and comparing them to what was stored in the DCPI profile database (dcpidb) during the data collection.

The analysis tools can perform breakdown analysis by image or by routine name. To do this successfully, the analysis tools require debug symbol information for the image analyzed. This requirement can be met in two ways:

- The image is linked /DEBUG, which might not be practical, because the image might be INSTALLed on the running system, which requires the image to be linked /NODEBUG

- The image is linked /NODEBUG/DSF, which creates a separate file (*imagename*.DSF) that contains all the debug symbol information for the image. Place this debug symbol file in the same directory as the image file itself. Another alternative is to place all debug symbol files in a separate location and define the logical name DBG$IMAGE_DSF_PATH to point to that directory.

The analysis tools provide slightly different views of the collected data. To perform complete profiling, you must use several of the analysis tools, which are described in the following table:

Tool	Description
DCPIPROF	The top-level analysis tool. It provides either a system-to-image breakdown of the collected samples, or an image-to-procedure breakdown of the collected samples. DCPIPROF is usually the first tool used to obtain an initial idea of the images in which the system is spending time. DCPIPROF is then used to obtain an initial idea of which routines within an image are spending the most time. DCPIPROF analysis works on data collected via the ProfileMe method and via aggregate events.
DCPILIST	The tool to use for detailed analysis within a procedure in a specified image. DCPILIST can correlate the collected samples to either source lines in the code, executed Alpha instructions or both. To do source code correlation, it also needs the actual source file of the analyzed routine. DCPILIST analysis works on data collected via the ProfileMe method and via aggregate events.
DCPICALC	Generates a control flow graph of the procedure or procedures specified for a given image. DCPICALC augments the graph with estimated execution frequencies of the basic blocks, cycle-per-instruction counts, and so on. DCPICALC works only on data collected via aggregate events.
DCPITOPSTALLS	Can be used to identify the instructions within the specified image or images that account for the most stalls. DCPITOPSTALLS only works on data collected via aggregate events
DCPIWHATCG	Generates the same type of control flow graph as DCPICALC, but instead of producing at the procedure level, it looks at different images. DCPIWHATCG works only on data collected via aggregate events.
DCPITOPS	Takes the output from a DCPICALC run and generates a PostScript™ representation of the execution of the image, for example using different font sizes to visualize the execution frequencies of the basic blocks. DCPITOPS only works on data collected via aggregate events.
DCPICAT	Presents the raw profile data in a human readable format. It is normally only used by the DCPI developer, but is included in the DCPI for OpenVMS kit for convenience. DCPICAT can handle all types of DCPI events.
DCPIDIFF	Compares a set of profiles and lists the differences between them. This can be very useful when looking at different test cases.

As indicated in the preceding table, several tools operate only on data that is sampled using aggregate events This somewhat limits the ease of analysis for ProfileMe data, but the richness of the ProfileMe data is sufficient to find all the causes without those tools. DCPICALC, DCPIWHATCG and DCPITOPSTALLS all use intimate knowledge of the Alpha CPU execution characteristics to apply qualified guesswork to find out where problems such as stalls occur. With ProfileMe, all data is collected on each profiled instruction; therefore, the rationale behind not supporting the tools on ProfileMe is that because the data is collected by the hardware itself, it is much more reliable.

Also note that all the DCPI tools use a UNIX-style command syntax. Because adding DCL syntax to the tools would not add anything to the functionality of DCPI, the decision was made to omit the time-consuming effort of providing a DCL interface to the tools.

Profiling of Code Generated "on the Fly"

Some applications such as Java™ generate code on the fly and then execute it. Standard DCPI needs a persistent on-disk image for the analysis because it reads the instructions from that image during the analysis. When building its profile during the data collection, DCPI must also build a loaded image map to calculate image offsets, and so on. None of these exist for dynamically generated code.

DCPI for OpenVMS includes an API for informing the DCPI daemon about the generated code. This API also provides a way to generate Debug Symbol Table (DST) entries for the generated code. The generated code and its associated DSTs are then written to a persistent on-disk "pseudo image," which is used during the analysis. This is an area where DCPI for OpenVMS has evolved beyond the DCPI version available on Tru64 UNIX.

DCPI Usage

A typical sequence of commands to run DCPI data collection is the following:

dcpid, one or more *dcpictl* commands, and finally *dcpictl quit* to stop the data collection.

By using the DCPI daemon in conjunction with the DCPI driver, you first collect data into on-disk profiles, which are stored into epochs on disk. The epochs are the only means of applying a time-line to the DCPI data. This is very important because it is absolutely impossible to see which data in an epoch were collected during, for example, a peak period of the load. The typical recommendation of a way to obtain good results when using DCPI is to keep the load stable within an epoch, because this is the only way to know what is being profiled. Divide any run that includes ramp-up, ramp-down, peak, and low activity on the system into epochs to correctly determine which profiles came from which test case. The names of the profiles come from the GMT time when they were created.

Commands to manipulate epochs during the data collection are:

- *Dcpid* by default creates a new epoch in the current DCPI database. Using the switch *–epoch* on the command line while starting *dcpid* does not create a new epoch, but rather uses the most recent one in the DCPI database.

- *Dcpictl* is an interface to the DCPI daemon during the data collection. Ways to manipulate epochs include the following:

    - *Dcpictl flush,* which performs a user-initiated flush of the in-memory profile data of the DCPI daemon and DCPI driver, into the current epoch in the DCPI database (the logical name DCPIDB points to the DCPI database). Flushing also occurs automatically throughout the data collection and into the current epoch.

o *Dcpictl epoch*, which flushes the in-memory profile data of the DCPI daemon and the DCPI driver into the current epoch, and then starts a new epoch.

Running the Data Collection

A typical way of starting the data collection is:

```
$ dcpid cmoveq$dka100:[dcpi.test]
dcpid: monitoring cycles
dcpid: monitoring imiss
dcpid: logging to comveq$dka100:[dcpi.test]dcpid-COMVEQ.log
```

Because the DCPI daemon runs as an interactive process on OpenVMS, you might want to use the following command to avoid locking up the terminal where *dcpid* is run:

```
$ spawn/nowait/input=nl: dcpid cmoveq$dka100:[dcpi.test]
%DCL-S-SPAWNED, process SYSTEM_187 spawned
dcpid: monitoring cycles
dcpid: monitoring imiss
dcpid: logging to comveq$dka100:[dcpi.test]dcpid-COMVEQ.log
```

On pre-EV67 processors, the default events to collect are *cycles* and *imis.* On EV67 and newer processors, the default events are *pm* (ProfileMe) and *cycles.*

To end the data collection, type the following command:

```
$ dcpictl quit
```

Analyzing the Data

After the data collection is completed (or during the data collection, if data has been flushed) you can then use *dcpiprof* to take an initial look at the collected profile data:

```
$ dcpiprof
dcpiprof: no images specified.  Printing totals for all images.
Column             Total  Period (for events)
------             -----  ------
cycles           1755906   65536
imiss              41991    4096

The numbers given below are the number of samples for each
listed event type or, for the ratio of two event types,
the ratio of the number of samples for the two event types.
===========================================================
 cycles       %    cum% imiss       % image
1349002  76.83%  76.83%  8154  19.42% DISK$CMOVEQ_SYS:[VMS$COMMON.SYSLIB]DECC$SHR.EXE
 176821  10.07%  86.90%   919   2.19% DISK$CMOVEQ_SYS:[VMS$COMMON.SYSLIB]LIBRTL.EXE
  65432   3.73%  90.62%   426   1.01%
DISK$ALPHADEBUG1:[DEBUG.EVMSDEV.TST.TST]LOOPER.EXE;1
  45788   2.61%  93.23%  8651  20.60% SYS$SYSROOT:[SYS$LDR]SYSTEM_SYNCHRONIZATI
  27039   1.54%  94.77%  4598  10.95% SYS$SYSROOT:[SYS$LDR]SYSTEM_PRIMITIVES.EX
  16045   0.91%  95.68%   844   2.01%
DISK$CMOVEQ_SYS:[VMS$COMMON.SYSEXE]DCPI$DAEMON.EXE;2
   7727   0.44%  96.12%  1969   4.69% SYS$SYSROOT:[SYS$LDR]RMS.EXE;
   6993   0.40%  96.52%  2102   5.01% SYS$SYSROOT:[SYS$LDR]SYS$PEDRIVER.EXE;
   6741   0.38%  96.91%  1762   4.20% SYS$SYSROOT:[SYS$LDR]PROCESS_MANAGEMENT_M
   6587   0.38%  97.28%  1215   2.89% SYS$SYSROOT:[SYS$LDR]F11BXQP.EXE;
   5742   0.33%  97.61%  1079   2.57% SYS$SYSROOT:[SYS$LDR]SYS$BASE_IMAGE.EXE;
   5385   0.31%  97.92%  1434   3.42% SYS$SYSROOT:[SYS$LDR]SYS$EWDRIVER.EXE;
   5344   0.30%  98.22%  1371   3.26% SYS$SYSROOT:[SYS$LDR]IO_ROUTINES_MON.EXE;
   5015   0.29%  98.51%  1024   2.44% unknown$MYNODE
```

This first example shows the output of the top-level *dcpiprof* run. The next step is to decide which image is interesting, and use *dcpiprof* to look into that image.

```
$ dcpiprof DISK$ALPHADEBUG1:[DEBUG.EVMSDEV.TST.TST]LOOPER.EXE;1
Column            Total  Period (for events)
------            -----  ------
cycles            84210   65536
imiss               540    4096

The numbers shown below are the number of samples for each
listed event type or, for the ratio of two event types, the
ratio of the number of samples for the two event types.
===========================================================
cycles       %    cum% imiss       % procedure                image
 65774  78.11%  78.11%   334  61.85% get_next_random          disk$alphadebug1..
 16892  20.06%  98.17%    93  17.22% analyze_samples          disk$alphadebug1..
  1543   1.83% 100.00%   112  20.74% collect_samples          disk$alphadebug1..
     1   0.00% 100.00%     1   0.19% main                     disk$alphadebug1.
```

Usually, you perform the next level of analysis by using *dcpilist* to look at the actual code lines/Alpha instructions that the samples are attributed to:

```
$ dcpilist –both -f dbg$tstevmsdev:[tst]looper.c get_next_random -
disk$alphadebug1:[debug.evmsdev.tst.tst]looper.exe
cycles imiss
     0     0          static int get_next_random (void)
     0     0          /*
     0     0          ** We want to get the next random number sample.
     0     0          ** The samples are SAMPLE_ITERATIONS calls apart.
     0     0          */
     0     0          {
     0     0           long int   i;
     0     0           int        sample;
    21     0
    21     0  0x2045c       STL             R31,#X000C(FP)
 42355   174           i = 0;
  3875    27  0x20460       LDL             R1,#X000C(FP)
 11536    31  0x20464       LDA             R1,#XFF9C(R1)
  3923    19  0x20468       LDL             R0,#X000C(FP)
 12001    50  0x2046c       ADDL            R0,#X01,R0
  3764    13  0x20470       STL             R0,#X000C(FP)
  3772    21  0x20474       BGE             R1,#X000006
                      . . . .
  3484    13  0x2048c       BR              R31,#XFFFFF4
 22013                 while (i++ < SAMPLE_ITERATIONS)
  4248    19  0x20478       BIS             R31,R31,R25
    37     0  0x2047c       LDQ             R26,#X0028(R2)
  3689    17  0x20480       LDQ             R27,#X0030(R2)
  7325    28  0x20484       JSR             R26,(R26)
  6714    38  0x20488       STL             R0,#X0008(FP)
     0     0             sample = rand ();
   195     2  0x20490       LDL             R0,#X0008(FP)
     0     0  0x20494       BIS             R31,FP,SP
    40     0  0x20498       LDQ             R26,#X0010(FP)
    44     1  0x2049c       LDQ             R2,#X0018(FP)
    85    37  0x204a0       LDQ             FP,#X0020(FP)
     0     0  0x204a4       LDA             SP,#X0030(SP)
     0     0           return (sample);
$
```

The real challenge with this detailed information is to understand why the system executes as it does, and why certain routines are used as often as they are. Then you need to look into the routines that need to be used frequently if they appear to have performance problems.

In the preceding examples, the top image is not LOOPER.EXE -- which might be the obvious guess, because a *monitor system* would show that the process running LOOPER.EXE is the top CPU consumer. The top image is, rather, the DECC runtime library. The *rand()* call in the *get_next_random()* routine calls into the DECC runtime library, which most likely also uses one or more routines in LIBRTL, thus having those two as top images. This example shows, in a fairly simple way, one reason why getting to the root cause of a problem might be a challenge. The cause of the "problems" seen here is LOOPER.EXE, because it makes excessive calls into the DECC runtime library. From the initial *dcpiprof,* believing that the DECC runtime library has problems is easy. Although this example is simplistic, it demonstrates that further analysis is often needed to find the root cause of the system behavior.

Some Basic Hints for DCPI Analysis

A few hints for DCPI analysis follow. To perform a full analysis of an application requires a very good understanding of the application itself.

- As indicated above, the DCPI notion of a time-line is called an epoch. No way of "time-stamping" the individual samples exists, other than creating a new epoch. To obtain predictable results when using DCPI, you must have as a goal a stable load throughout an epoch. No way exists to analyze less than an epoch afterwards. Creation of epochs is done during data collection.

- DCPI samples the whole OpenVMS system, not just the "interesting" program. Calls that are made into shareable images reflect the sum of all the calls made by ALL programs currently running on the system. A way to break this up into images run within different processes is to start the data collection with *$dcpid –bypid 'imagename'.*

- Be careful when drawing conclusions. As the above example illustrates, drawing incorrect conclusions is quite easy. Also, a high number of samples in an image/routine might not mean that this image/routine has any performance problems. A high number of samples means only that the image/routine was used frequently. You need to analyze further to find the root cause of the sample count.

- The DCPI daemon, which is a central piece of the DCPI data collector, is a normal user process. On a heavily loaded system, the DCPI daemon could be starved for CPU time, which might be seen as "dropped samples" in the DCPI daemon log file (dcpid-*nodename*.log) in the DCPI database. The DCPI database is defined by the logical name *dcpidb.* In some cases, it is important to start the data collection with a *dcpid –nice 'priority'* to increase the priority of the DCPI daemon process.

- The DCPI daemon scans all available processes during its initialization, to build an image map for the images in each process. On a system with many processes and a high load, this initial scan can take a considerable amount of time. If the system is running OpenVMS V7.3 or higher, starting the DCPI daemon ahead of time, when the system is relatively idle greatly reduces the time of the DCPI daemon initialization.

- All activity on the system, or lack of activity, is reflected in the collected data. If the system is idle, nearly 100% of the time will be attributed to SCH$IDLE() in PROCESS_MANAGEMENT.EXE. This exec loaded image contains other important code; therefore, a high number of samples might indicate something else. If the percentage of samples in PROCESS_MANAGEMENT.EXE is similar to the percentage of idle time, it is fairly safe to make this assumption.

- Try to minimize the noise level during the data collection by stopping unused CPUs or unneeded software, or both. Do this only during a second-level data collection, when you are narrowing down the causes of a problem.

A high number of cycles might be normal for the images or routines seen during analysis. When using ProfileMe, to find images or routines with possible problems, look at the RETIRED/CYCLES ratio of the routine. Ideally, the Alpha chip is capable of sustaining 4 instructions per cycle. Any routine with a ratio of 3 is probably impossible to improve, while a routine with a ratio below 1 is a good suspect for a routine with performance problems.