NEXTSTEP

Title: Using sampler as a tool for profiling and performance tuning

Entry Number: 1898
Last Updated: 13 Apr 1995

Keywords: profiling, performance tuning, memory leak, statistical sampling

Questions

Q: I want to do some performance tuning of my EOF app, but there are no profiling libraries in the 1.1 Release of the Enterprise Objects Framework. The Release Notes mention that NeXT plans to provide a profiling tool via NeXTanswers. Where is the tool?

Answers

A: The tool is named **sampler** and is available as NeXTanswer #1899

sampler is based on statistical sampling, which means that the program's call stack is stashed away at pseudo-random intervals for further analysis. Each sample consists of a snapshot of the stack in a very compact form, and samples are collected in a temporary file (one file per thread, actually). A companion analysis program, **analyze**, given these sample files and the path of the executable, produces various analysis on the data. Because **sampler** reflects actual user time rather than CPU time, time spent in inter process communication, system calls, or even paging shows up in the analysis. Also, sampling a program does dot change the behavior of the program (performance-wise) except by slowing down the entire machine by a few percent. In most situations, **sampler** is therefore a more appropriate tool for improving responsiveness than **gprof**, which only

measures the CPU time, and is highly affected by the Heisenberg principle.

An unrelated utility, **leaks**, can find memory leaks using a similar technique than the one used in MallocDebug, but is very lightweight and does not require relinking.

To use **sampler**:

1) Install the binaries in a convenient location, for example, your ~/bin directory.
2) Make sure that the application you are sampling is currently running
3) Generate a *samples* file. The command syntax is:

```
sample <pid> <sampling time (secs)> { <msecs between samples> }
```

The last parameter specifies the time elapsed between snapshots of the call frame stack and is optional (the default interval is 50 msecs between samples, which should be fine for most applications. On a fast machine you may want to decrease this interval).

As an example, to sample the Workspace, you can do:

```
Assuming that Worskpace <pid> is 1280
localhost> sample 1280 20
Sampling 1280 (1 threads) each 50 msecs 800 times
Samples for thread 0 in /tmp/pid1280_0.samples: 22984 bytes
Average delta between samples: 59 msecs                 max: 136 msecs
```

4) Analyze the samples file generated in /tmp/pid1280_0.samples. The command syntax is:

```
analyze <samples_file> <symbol_file> {<filter>}
```

The symbol_file is your program executable. You can indicate the executable that is

running, or a version with symbols (obtained using the **-g** compile option). The filter, optional, can be used to indicate sole interest in stacks containing a certain symbol.

To continue with the previous example of sampling the Workspace, you can do:

```
localhost> analyze /tmp/pid1280_0.samples /usr/lib/NextStep/Workspace.app/WM.app/WM
> /tmp/analysis_0
```

This will create an ASCII analysis file that you can view and analyze.
Using the Edit application, you can first contract the output, i.e., Format->Structure->Contract All (Cmd-0), to see what the different sections are, like:

```
Analyzing sampling of /usr/lib/NextStep/Workspace.app/WM.app/WM
Call graph
Sort by top of stack:
Sort by top of stack, same collapsed:
```

Analysis of the Sort by top of stack outputs

The outputs in the two sections "Sort by top of stack" are similar, except that same functions have been collapsed to one single output line in the section "Sort by top of Stack, same collapsed".
This last output is very useful to do a general analysis of the time spent in your application.
Here is a sample output:

```
Sort by top of stack, same collapsed:
msg_receive_trap (top of stack)572
msg_send_trap (top of stack)   63
lstat (top of stack)  38
stat (top of stack)   19
msg_rpc_trap (top of stack)    16
```

```
objc_msgSend (top of stack)    9
NXMapGet (top of stack)    4
ur_cthread_self (top of stack)  4
statfs (top of stack)  3
syscall (top of stack) 3
```

In the case of this sampling obtained while browsing the file system with Workspace, one could see that the majority of the time is spent in msg_receive_trap, which is typical of waiting for an event. msg_send_trap and msg_rpc_trap usually denote transmission to the WindowServer. lstat, stat, statfs, readdir are Unix system calls. So in this example, out of 228 useful samples (800 - 572), 79 (63 + 16) had the process waiting for the window server, 63 (38 + 19 + 3 + 3) had the process waiting to get some information from the file system, and 9 were due to ObjC messaging. These numbers are fairly typical, with interprocess communication taking the lion's share of the elapsed time - and with the Objective C runtime only playing a minor role. Note that because sampling is statistic, fewer than 5 samples is usually not very significant.

Analysis of The Call Graph

Sample output:

```
Total number in stack (recursive counted multiple):
Call graph:
    start: 800
     main: 800
      -[Application run]: 800
       NXGetOrPeekEvent: 574 (71%)
_DPSGetOrPeekEvent: 574
 msg_receive_trap: 459 (79%)
    msg_receive_trap (top of stack): 459
 checkTEs: 96 (16%)
    setCPath: 79 (82%)
     -[Object perform:with:]: 79
      fileInfoZone: 58 (73%)
```

```
setGlobalTS: 58
 setGlobalTS: 58
  setGlobalTS: 58
   fileInfoZone: 58
istringToSEL: 37 (63%)
    istringToSEL: 36 (97%)
    normalizePath: 16 (44%)
        splitPath: 4 (25%)
         NXHashTableUniqueElement: 4
          +[List new]: 2 (50%)
             +[List newCount:]: 2
              +[Object allocFromZone:]: 2
               _internal_class_createInstanceFromZone: 2
                memset: 1 (50%)
             memset (top of stack): 1
                nxzonemalloc: 1 (50%)
             nxzonemallocnolock: 1
              nxzonemallocnolock (top of stack): 1
          splitPath: 1 (25%)
             -[Object perform:with:]: 1
```

Recursive counted multiple means that recursive functions are listed multiple times as they are called within the call frame stack.

5) There is another way to look at the call frame stack interactively by using the **Sampler.app** provided with this sampler package. You need to provide the pid, the time interval (in msecs) and the location of your executable when running this app. As you start running your program, the call frame stack will change dynamically as snapshots are taken, and the functions that are called constantly will stay black, while functions that change depending on your operations will be grayed out. You can also determine which function is taking a really long time (such as the connect operation in a database application).
At any time, the **Stop** button will give you a frozen view of the current snapshot. **Sampler.app** can be used to understand coarse problems (such as timeouts).

6) The sampler package also provides a lightweight tool to detect memory leaks. The command syntax is:

```
leaks <pid>1 ... <pid>n
```

This leak detector is not as sophisticated as MallocDebug, since it doesn't give a backtrace of the methods that eventually produce the leaks, but it can be used to determine problem areas in your application without the need to link with libMallocDebug.a.

Note: Only Intel and Motorola architectures are currently supported by sampler.

Valid for: EOF 1.0, EOF 1.1, NEXTSTEP 3.2 Developer, NEXTSTEP 3.3 Developer.

See Also:

/NextLibrary/Documentation/NextDev/Concepts/Performance