Measuring Performance and Memory Usage

Measuring Performance

By just using your application you may already have a sense of where the problem liesÐfor example, load times are too long, redraw time is too long, or maybe response time is sluggish. In fact, this information is organized along those lines since specific techniques are often useful in solving each of those problems. However, there are a number of tools that can be used to get a qualitative and quantitative sense of where the cycles are being consumed within your application. The purpose of this section is to elaborate on them.

gprof

The standard profiling tool in the UNIX®world is **gprof**, and it's an excellent tool for understanding where in your process the cycles are being consumed. It generates a call graph for your application showing how many times a given function was called, and how much time was spent in that function and the function's descendents (that is, functions called by the given function). By analyzing the output from **gprof** you can quickly identify those areas where it makes sense to focus your optimization efforts. To use **gprof**:

- Compile the application using **make profile**. This compiles your program with the **-pg** option, which indicates profiling and generates an executable file named **appName.profile**.

- Run the application.
- When you exit, a file named **gmon.out** will be placed in your home directory.
- Run **gprof** to create the profile:

```
gprof appName.profile/appName gmon.out >profile
```

- Use your favorite editor to examine the profile file.
- Or if you want to print the file, type

```
gprof appName.profile/appName gmon.out |enscript
```

See the UNIX manual page on **gprof** for an explanation of the output of **gprof** and its many options.

However, there's one extremely important caveat with **gprof**: It does not measure the time spent by the Window Server on your behalf, because **gprof** only measures time spent within a given process. Effectively this means that **gprof** is not very useful in measuring time spent drawing, which is a serious problem because drawing time often accounts for the bulk of the total time. You must use other techniques to analyze the drawing time.

Qualitatively Measuring Drawing Performance

There are two command-line arguments that can be used to quickly get a qualitative sense of your drawing performance: NXAllWindowsRetained and NXShowPS.

NXAllWindowsRetained

The command-line argument NXAllWindowsRetained is extremely useful for identifying situations in which you're

doing unnecessary drawing. NXAllWindowsRetained makes all your windows visible, and they're made retained so all drawing is done directly on the screen. Hence, by visual inspection, you can see all of the drawing that's going on in your application. NXAllWindowsRetained is often a good first step in diagnosing performance problems. To use NXAllWindowsRetained, launch your application from a Terminal window, giving NXAllWindowsRetained as an argument:

```
appName -NXAllWindowsRetained YES
```

NXShowPS

The command-line argument NXShowPS is a useful aid in diagnosing inefficient drawing, because it writes all the Display PostScript®code going to the Window Server out to **stderr** as well. While use of NXAllWindowsRetained will diagnose unnecessary drawing, use of NXShowPS will diagnose inefficient drawing. While the volume of PostScript code is often a bit overwhelming, you should look for things like multiple erases, clips, and so on. To use this option, launch your program from a Terminal window, giving NXShowPS as an argument:

```
appName -NXShowPS YES
```

For finer-grain inspection, use the **showps** and **shownops** commands within GDB to accomplish the equivalent of NXShowPS for selected portions of your application. In practice, this is more useful than using NXShowPS for your entire application, since you're typically interested only in the PostScript code being emitted as a result of a particular method.

Quantitatively Measuring Drawing Performance

To get hard numbers on the time being consumed by an application in preparation for drawing, or by the Window Server on your behalf when it's executing drawing commands, you must use interval timing. There are two standard techniques:

- Bracket the relevant code with calls to **gettimeofday()** and calculate the elapsed time. Note that this will represent the ªwall time,º which is, after all, what the user cares about; it'll account for time spent within your application as well as within the Server.
- Bracket the relevant code with calls to **getrusage()** and **PSusertime()** to calculate the elapsed CPU time spent within your process, and the time spent within the Server on behalf of your application's PostScript context.

To illustrate these two techniques, we've provided you with a simple Timing class that provides methods that implement the two techniques (see **ListingsForTimingClass.rtf**). To use the Timing class, you create an instance of the class, send a message to the object when you want to begin timing, and then send another message to it when you want to end timing. It'll keep track of the number of times the timing interval has been entered and the cumulative elapsed time.

Below are two methods from the Timing class that are used to calculate elapsed wall time.

```
-wallEnter
{
    cumTimesEntered++;
    NXPing();
    gettimeofday(&realtime,&tzone);
    synctime = realtime.tv_sec +realtime.tv_usec/1.0E6;
    return self;
}
```

```
-wallLeave
{
    double eTime;
    NXPing();
    gettimeofday(&realtime,&tzone);
    eTime = (-synctime + realtime.tv_sec
                + realtime.tv_usec /1.0E6) - tare;
    cumWallTime += eTime;
    return self;
}

-tare
{
    struct timezone tzone1;
    struct timeval realtime1;
    struct timeval realtime2;
    NXPing();
    gettimeofday(&realtime1,&tzone1);
    NXPing();
    gettimeofday(&realtime2,&tzone1);
    tare = (-realtime1.tv_sec + realtime2.tv_sec) +
        (-realtime1.tv_usec+ realtime2.tv_usec)/1.0E6;
    return self;
}
```

This approach is reasonably accurate, assuming the following:

- There are no other active processes on the machine (usually true).
- The unit of work being done is large enough that the round-trip overhead doesn't consume most of the time. The purpose of **tare** is to account for the round-trip time and measurement latency.

- You're not seriously affected by the timing variance this puts into your program (that is, don't do this in production code!)

The following two methods illustrate the approach of using **getrusage()** and **PSusertime()** to calculate the time spent within your process, and the time spent within the Server on behalf of your application's PostScript context.

```
-psEnter
{
    cumTimesEntered++;
    PSusertime(&stime);
    getrusage(RUSAGE_SELF,&rtime);
    synctime = (rtime.ru_utime.tv_sec +
        rtime.ru_stime.tv_sec) +
        (rtime.ru_utime.tv_usec +
        rtime.ru_stime.tv_usec)
        /1.0E6;
    return self;
}

-psLeave
{
    int et;
    double appTime;
    double psTime;
    getrusage(RUSAGE_SELF,&rtime);
    appTime  = ((rtime.ru_utime.tv_sec +
        rtime.ru_stime.tv_sec) +
        (rtime.ru_utime.tv_usec +
        rtime.ru_stime.tv_usec)
        /1.0E6) -synctime;
```

```
        cumAppTime += appTime;
        PSusertime(&et);
        psTime = ((et-stime)/1000.0);
        cumPSTime += psTime;
        return self;
    }
```

In this case, **appTime** represents the elapsed CPU time spent in the process and in the system on behalf of the process, and **psTime** represents the CPU time spent in the Server on your behalf. The equation (**psTime**/(**appTime** +**psTime**)) will give you the percentage of the time spent in the Server.

The second approach is very useful for seeing the breakdown between application time and Server time within a chunk of code. For example, you can use this to answer questions like, ªWhile wrapping and drawing a page of text, how much time is spent in the Server and how much time in the application?º You can then decide whether to go after the Server time with **showps** and timing your graphics operations, or go after the application time with **gprof**.

Either technique is extremely useful in understanding where the time is going, so you can then focus on solving the relevant problem. With either approach you should be sure to average over many trials to get an accurate value. You should also be sure to remove all timing code from production code.

Measuring Memory Usage

When measuring memory usage, you need to measure not only the amount of memory consumed by your process, but also the amount of memory consumed by the Window Server on your application's behalf. The tools described in this section allow you to measure both.

ps

The UNIX utility **ps** is useful for only a gross measure of how much memory is consumed by your process. Because of the Mach virtual memory model, the numbers reported by **ps** are extremely inflated. For example, every program has a large stack allocated to it, but only the pages that are actually touched ever come into existence in RAM or in the swapfile. They're created zero-filled on demand. However, the total amount of virtual memory allocated for your stack shows up in your virtual process size. The shared libraries also appear in your virtual and resident sizes. Mapped files can also increase virtual sizes without increasing the resident size appreciably. The best use of the virtual size is finding substantial VM leaks. For most other purposes, resident size is what counts.

The **ps** utility is mostly useful in diagnosing large memory leaks. By running **ps -ux** in a Terminal window at periodic intervals, and comparing the virtual and resident sizes of your application at the two points in time, you may be able to identify large memory leaks within your application.

size

The UNIX utility **size** is useful for determining the size of the various portions of your executable file (text, data, interface segments, and so on) on disk. For more information, see the UNIX manual page for **size**.

MallocDebug

The **MallocDebug** application is the best source for detailed information on your application's use of the malloc

heap. **MallocDebug** lets you dynamically view a list of all the heap data your application has allocated, sorted by size, zone, allocation time or the stack backtrace at the time of allocation. **MallocDebug** lets you find memory leaks with the press of a single button. This tool also lets you see which heap items are accessed for a particular operation, so that you can divide your data into zones for better locality of reference. For more information, see **/NextLibrary/Documentation/NextDev/DevTools/08_MallocDebug**.

Winfo

The application **Winfo** is an extremely useful tool for measuring memory consumed by your application in the form of window backing stores within the Window Server. **Winfo** graphically shows you all the windows your app has created, both on-screen and off-screen, and how much memory those windows consume. In addition, windows that have been promoted to having color or alpha are highlighted for easy detection. The application is available from NeXT technical support and on various archive servers.

-NXShowAllWindows

To measure how your application uses off-screen windows, run your application with the option "-NXShowAllWindows YES". This forces all the off-screen windows to be visible on screen. Look for redundant bitmap caches, or caches for images you rarely use.