

Performance Tuning Self-Teaching Guide

*by Mai Nguyen, NeXT Developer Support
and Randy Nelson, NeXT Developer Training*

Overview

This document is designed to provide hands-on experience with some performance tuning tools, based on an example called VisibleView from **/NextDeveloper/Examples**. The example has evolved into three versions, with the original version containing most of the performance flaws, and the final version showing the corrections to these flaws.

The document is organized as follows:

- Using **gprof** with VisibleView-01
- Using the Process Monitor
 - Measuring Drawing Performance
- Finding Memory Leaks
- Other Steps to Improve VisibleView-01
- References

Using gprof with VisibleView-01

Caveat: The **gprof** tool only measures time spent within a given process. It does not measure the time spent by the Window Server on your behalf. Please refer to the Section ^aQuantitatively Measuring Drawing Performance^o for these measurements.

In the following step-by-step description, we will use **gprof** to locate areas in VisibleView that need optimization.

Step 1: Generate an application ready for profiling

Add a Makefile.preamble which includes a macro:

```
PROF_LIBS = -lNeXT_s -lsys_s
```

This will override the default profiling libraries **-lNeXT_p** and **-lsys_p** which profile all the methods called by Appkit. This will limit the profiling output to the methods used by the application itself only.

Compile the application in a Shell with **make profile**, which causes the application to be compiled with the **-pg** option, and generates an executable called **VisibleView.profile**.

Step 2: Run the application and generate profiles

In a shell window type:

```
localhost> visibleOne.profile
```

A possible way to profile the application is to generate a profile for a specific set of operations, such as:

- Start the program and stop it to profile the minimum set of procedures for the program initialization. Save this profile.

```
localhost> gprof visibleOne.profile gmon.out > startUpProfile
```

- Perform all operations that the application can do, such as going through the Tools Menu, bringing up the Coordinates, Frame, Statistics, Appearance Panels, etc. Save this profile.

```
localhost> gprof visibleOne.profile gmon.out > commonOpsProfile
```

Note: After each program run, a file **gmon.out** will be generated automatically, overwriting the old file. To keep different profiles, you should rename this **gmon.out** file before running a new profiling pass. Or you can generate a profile file first before starting a new run. Another way to profile is to let a tester use the program for a few days, then use the profiles generated.

You can also turn profiling ON and OFF from your code, with the function **moncontrol()**. This allows to measure the cost of particular operations. To stop the collection of histogram ticks and call counts, use **moncontrol(0)**. To resume the profiling, call **moncontrol(1)**. The output file **gmon.out** will be produced upon program exit regardless of the state of **moncontrol**. See the Unix man pages for more information.

Step 3: Analyze the output

The profile output consists of three main parts:

Part 1. The call graph profile gives detailed information about each function called during the program execution, about its parent functions (the callers), and about its children (the callees).

Part 2. The flat profile lists all the functions with useful timing information (such as average time spent for the function itself,

how many times the function gets called, etc.).

Part 3. The index section lists all the function names and their corresponding index for quick reference.

We are mostly interested in **Part 1** and **Part 2** for this small application. **Part 3** can be useful if you have many function calls to trace in your program.

Sample Output for Part 1 (Call Graph Profile):

granularity: each sample hit covers 4 byte(s) for 33.33% of 0.05 seconds

index	%time	self	descendents	called/total	parents
				called+self called/total	name index children
[1]	100.0	0.00	0.05	125/125	_exit [3]
		0.00	0.05	125	-[VisibleOne drawSelf::] [1]
		0.00	0.05	125/125	-[VisibleOne drawSomePS] [2]
		0.00	0.00	125/125	-[VisibleOne updateVitals] [12]
		0.00	0.00	124/124	-[SubViewFramer frameMe:] [13]
		0.00	0.00	122/122	-[VisibleOne drawGrid] [15]

[2]	100.0	0.00	0.05	125/125	-[VisibleOne drawSelf::] [1]
		0.00	0.05	125	-[VisibleOne drawSomePS] [2]
		0.00	0.04	375/375	-[VisibleOne mass] [4]
		0.00	0.01	125/500	-[VisibleOne rec] [5]
		0.00	0.00	125/500	-[VisibleOne tri] [8]
		0.00	0.00	125/500	-[VisibleOne cir] [10]

[3]	100.0	0.00	0.05		_exit [3]
		0.00	0.05	125/125	-[VisibleOne drawSelf::] [1]
		0.00	0.00	37/37	-[VisibleOne setSelfOrigin:] [19]
		0.00	0.00	37/37	-[VisibleOne frameMove:] [18]
		0.00	0.00	20/20	-[VisibleOne setSelfRotation:] [20]
		0.00	0.00	14/14	-[VisibleOne frameRotate:] [21]
		0.00	0.00	11/11	-[VisibleOne frameChangeSize:] [22]
		0.00	0.00	4/4	-[VisibleOne mouseExited:] [25]
		0.00	0.00	4/4	-[VisibleOne mouseEntered:] [24]
		0.00	0.00	4/4	-[VisibleOne hitTest:] [23]
		0.00	0.00	3/3	-[VisibleOne newDisplayState:] [26]
		0.00	0.00	1/1	-[VisibleOne mouseUp:] [33]
		0.00	0.00	1/1	-[VisibleOne mouseDown:] [32]
		0.00	0.00	1/1	-[VisibleOne acceptsFirstResponder] [30]
		0.00	0.00	1/1	-[VisibleOne setSelfScale:] [43]
		0.00	0.00	1/1	-[VisibleOne appDidInit:] [31]
		0.00	0.00	1/1	-[VisibleOne setAppender:] [34]
		0.00	0.00	1/1	-[Appender setScrollView:] [29]
		0.00	0.00	1/1	-[VisibleOne setTranslateX:] [44]
		0.00	0.00	1/1	-[VisibleOne setTransparencyIndicator:] [46]
		0.00	0.00	1/1	-[VisibleOne setVitalMatrix:] [47]
		0.00	0.00	1/1	-[VisibleOne setTranslateY:] [45]
		0.00	0.00	1/1	-[VisibleOne setScaleY:] [42]
		0.00	0.00	1/1	-[VisibleOne setScaleX:] [41]
		0.00	0.00	1/1	-[VisibleOne setDrawGridIndicator:] [36]
		0.00	0.00	1/1	-[VisibleOne setCompositeIndicator:] [35]
		0.00	0.00	1/1	-[VisibleOne setFrameWidth:] [38]
		0.00	0.00	1/1	-[VisibleOne setFrameHeight:] [37]
		0.00	0.00	1/1	-[VisibleOne setFrameX:] [39]
		0.00	0.00	1/1	-[VisibleOne setFrameY:] [40]
		0.00	0.00	1/1	+ [VisibleOne newFrame:] [28]
		0.00	0.00	1/1	_main [54]

		0.00	0.04	375/375	-[VisibleOne drawSomePS] [2]
[4]	75.0	0.00	0.04	375	-[VisibleOne mass] [4]
		0.00	0.02	375/500	-[VisibleOne rec] [5]
		0.00	0.01	375/500	-[VisibleOne tri] [8]
		0.00	0.00	375/500	-[VisibleOne cir] [10]

		0.00	0.01	125/500	-[VisibleOne drawSomePS] [2]
		0.00	0.02	375/500	-[VisibleOne mass] [4]
[5]	66.7	0.00	0.03	500	-[VisibleOne rec] [5]
		0.03	0.00	500/500	-[VisibleOne recfill] [6]
		0.00	0.00	500/500	-[VisibleOne recstroke] [11]

		0.03	0.00	500/500	-[VisibleOne rec] [5]
[6]	66.7	0.03	0.00	500	-[VisibleOne recfill] [6]

		0.02	0.00	1000/1000	-[VisibleOne tri] [8]
[7]	33.3	0.02	0.00	1000	-[VisibleOne tripath] [7]

		0.00	0.00	125/500	-[VisibleOne drawSomePS] [2]
		0.00	0.01	375/500	-[VisibleOne mass] [4]
[8]	33.3	0.00	0.02	500	-[VisibleOne tri] [8]
		0.02	0.00	1000/1000	-[VisibleOne tripath] [7]

[9]	0.0	0.00	0.00	1000/1000	-[VisibleOne cir] [10]
		0.00	0.00	1000	-[VisibleOne cirpath] [9]

		0.00	0.00	125/500	-[VisibleOne drawSomePS] [2]
		0.00	0.00	375/500	-[VisibleOne mass] [4]
[10]	0.0	0.00	0.00	500	-[VisibleOne cir] [10]
		0.00	0.00	1000/1000	-[VisibleOne cirpath] [9]

		0.00	0.00	500/500	-[VisibleOne rec] [5]
[11]	0.0	0.00	0.00	500	-[VisibleOne recstroke] [11]

		0.00	0.00	125/125	-[VisibleOne drawSelf::] [1]
[12]	0.0	0.00	0.00	125	-[VisibleOne updateVitals] [12]

		0.00	0.00	124/124	-[VisibleOne drawSelf::] [1]
[13]	0.0	0.00	0.00	124	-[SubViewFramer frameMe:] [13]
		0.00	0.00	124/124	-[VisibleOne wantsTransparency] [14]

		0.00	0.00	124/124	-[SubViewFramer frameMe:] [13]
[14]	0.0	0.00	0.00	124	-[VisibleOne wantsTransparency] [14]

		0.00	0.00	122/122	-[VisibleOne drawSelf::] [1]
[15]	0.0	0.00	0.00	122	-[VisibleOne drawGrid] [15]

		0.00	0.00	1/69	-[VisibleOne appDidInit:] [31]
		0.00	0.00	1/69	-[VisibleOne acceptsFirstResponder] [30]
		0.00	0.00	1/69	-[VisibleOne mouseDown:] [32]
		0.00	0.00	1/69	-[VisibleOne mouseUp:] [33]
		0.00	0.00	4/69	-[VisibleOne mouseEntered:] [24]
		0.00	0.00	4/69	-[VisibleOne mouseExited:] [25]
		0.00	0.00	8/69	-[VisibleOne hitTest:] [23]
		0.00	0.00	49/69	-[VisibleOne setTrackingRect] [17]
[16]	0.0	0.00	0.00	69	-[Appender appendToText:] [16]

		0.00	0.00	1/49	-[VisibleOne appDidInit:] [31]
		0.00	0.00	11/49	-[VisibleOne frameChangeSize:] [22]
		0.00	0.00	37/49	-[VisibleOne frameMove:] [18]
[17]	0.0	0.00	0.00	49	-[VisibleOne setTrackingRect] [17]
		0.00	0.00	49/69	-[Appender appendToText:] [16]

This is a sample of the call graph profile generated after performing the following operations:

- Start up the program.
- Bring up all the inspector panels.
- Do some drawing operations, such as compositing, bringing up the grid, rotation, translation, scaling, etc.

The leftmost function within an index is the major function under analysis. The functions above it are the parent functions. The functions below it are the descendent functions.

For example, in the case of index [1], **drawSelf::** is the major function. Its parent function is **_exit::**.

%time is only indicated for the major function. It's the percentage of the total time of the program accounted for by this function and its descendents.

For an example of how the major function interacts with its parents and descendents, look at the output for index [17], which shows that **[VisibleOne setTrackingRect]** was called 49 times. It was called 1 time by **[VisibleOne appDidInit:]**, 11 times by **[VisibleOne frameChangeSize:]**, and 37 times by **[VisibleOne frameMove:]**. The number of times it gets called by the particular function is the numerator of the fraction (1, 11, 37); the denominator represents the total number of times it's being called. For the descendents, **appendToText:** was called by **setTrackingRect:** 49 times out of a total of 69. Looking at index [16] will give the full break-out on function calls for **appendToText:**.

The **self** and **descendents** columns indicate the time usage break-out for self and its descendents. For example, in the case of index[2], even though the time spent by **drawSomePS** is listed as 0.00, its descendents use 0.05 time, split as 0.04 for **[VisibleOne mass]** and 0.01 for **[VisibleOne rec]**. The total of all the times used by self and its descendents must add up as 0.05.

In the above example profile, **_exit**, **drawself::** and **drawSomePS** take up 100% of time. It means that **drawself::** (and its descendents) are the big time consumers, while the remaining methods are rather negligible. When times are in the range of 0.0x, it could be that the Window Server time is much more noticeable than the CPU time recorded by **gprof**.

You can use the call graph profile in conjunction with the flat profile to quickly spot the problem areas. Usually, the relevant information in the call graph profile are listed in the first few pages, since the profile is ordered by the **% time** value.

Similarly, look at the cumulative time in seconds in the flat profile (shown below). The first pages are interesting, until that cumulative time reaches a plateau.

Sample Output for Part 2 (Flat Profile):

granularity: each sample hit covers 4 byte(s) for 33.33% of 0.05 seconds

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
66.7	0.03	0.03	500	0.06	0.06	-[VisibleOne recfill] [6]
33.3	0.05	0.02	1000	0.02	0.02	-[VisibleOne tripath] [7]
0.0	0.05	0.00	1000	0.00	0.00	-[VisibleOne cirpath] [9]
0.0	0.05	0.00	500	0.00	0.00	-[VisibleOne cir] [10]
0.0	0.05	0.00	500	0.00	0.06	-[VisibleOne rec] [5]
0.0	0.05	0.00	500	0.00	0.00	-[VisibleOne recstroke] [11]
0.0	0.05	0.00	500	0.00	0.03	-[VisibleOne tri] [8]
0.0	0.05	0.00	375	0.00	0.09	-[VisibleOne mass] [4]
0.0	0.05	0.00	125	0.00	0.38	-[VisibleOne drawSelf::] [1]
0.0	0.05	0.00	125	0.00	0.38	-[VisibleOne drawSomePS] [2]
0.0	0.05	0.00	125	0.00	0.00	-[VisibleOne updateVitals] [12]
0.0	0.05	0.00	124	0.00	0.00	-[SubViewFramer frameMe:] [13]
0.0	0.05	0.00	124	0.00	0.00	-[VisibleOne wantsTransparency] [14]
0.0	0.05	0.00	122	0.00	0.00	-[VisibleOne drawGrid] [15]
0.0	0.05	0.00	69	0.00	0.00	-[Appender appendToText:] [16]
0.0	0.05	0.00	49	0.00	0.00	-[VisibleOne setTrackingRect] [17]
0.0	0.05	0.00	37	0.00	0.00	-[VisibleOne frameMove:] [18]
0.0	0.05	0.00	37	0.00	0.00	-[VisibleOne setSelfOrigin:] [19]
0.0	0.05	0.00	20	0.00	0.00	-[VisibleOne setSelfRotation:] [20]
0.0	0.05	0.00	14	0.00	0.00	-[VisibleOne frameRotate:] [21]
0.0	0.05	0.00	11	0.00	0.00	-[VisibleOne frameChangeSize:] [22]
0.0	0.05	0.00	4	0.00	0.00	-[VisibleOne hitTest:] [23]
0.0	0.05	0.00	4	0.00	0.00	-[VisibleOne mouseEntered:] [24]
0.0	0.05	0.00	4	0.00	0.00	-[VisibleOne mouseExited:] [25]
0.0	0.05	0.00	3	0.00	0.00	-[VisibleOne newDisplayState:] [26]
0.0	0.05	0.00	1	0.00	0.00	+ [SubViewFramer newFrame:] [27]
0.0	0.05	0.00	1	0.00	0.00	+ [VisibleOne newFrame:] [28]
0.0	0.05	0.00	1	0.00	0.00	-[Appender setScrollView:] [29]

0.0	0.05	0.00	1	0.00	0.00	-[VisibleOne acceptsFirstResponder] [30]
0.0	0.05	0.00	1	0.00	0.00	-[VisibleOne appDidInit:] [31]
0.0	0.05	0.00	1	0.00	0.00	-[VisibleOne mouseDown:] [32]
0.0	0.05	0.00	1	0.00	0.00	-[VisibleOne mouseUp:] [33]
0.0	0.05	0.00	1	0.00	0.00	-[VisibleOne setAppender:] [34]
0.0	0.05	0.00	1	0.00	0.00	-[VisibleOne setCompositeIndicator:] [35]
0.0	0.05	0.00	1	0.00	0.00	-[VisibleOne setDrawGridIndicator:] [36]
0.0	0.05	0.00	1	0.00	0.00	-[VisibleOne setFrameHeight:] [37]
0.0	0.05	0.00	1	0.00	0.00	-[VisibleOne setFrameWidth:] [38]
0.0	0.05	0.00	1	0.00	0.00	-[VisibleOne setFrameX:] [39]
0.0	0.05	0.00	1	0.00	0.00	-[VisibleOne setFrameY:] [40]
0.0	0.05	0.00	1	0.00	0.00	-[VisibleOne setScaleX:] [41]
0.0	0.05	0.00	1	0.00	0.00	-[VisibleOne setScaleY:] [42]
0.0	0.05	0.00	1	0.00	0.00	-[VisibleOne setSelfScale:] [43]
0.0	0.05	0.00	1	0.00	0.00	-[VisibleOne setTranslateX:] [44]
0.0	0.05	0.00	1	0.00	0.00	-[VisibleOne setTranslateY:] [45]
0.0	0.05	0.00	1	0.00	0.00	-[VisibleOne setTransparencyIndicator:] [46]
0.0	0.05	0.00	1	0.00	0.00	-[VisibleOne setVitalMatrix:] [47]
0.0	0.05	0.00	1	0.00	0.00	_main [54]

In our example, the plateau is being reached at the third line, which matches with the call graph profile, where most of the time is spent on the descendents of **drawSelf::** (**rectfill** and **tripath**), see index [4], [5], [6] and [7].

The **calls** column lists again the number of times the method or function was invoked. The trouble areas would be when a method or function gets called frequently and also has a high average number of ms spent per call (ms/call).

Using the Process Monitor to Measure Memory Usage

For a basic introduction to the Process Monitor application, see Chapter 4 in the *NeXT Development Tools* manual of the

NeXT Developer's Library.

Step 1: You can start both VisibleView version 01 and version 02. However, these applications must have different names, so that the Process Monitor can differentiate them. For example, visibleOne for version 01, and VisibleView for version 02.

Step 2: Start up the ProcessMonitor application, which is located in **/NextDeveloper/Apps**.

Note: If you start the Process Monitor before the other applications, such as VisibleView and visibleOne, select Processes in the Process Monitor menu, then select Update. The Processes panel will be updated to reflect the new running applications.

Step 3: Select Processes in order to see all the applications that can be monitored. Select visibleOne.

Step 4: Bring up the Malloc Inspector from the inspector panel.

Step 5: The first number is the default page; if you have done a zone malloc, there will be a different number for each zone. Look at Memory Malloced at start up and efficiency level (efficiency = malloced bytes/ number of pages *8K), and then look at the difference when more panels are added.

Repeat the same steps for VisibleView. Since VisibleView has taken out the infoPanel nib, and there are no extra sounds in the nib files, the efficiency level should be higher.

Note: In order to monitor another application, you have to select Processes from the Process Monitor main menu and choose the new application to be monitored. The Malloc Inspector window will reflect the changes immediately.

Step 6: Bring up the Display Postscript Inspector from the inspector panel. The Display Postscript Inspector gives the Virtual Memory size of the running application, as well as the Backing Store size. From the Process Monitor main menu select Monitor, then Start Monitor. Notice the Display Postscript Monitor graph, which will shoot up as you bring in more panels (coordinates, frames, preferences, statistics). This could be a quick test for memory leaks, if you see that the amount of

memory keeps increasing even though you have closed the window. For example, in a text document application, when a new document is created the amount of memory should increase, and decrease when it gets closed. You can try this with the Edit application.

Note: If you are currently monitoring the visibleOne application, you have to close the current Display Postscript Monitor, reselect Edit in the Processes Panel, and select Start Monitor again in order to inspect the new application. The Display Postscript Monitor displays on its left the icon of the application currently under inspection. In the case of visibleOne, since all the panels are contained in one single nib file and they don't get freed when closed, you'll notice that the graph will rapidly reach a plateau.

Qualitatively Measuring Drawing Performance

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

NXShowAllWindows

The command-line argument **NXShowAllWindows** is extremely useful for identifying situations in which you're doing unnecessary drawing. **NXShowAllWindows** makes all your windows visible (including those holding Bitmap objects), 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. **NXShowAllWindows** is often a good first step in diagnosing performance problems. To use **NXShowAllWindows**, launch your application from a Shell window, giving **NXShowAllWindows** as an argument:

```
VisibleView -NXShowAllWindows YES
```

Use this command to start up both `visibleOne` and `VisibleView`. Note that in Version 1, the NeXT logo has been created as an off-screen window even before the composite selection is checked in Preferences. In Version 2, the NeXT logo won't come up until it's needed for compositing.

Note: You can also use a tool (developed by the Lotus Improv team) called Winfo. This tool is useful for finding windows that allocate alpha or color planes unnecessarily, thereby wasting large amounts of memory. The Winfo tool is included with this exercise folder.

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 **NXShowAllWindows** 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 shell window, giving **NXShowPS** as an argument:

```
appName -NXShowPS YES
```

For finer-grained 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 some method. In our example, you might want to inspect the method **drawSomePS** more closely with **showps**, since it looks like a good candidate for optimization from the **gprof** profiling.

Quantitatively Measuring Drawing Performance

We'll use a simple interval timer as described in the Technical Notes on Performance Tuning to measure not only the time

spent within the processD(which is shown with **gprof**), but also the time spent in other processes, most notably the Window Server. This timer will measure the CPU time spent within an interval delineated by a pair of messages to the Timing object.

Step 1: In the file **VisibleOne.m**, turn the debugging flag on by changing **#define TIMING_DEBUG 1**. If you want to do the same for your application, you should include the **Timing.h** file in the file where you want to call the timing routines. Also, you'll have to add the **Timing.[hm]** files to your project directory with Interface Builder.

Step 2: Run **make** again.

Step 3: Start your VisibleOne application from a shell window. The drawing measurements will be shown in the shell window. If you start it up from the Workspace, the measurements will be shown in the Console window instead. In our example, we want to measure the **drawGrid** method. Here is some sample output:

```
localhost> visibleOne.debug
Timer 4 : 1 trials App: 0.000000  Server: 0.048000
                    Percent Server: 1.000000 Total: 0.048000
Timer 4 : 1 trials App: 0.000000  Server: 0.032000
                    Percent Server: 1.000000 Total: 0.032000
Timer 4 : 1 trials App: 0.000000  Server: 0.048000
                    Percent Server: 1.000000 Total: 0.048000
Timer 4 : 1 trials App: 0.000000  Server: 0.048000
                    Percent Server: 1.000000 Total: 0.048000
Timer 4 : 1 trials App: 0.000000  Server: 0.032000
                    Percent Server: 1.000000 Total: 0.032000
Timer 4 : 1 trials App: 0.000000  Server: 0.032000
                    Percent Server: 1.000000 Total: 0.032000
Timer 4 : 1 trials App: 0.000000  Server: 0.048000
                    Percent Server: 1.000000 Total: 0.048000
Timer 4 : 1 trials App: 0.000000  Server: 0.032000
                    Percent Server: 1.000000 Total: 0.032000
Timer 4 : 1 trials App: 0.000000  Server: 0.032000
                    Percent Server: 1.000000 Total: 0.032000
```

```

Timer 4 : 1 trials App: 0.000000  Server: 0.048000
                Percent Server: 1.000000 Total: 0.048000
Timer 4 : 1 trials App: 0.000000  Server: 0.032000
                Percent Server: 1.000000 Total: 0.032000
Timer 4 : 1 trials App: 0.000000  Server: 0.048000
                Percent Server: 1.000000 Total: 0.048000
Timer 4 : 1 trials App: 0.015351  Server: 0.032000
                Percent Server: 0.675804 Total: 0.047351
Timer 4 : 1 trials App: 0.000000  Server: 0.048000
                Percent Server: 1.000000 Total: 0.048000
Timer 4 : 1 trials App: 0.000000  Server: 0.032000
                Percent Server: 1.000000 Total: 0.032000
Timer 4 : 1 trials App: 0.000000  Server: 0.048000
                Percent Server: 1.000000 Total: 0.048000
Timer 4 : 1 trials App: 0.000000  Server: 0.048000
                Percent Server: 1.000000 Total: 0.048000
Timer 4 : 1 trials App: 0.000000  Server: 0.032000
                Percent Server: 1.000000 Total: 0.032000
Timer 4 : 1 trials App: 0.000000  Server: 0.048000
                Percent Server: 1.000000 Total: 0.048000
Timer 4 : 1 trials App: 0.000000  Server: 0.032000
                Percent Server: 1.000000 Total: 0.032000
Timer 4 : 1 trials App: 0.015344  Server: 0.032000
                Percent Server: 0.675904 Total: 0.047344

```

The timer is given a tag number (4). The second number (1) indicates the number of times this timer function is called. In our case, there is no loop (i.e., the function is called once only). In general, you'll want to average the timing measurements over a certain period of time. Since **drawGrid** is being called frequently (122 times from the **gprof** listing included in this document—see index [15]), we can just look at the number average from the standard output. The number following **App:** is the CPU time spent in the application. Here, it's mostly zero time, and it's being confirmed in the **gprof** listing. However, the time spent in the **Server**, which follows next, varies between 0.032 and 0.048. This is the time spent in the Window Server on behalf of your application. The **Percent Server** is the percentage of time consumed by the server over the total CPU time. Finally, the **Total** is the sum of the Application time and the Server time. Note that the time unit is in seconds.

This exercise shows that you'll need to improve your drawing method, so that it can become more efficient.

Finding Memory Leaks: MallocDebug vs. ProcessMonitor

While ProcessMonitor is an ideal tool for looking at Memory Usage, MallocDebug is very useful for detecting memory leaks. However, we don't recommend using both tools together, since MallocDebug has some limitations. MallocDebug can only allocate a total of about 40 megabytes throughout the life of the application, even if allocated memory is immediately freed. Also, the VM size of your program could become significantly higher after linking with **libMallocDebug.a**. For example, run **ps -ux** before the link with **libMallocDebug.a** and after the link, and note the VSIZE values in each case.

Step 1: Start up Interface Builder.

Step 2: Open your project directory **IB.proj**, and select Inspector under the Tools Menu.

Step 3: Add the file **/usr/lib/libMallocDebug.a** to Other libs.

Step 4: Run make again to relink your application with MallocDebug.

Step 5: Start up your application from the Workspace. Start MallocDebug from **/NextDeveloper/Apps**.

Step 6: Select Application from the MallocDebug main menu, and Open.... Select visibleOne.

Step 7: Select Leaks.

Sample Output from MallocDebug, after selecting Leaks:

```

Zone:      Address:      Size:      Function:
default 0x0010f290      9      NXCopyStringBufferFromZone, _NXDecodeString, ReadValues, NXReadTypes, -[Bitmap
read:], InternalReadObject, ReadValues, NXReadArray
default 0x0019643c      13      NXCopyStringBufferFromZone, _NXDecodeString, ReadValues, ReadValues, NXReadArray, -
[Storage read:], InternalReadObject, ReadValues
default 0x00199ee8      80      class_createInstanceFromZone, InternalReadObject, ReadValues, NXReadTypes, -
[WindowTemplate read:], InternalReadObject, ReadValues, ReadValues
default 0x001bc8dc      100      -[VisibleOne hitTest:], -[View hitTest:], -[View hitTest:], -[FrameView hitTest:], -
[Window sendEvent:], -[Application sendEvent:], -[Application run], main
default 0x001bc97c      100      -[VisibleOne hitTest:], -[View hitTest:], -[View hitTest:], -[FrameView hitTest:], -
[Window sendEvent:], -[Application sendEvent:], -[Application run], main

```

The Zone field lists the memory zone name. In the case of VisibleView, the default zone is being used. There are some known memory leaks that were mentioned in the 2.0 Release Notes. See also:

</NextLibrary/Documentation/NextDev/ReleaseNotes/DeveloperApps.rtf>

In the above listing, there are 2 known leaks:

- A 9-byte leak caused by **[Bitmap read:]**ⒹUnarchiving a Bitmap object leaks the name of the Bitmap. Note that in the VisibleView-01, the Bitmap name is "nextLogo", which is 8 bytes plus one for the string terminator.
- A 13-byte leak caused by the unarchiving of the NIB file.

The function list on the sample output is a function backtrace. The leftmost function is the one that calls **malloc()**, and the function calls are stacked from left to right.

Remaining leaks:

- An 80-byte leak caused apparently by unarchiving an object in a nib file.

- A 100-byte leak each time the method **hitTest:** is called.

Other Steps To Improve VisibleView-01

Create separate nib files:

- Inspect the files **VisibleOne.[hm]**. Note that all the **set*** methods for setting outlets are still present.
- Inspect the **visibleOne.nib** file.
- Note that there is only one single nib file.
- Move the info panel to a separate nib file.

Run gprof again:

- Repeat the profiling steps as described in the section ^aUsing gprof with VisibleView-01.^o
- Compare your profiling outputs with the one generated in the section ^aUsing gprof with VisibleView-01.^o What do you notice?
- When working with the newer version, do you perceive any performance improvement?

More Improvements To VisibleView-01 :

- Use the following check list to see what other improvements you can add to VisibleView-01. Afterwards, run a new set of measurements using **gprof**.

Drawing Performance:

- Use best drawing methods (user paths, **xyshow**, **NXRectFillListWithGrays()**,^{1/4}).
- Send all windows **useOptimizedDrawing:** unless this causes incorrect display.
- No redundant PostScript from App's Views (check using **showps** around **drawSelf::** in **gdb**).
- Use **setwindowtype** operator where appropriate to give better drawing feedback.

App Memory Usage:

- Optimize all data structures for size, fragmentation and locality. Zone malloc and private allocators considered. See also **/NextLibrary/Documentation/NextDev/Notes/MakingAppsFly/LinkOptimization.rtf** and **/NextLibrary/Documentation/NextDev/ReleaseNotes/Zones.rtf**
- Optimize text and data working set for common operations. App is scatter-loaded.
- No leaks show up when using ^aLeaks^o feature of MallocDebug.
- No leaks of kernel objects (ports, open files, vm_regions).
- Use copy-on-write memory where possible (memory mapped files, out-of-line data in messages, **vm_copy()** for large regions).
- Use **setStringValueNoCopy:** instead of **setStringValue:** where appropriate.

Window Server Memory Usage:

- All windows should be deferred.
- One-shot windows should be used where possible (remember the 040 drawing speed!).
- No windows with alpha channel on screen (check with Winfo app).
- No unnecessary depth promotion. Use **setpattern** to draw 1/6, 1/2 and 5/6 grays.

Running App with -NXShowAllWindows YES or -NXAllWindowsRetained YES:

- No redundant or unused off-screen bitmaps.
- No unused windows. (Create them lazily by moving to separate nib section.)
- No redundant drawing. (Check out Window's **disableDisplay**, **displayIfNeeded**.)

Interface Builder:

- One window per IB module. Info panel removed from main interface module.
- No nib sections loaded with names, including the one loaded from **main()** (must use **loadNibSection:owner:withNames:NO**).
- All images added to project. (images load faster from Mach-O, allows sharing of images between nib sections).
- All set* methods for setting outlets removed.
- No extra sounds, images or scratch objects in nib files.
- All windows and panels that are created using IB must have the option "Free when closed" set. Use the IB Inspector panel.
Otherwise, loading these nib files in dynamically will leave memory leaks, since these windows or panels won't be freed when you close them.

Images:

- Use NXImage compression where applicable.
- Don't composite with NX_COPY (use NX_SOVER, which doesn't allocate alpha channel).
- Replace all instances of obsolete Bitmap class with NXImage class.

Hardware configuration:

It is very important to test (and even develop) applications on machines that reflect the most common user configuration. Therefore, remember to run your app with the memory set to 8 megabytes. To do this, type the following command to the ROM boot monitor:

```
>bsd sdmach mem=8192
```

The **mem** argument specifies the number of 8K pages. The boot messages will tell you how much memory is used. You can also use the **hostinfo** command in a shell window to display the system information.

Remember to schedule time in your development cycle for performance tuning your app!

References

- The Technical Support Note on Performance Tuning, written by Bruce Blumberg
See **/NextLibrary/Documentation/NextDev/Notes/MakingAppsFly**
- The 2.0 Release Notes
See **/NextLibrary/Documentation/NextDev/ReleaseNotes/Performance.rtf**