Reducing Memory Usage

Memory usage is the most critical performance issue on NEXTSTEP platforms. An operating system with virtual memory can be a double-edged sword. While virtual memory is a key component to building a useful multitasking user environment, when there is not enough physical memory to satisfy all the applications, servers, and deamons, the resulting paging can be very frustrating to the user.

When using NEXTSTEP, users typically use a number of applications in concert. Features like drag-and-drop, Object Links and Services further serve to tie applications together. The result is that the overall performance of the system is as important as the individual performance of a particular application. To achieve good system performance, it's essential that applications be "good citizens" of virtual memory. It isn't acceptable for an application to create panels that the user will never see at startup time, or build bitmap caches of images that are rarely drawn, or that could be drawn about as quickly with userpaths or other PostScript. As CPU speed increases, there are fewer and fewer areas where trading off using more memory for less CPU is appropriate. When you are faced with such a tradeoff, consider carefully whether the excess memory used really brings a substantial benefit to the user. **When in doubt, lean towards saving memory.**

The following are some tips concerning memory usage.

Bitmaps and Off-Screen Windows

A common performance optimization is to create off-screen window using NXImage objects, draw into them once, and then composite on-screen. As mentioned earlier, you should use this technique judiciously, since it can consume a lot of memory in the Window Server. Consider using user paths as an alternative when possible. Perhaps most important, create these NXImages only as you need them, and remember to free them when you're done with them.

To look for unnecessary off-screen windows, run your application with the option "-NXShowAllWindows YES" to force all windows on-screen.

Garbage Collection

Remember you're the garbage collector for your application. Objects will not get freed unless you explicitly free them. If you're done with an object, free it! Remember also that if an object's instance variables point off to other objects or **malloc**'ed memory, you may need to explicitly free them as well (that is, freeing an object doesn't necessarily free its instance variables). In general, the Application Kit classes free their instance variables as appropriate (for example, freeing a Window frees all of the Views within that Window's view hierarchy). However, when you create a subclass of an Application Kit class and add instance variables, you'll need to override the **free** method for that class and do the following:

```
-free
{
/* free new instance variables */
/* . . . */
/* now do what your super class would normally do */
return [super free];
}
```

If your application seems to grow over time, check to be sure that objects are freed when no longer needed. Alternatively, see if you can reuse existing objects.

The **MallocDebug** application is extremely useful for finding memory leaks.

NXUniqueString()

The Application Kit provides a number of functions (**NXUniqueString()**, **NXUniqueStringNoCopy()**, and some others) that create unique strings which are allocated once and then can be shared. The unique strings created are of type NXAtom, which means that they can be compared using **==** rather than **strcmp**. Creating the strings is quick because the strings are kept in a hash table. Once created, the strings are read-only and should not be deallocated.

Use of unique strings can pay off handsomely in cases where you have read-only strings and there's repetitive use of the same string. A perfect example of their use is in the mail summary window in Mail. The strings

contained in the date and name fields should be instances of unique strings, since they're read-only and the same string is used many times.

loadNibSection:owner:withNames:

As mentioned earlier, if you don't use the names generated by Interface Builder, you can reduce the amount of memory associated with reading in an interface section by using the **loadNibSection:owner:withNames:** method, passing NO as the argument for **withNames:**. See the earlier section ªImproving Launch Times of the Application and Panelsº for more information.

Lazy Creation of Objects

You can reduce the working size of your application by not instantiating objects until you need them. This is particularly true if the object is used infrequently or not at all unless the user takes a particular action. By not instantiating the object until you need it, your application doesn't grow until it's absolutely necessary to do so.

Along these lines, be sure to place each window or panel in its own Interface Builder document. It is extremely wasteful to have many panels and windows all in the same nib file, because all these objects are loaded in to memory and instantiated when any of them is used.

In addition almost all windows in an application should be made deferred and "one-shot" in Interface Builder's Window inspector. This means that the actual backing store for the window will be created only when it is needed, and freed when the window is taken from the screen. Only windows that are very expensive to draw should not be one-shot. This is especially a win for panel, which are often dismissed from the screen and remain invisible for a long time. Since color application icons cause most panels to be promoted to 16-bit depth, it is much better to allow the one-shot mechanism to deallocate the backing store when the panel is closed, providing an immediate pool of free memory to the system, than to force the system to write these pages into the swapfile for later retrieval.

Zone Allocation

Zone allocation is a powerful feature that allows you to group your application's heap data onto different sets of pages, or zones. This reduces the working set for various operations within your application. See **ZoneAllocation.rtf** for more information about using zones.

A Note on Symbols

It's worth noting that compiling with or without symbols has no effect on the in-memory size of your application. When the compiler generates symbols, they're put in a section at the end of the executable file. However, they're only brought into memory by the debugger. Thus, while stripping your program will significantly reduce the size of your application on the disk, it'll have no effect on memory usage or performance.

On the subject of compiler flags, using the **-O** (optimization) option can yield potential gains in performance, as well as somewhat smaller code, so you should always compile your production version with optimization turned on. In fact, as long as the code is also compiled with symbols, you can debug the optimized code, although be warned that the optimization sometimes fools GDB.