Cocoa Performance Guidelines

Performance

2009-08-11

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Listings

Introduction to Cocoa Performance Guidelines

This document provides practical advice and tips on how to improve the performance of your Cocoa applications. If you create any type of Cocoa program, including command-line tools, you will find information to help you improve its performance.

Organization of This Document

This document contains the following articles:

- "General Recommendations" (page 9) provides some general tips for achieving good performance with Cocoa.
- "Unblocking Your User Interface" (page 13) explains ways to perform lengthy tasks in a way that keeps your user interface responsive to user commands.
- "Using Views Effectively" (page 15) explains how to get the best performance out of your custom NSView subclasses.
- "Cocoa Live Window Resizing" (page 19) provides tips for speeding up your drawing code during a live window resize.
- "Cocoa Bindings Tips" (page 23) provides basic tips for improving performance when using Cocoa bindings.
- "Improving NSBezierPath Rendering Times" (page 25) describes ways to speed up drawing operations involving the NSBezierPath object.
- "String Management" (page 27) provides tips for using strings effectively in your application.
- "Notifications" (page 29) provides tips for using notifications efficiently and offers alternatives for communicating among the objects in your application.

General Recommendations

The following sections provide some basic guidance for achieving good performance with the Cocoa framework.

Always Measure

Never assume you know where performance problems lie. The only way to know for sure is to measure the actual performance of your Cocoa application and use data from various tools to identify the real problems.

Use a Scalable Architecture

Choose a scalable application architecture. Your choice of algorithms can make a big difference in performance. Choose algorithms that scale well to the intended data set. Test your program on large data sets to make sure your performance is appropriate.

Don't Fight the Framework

The Cocoa framework provides many of the features you need to implement advanced applications. In addition, improvements to Cocoa are being made regularly. By taking advantage of the facilities Cocoa provides for extending or modifying framework behavior, such as delegation and subclassing, you make your applications more likely to benefit from future improvements.

Be sure to use data structures and data types that are compatible with the methods you need to call. Constructing or converting a data structure at runtime solely for the purpose of calling a system function adds an unnecessary performance penalty to your application. If you can incorporate a compatible version of the data structure into your model, you can eliminate that penalty.

Limit Your Use of Autoreleased Objects

Many object factory methods return autoreleased objects by convention. Although this technique ensures that objects are released properly, it is often unnecessary. If you create temporary objects within a method, you know that those objects can be released when the method finishes. Rather than have them linger until the next time the autorelease pool is cleared, it is better to release them immediately.

Releasing objects right away can help you avoid potential memory spikes. If you allocate thousands of temporary objects before the next sweep of the autorelease pool, those objects could be paged to disk as memory space runs low. During the sweep, those same objects would then be read back into memory so that they could be deleted, which is a waste of time.

Be Aware of Indeterminate Time Operations

Because of latency and transfer speeds, file-system and network requests can potentially take much longer to complete than other types of operations. If you access file or network data synchronously, your application might appear frozen while it waits for the data to return. Therefore, it is usually better to perform these types of operations asynchronously or on a secondary thread. You might also want to consider firing off a timer if the operation does not complete within a certain amount of time. You can use this timer to notify the user of the delay or provide an option to cancel the operation.

Consider Data Caching Carefully

You should always approach data caching carefully and measure the performance of your implementation. Data caching can significantly improve the speed of your code or hopelessly degrade it. Identifying the right data to cache is a difficult task and is dependent on your design. If you cache the wrong data, you could waste memory storing unused data. Even if you cache the right data, that data could still be paged to disk during low-memory conditions.

Be sure to test your caching implementation under a variety of circumstances, especially in low-memory situations. If you find your data is being paged out to disk, consider reducing the size of your cache or eliminating it altogether. At that point, caches could cause three disk operations instead of just one.

Defer Operations Whenever Possible

Reducing the amount of work you do at critical moments is one of the best ways to give users the perception of speed in your application. Deferring operations whose results are not needed right away can help improve performance at those critical moments, as well as at other times too.

Drawing is a particular area where deferring operations can yield significantly better performance. Rather than force your window to update each time you make a change, simply invalidate areas that need to be redrawn. There is a good chance that more updates might have come before the window contents actually need to be redrawn. By invalidating, you can coalesce those drawing operations and perform them all at once.

Another place where applications typically take a big performance hit is in loading plug-ins and other external code modules that are not needed right away. If you load a code module or plug-in that is never used, it is a waste of both CPU time and memory. If you wait until the code is actually called, you may incur a small penalty by loading the code at that time but you guarantee that the code is actually used.

Defer the Display of Dynamically Updated Data

If you are loading data dynamically and displaying it in a view, avoid telling your view to update itself too often. It is better to delay the display of that data a short amount of time rather than force your view to redraw its contents immediately, especially if more data could arrive at any time.

For example, suppose you have a script that generates data to be displayed in a table view. If the script provides an initial flood of data and then sends periodic updates, calling the `reloadData` method each time you received a piece of the data could lead to numerous useless updates. Instead, you should defer the call to the `reloadData` method until a certain amount of time has passed. If more data comes in during that time, you can then cancel the current request and initiate a new one. Listing 1 shows you some sample code for how to do this.

Listing 1 Cancelling and restarting a reloadData operation

```
// Cancel the old request
[NSObject cancelPreviousPerformRequestsWithTarget:myTableView
selector:@selector(reloadData) object:nil];

// Initiate a new request after a brief delay.
[myTableView performSelector:@selector(reloadData) withObject:nil afterDelay:0.3];
```

Use Binary Serializations Whenever Possible

If you use the NSPropertyListSerialization class to read and write property-list information, you should format your property list files as binary XML whenever possible. Binary data is significantly more compact than its text-based counterpart. As a result, reading and writing binary-based data is significantly faster (often 5 to 20 times faster) than reading and writing the corresponding text-based data.

Support for the binary XML format was introduced in Mac OS X v10.2. Although users cannot view the contents of a binary property-list file directly using a text editor, they can use the Property List Editor application to read them.

Note: Some classes, such as NSUnarchiver and NSArchiver, already read and write data using binary file formats.

For more information on property lists and binary file formats, see *Property List Programming Guide*.

Unblocking Your User Interface

In a Cocoa application, the main thread runs the user interface, that is, all drawing and all events are handled on the main thread. If your application performs any lengthy synchronous operations on that thread, your user interface can become unresponsive and trigger the spinning cursor. To avoid this, you should shorten the amount of time consumed by those operations, defer their execution, or move them to secondary threads.

Deferring the Execution of Operations

If you have operations that can be deferred or broken into chunks and performed incrementally, doing so can help performance. Operations that are iterative or modular in nature, such as scratch calculations, can usually be performed in small chunks. By performing a few of these calculations at idle time using asynchronous notifications, you can avoid blocking your main thread. To register for idle time notifications, you would simply post a notification object to the default queue and ask for it to be dispatched at idle time, as shown in the following example:

```
NSNotification* myNotification = [NSNotification
notificationWithName:@"MyIdleNotification" object:myIdleHandlerObject];

[[NSNotificationQueue defaultQueue] enqueueNotification:myNotification
postingStyle:NSPostWhenIdle];
```

Timers offer another way to do small amounts of work at more regular intervals. The advantage of timers is that they fire at a known time. The disadvantage is that they fire regardless of how busy your application is, which could still cause a perceptible delay. For information on how to set up a timer, see *Timer Programming Topics*.

Using Run Loop Observers

Another way to defer an operation is to hook into the run loops of your application's threads. By adding a CFRunLoopObserver to a run loop, you can receive notifications at certain points in the execution of that run loop. For example, you can be notified before timers are processed, before input sources are processed, or before or after the run loop sleeps.

To create a run loop observer, you must first define a callback function of type `CFRunLoopObserverCallBack`. The following code shows a sample implementation of this method. The `info` parameter contains data you want passed to your handler when it is called, which in this case is an instance of the fictional MyObject.

```
void MyRunLoopObserver(CFRunLoopObserverRef observer,
                           CFRunLoopActivity activity,
                           void* info)
{
    MyObject* theObject = (MyObject*)info;
```

```
    // Perform your tasks here.
}
```

After you define your callback function, you need to create the run loop observer and install it on the current run loop. The following code shows you how to do this from a Cocoa method. The code requests that the callback function be run before any timers are processed or before the run loop goes to sleep. Note that the last parameter to `CFRunLoopObserverCreate` is the `self` variable, which is assumed to be the instance of MyObject that the callback handler expects.

```
- (void) installRunLoopObserver
{
    CFRunLoopObserverRef myObserver = NULL;
    int myActivities = kCFRunLoopBeforeTimers | kCFRunLoopBeforeWaiting;

    // Create the observer reference.
    myObserver = CFRunLoopObserverCreate(NULL,
                            myActivities,
                            YES,        /* repeat */
                            0,
                            &MyRunLoopObserver,
                            (void*)self);

    if (myObserver)
    {
        // Now add it to the current run loop
        CFRunLoopAddObserver(CFRunLoopGetCurrent(),
                            myObserver,
                            kCFRunLoopCommonModes);
    }
}
```

Using Threads

Secondary threads are an excellent way to perform lengthy operations without blocking your user interface. If your application performs several sequential operations, you could also create a worker thread to handle new requests as they arise. The advantage of threads is that they can offer significant performance gains, especially on multiprocessor systems. The disadvantage is that you must plan carefully to make sure two or more threads do not try to manipulate the same data at the same time.

For more information about creating and managing threads, see *Threading Programming Guide*.

Using Views Effectively

Cocoa views provide extensive support for drawing and managing visual content. However, as with any object, improper use can lead to performance degradation. The following sections offer guidelines on how to get the most out of your custom views.

Be Opaque

If you implement a custom subclass of NSView, you can accelerate the drawing performance by declaring your view object as opaque. An opaque view is one that fills its entire bounding rectangle with content. The Cocoa drawing system does not send update messages to a superview for areas covered by one or more opaque subviews.

The `isOpaque` method of NSView returns `NO` by default. To declare your custom view object as opaque, override this method and return `YES`. If you create an opaque view, remember that your view object is responsible for filling its bounding rectangle with content.

Draw Minimally

Poor drawing performance is often caused by an application doing too much work. The `drawRect:` method of NSView receives a rectangle that identifies the area that needs to be redrawn. In your implementation of this method, you should always check the rectangle and avoid drawing content that lies outside its boundaries.

Drawing Content

Although checking your content against the bounding rectangle passed into `drawRect:` is a good start, Cocoa does provide additional methods for determining what needs to be redrawn. The availability of these methods depends on the running version of Mac OS X though, so you might need to check to make sure a method is available before attempting to use it.

In Mac OS X version 10.3 and later, Cocoa applications have two ways of obtaining a more refined version of the drawing rectangle. The rectangle passed into an NSView `drawRect:` method is formed by creating a union of all the dirty rectangles. However, if updated areas are small and far apart, the union area can often be much larger and contain a lot of unchanged content. Instead of using this rectangle, you can instead call the view's `getRectsBeingDrawn:count:` method to get an array of the individual update rectangles and use the `NSIntersectsRect` function to compare them to the bounding rectangle of your object. An even simpler way to do this is to call the `needsToDrawRect:` method, which performs both of these steps for you.

The following example shows a simple `drawRect:` method that draws the objects in a custom view. In this example, the MyDrawableThing class does not descend from NSView but instead defines an object with a bounding rectangle and some content that the view knows how to draw.

```
- (void) drawRect:(NSRect)aRect
{
    NSEnumerator* myEnumerator = [myArray objectEnumerator];
    MyDrawableThing* currentThing;

    while (currentThing = [myEnumerator nextObject])
    {
        // Draw the thing if it is in one of the update rectangles.
        if ([self needsToDrawRect:[currentThing bounds]])
        {
            [self drawThing:currentThing];
        }
    }
}
```

You can use the Quartz Debug tool to see where your application is drawing and to find areas where it is drawing content redundantly. For more information, see “Measuring Drawing Performance” in *Drawing Performance Guidelines*.

Invalidating Portions of Your View

Cocoa provides two techniques for redrawing the content of your views. The first technique is to draw the content immediately using the `display`, `displayRect:`, or related methods. The second is to draw the content at a later time by marking portions of your view as dirty and in need of an update. This second technique offers significantly better performance and is appropriate for most situations.

NSView defines the methods `setNeedsDisplay:` and `setNeedsDisplayInRect:` for marking portions of your view as dirty. Cocoa collects the dirty rectangles and saves them until the top of your run loop is reached, at which point your view is told to redraw itself. The rectangle passed into your `drawRect:` routine is a union of the dirty rectangles, but applications running on Mac OS X version 10.3 and later can get a list of the individual rectangles, as described in "Drawing Content" (page 15).

In general, you should avoid calling the `display` family of methods to redraw your views. If you must call them, do so infrequently. Because they cause an immediate call to your `drawRect:` routine, they can cause performance to slow down significantly by preempting other pending operations. They also preclude the ability to coalesce other changes and then redraw those changes all at once.

Avoid the Overuse of Views

NSView offers tremendous flexibility in managing the content of your windows and provides the basic canvas for drawing your application’s content. However, when you consider the design of your windows, think carefully about how you use views. While views are a convenient way to organize content inside a window, if you create a complex, deeply nested hierarchy of views, you might experience performance problems.

Although Cocoa windows can manage a relatively large number of views (around one hundred) without suffering performance noticeable problems, this number includes both your custom views and the standard system controls and subviews you use. If your window has hundreds of custom visual elements, you probably

do not want to implement them all as subclasses of NSView. Instead, you should consider writing your own custom classes that can be managed by a higher-level NSView subclass. The drawing code of your NSView subclass can then be optimized to handle your custom objects.

A good example of when to use custom objects is a photo browser that displays thumbnail images of hundreds or even thousands of photos. Wrapping each photo in an NSView would both be prohibitively expensive and inefficient. Instead, you would be better off by creating a lightweight class to manage one or more photos and a custom view to manage that lightweight class.

Cocoa Live Window Resizing

Live window resizing is an area where poorly optimized drawing code becomes particularly apparent. When the user resizes your window, the movement of the window should be smooth. If your code tries to do too much work during this time, the window movement may seem choppy and unresponsive to the user.

The following sections introduce you to several options for improving your live resizing code. Depending on which versions of Mac OS X you are targeting, you might use one or more of these options in your implementation.

Draw Minimally

When a live resize operation is in progress, speed is imperative. The simplest way to improve speed is to do less work. Because quality is generally less important during a live resize operation, you can take some shortcuts to speed up drawing. For example, if your drawing code normally performs high-precision calculations to determine the location of items, you could replace those calculations with quick approximations during a live resize operation.

NSView provides the `inLiveResize` method to let you know when a live resize operation is taking place. You can use this method inside your `drawRect:` routine to do conditional drawing, as shown in the following example:

```
- (void) drawRect:(NSRect)rect
{
    if ([self inLiveResize])
    {
        // Draw a quick approximation
    }
    else
    {
        // Draw with full detail
    }
}
```

Another way to minimize work is to redraw only those areas of your view that were exposed during the resize operation. If you are targeting your application for Mac OS X version 10.3, you can use the `getRectsBeingDrawn:count:` method to retrieve the rectangles that were exposed. If you are targeting Mac OS X version 10.4 or later, the `getRectsExposedDuringLiveResize:count:` method is provided to return only the rectangles that were exposed by resizing.

Cocoa Live Resize Notifications

Starting with Mac OS X v10.1, you can use the `viewWillStartLiveResize` and `viewDidEndLiveResize` methods of NSView to help optimize your live resize code. Cocoa calls these methods immediately before and immediately after a live resize operation takes place. You can use the `viewWillStartLiveResize` method to cache data or do any other initialization that can help speed up your live resize code. You use the `viewDidEndLiveResize` method to clean up your caches and return your view to its normal state.

Cocoa calls `viewWillStartLiveResize` and `viewDidEndLiveResize` for every view in your window's hierarchy. This message is sent only once to each view. Views added during the middle of a live resize operation do not receive the message. Similarly, if you remove views before the resizing operation ends, those views do not receive the `viewDidEndLiveResize` message.

If you use these methods to create a low-resolution approximation of your content, you might want to invalidate the content of your view in your `viewDidEndLiveResize` method. Invalidating the view causes it be redrawn at full resolution outside of the live resize loop.

If you override either `viewWillStartLiveResize` or `viewDidEndLiveResize`, make sure to send the message to `super` to allow subviews to prepare for the resize operation as well. If you need to add views before the resize operation begins, make sure to do so before calling `super` if you want that view to receive the `viewWillStartLiveResize` message.

Preserve Window Content

In Mac OS X v10.4 and later, Cocoa offers you a way to be even smarter about updating your content during a live resize operation. Both NSWindow and NSView include support for preserving content during the operation. This technique lets you decide what content is really invalid and needs to be redrawn.

To support the preservation of content, you must do the following:

1. Override the `preservesContentDuringLiveResize` method in your custom view. Your implementation should return `YES` to indicate that the view supports content preservation.

2. Override your view's `setFrameSize:` method. Your implementation should invalidate any portions of your view that need to be redrawn. Typically, this includes only the rectangular areas that were exposed when the view size increased.

To find the areas of your view that were exposed during resizing, NSView provides two methods. The `rectPreservedDuringLiveResize` method returns the rectangular area of your view that did not change. The `getRectsExposedDuringLiveResize:count:` method returns the list of rectangles representing any newly exposed areas. For most views, you need only pass the rectangles returned by this second method to `setNeedsDisplayInRect:`. The first method is provided in case you still need to invalidate the rest of your view.

The following example provides a default implementation you can use for your `setFrameSize:` method. In the example below, the implementation checks to see if the view is being resized. If it is, and if any rectangles were exposed by the resizing operation, it gets the newly exposed rectangles and invalidates them. If the view size shrunk, this method does nothing.

```
- (void) setFrameSize:(NSSize)newSize
```

```
{
    [super setFrameSize:newSize];

    // A change in size has required the view to be invalidated.
    if ([self inLiveResize])
    {
        NSRect rects[4];
        int count;
        [self getRectsExposedDuringLiveResize:rects count:&count];
        while (count-- > 0)
        {
            [self setNeedsDisplayInRect:rects[count]];
        }
    }
    else
    {
        [self setNeedsDisplay:YES];
    }
}
```

Cocoa Bindings Tips

If your application uses Cocoa bindings, you should be aware of the differences between different types of value bindings. If you are using bindings to load images or large amounts of text, using the `data` or `value` bindings can offer significant performance advantages over the `valuePath` or `valueURL` bindings.

When you use the `data` or `value` bindings to load data, the bindings system retrieves the needed data directly from the attached model object, which is usually in memory. Conversely, when you use the `valuePath` and `valueURL` bindings, the system may have to retrieve the data from a hard drive or other slow device where the information resides. Because the bindings system does not know anything about your data model, it cannot effectively cache your data internally; it must retrieve it each time.

During scrolling or window resizing, the system may need to load bound data many times to match the currently visible contents of the window. Retrieving data repeatedly from the hard drive during these operations can slow down your application considerably. Therefore, if your view may be bound to large amounts of data, such as large text blocks or many images, it is better to use the `data` or `value` bindings.

Improving NSBezierPath Rendering Times

If you are using the NSBezierPath object to draw paths in a Cocoa application and you are not as interested in the absolute correctness of the rendered paths, there are ways to speed up the rendering times for complex paths. Prior to rendering an NSBezierPath object, the Core Graphics engine searches the path for any intersecting line segments. For each intersection it finds, the engine then rasterizes the line joint according to the chosen settings. If the NSBezierPath object contains a large number of intersecting line segments, the cost of these operations can become significant. If you do not need these intersections to be rendered precisely, you might try adding fewer line segments to your NSBezierPath objects prior to rendering.

If you are drawing many rectangles, you might want to avoid using NSBezierPath altogether. NSBezierPath contains the convenience method `strokeRect:` for drawing rectangles. However, the path created by this method consists of four intersecting line segments, with properly rendered corners. If you do not need the rectangle to be drawn so precisely, you might want to use the `NSFrameRect` family of functions instead. These functions draw the sides of a rectangle using four nonintersecting lines, which can be drawn much faster. The `NSFrameRect` family of functions consist of the following functions, declared in `NSGraphics.h`.

```
void NSFrameRect(NSRect aRect);
void NSFrameRectWithWidth(NSRect aRect, float frameWidth);
void NSFrameRectWithWidthUsingOperation(NSRect aRect, float frameWidth,
NSCompositingOperation op);
```

Keep in mind that using these techniques involves a correctness-versus-efficiency trade-off. You should not make this trade-off unless you are trying to solve a specific performance problem.

String Management

Strings are used extensively throughout Cocoa, and it's very likely your application uses many of them as well. If you run Shark and notice that your application spends a fair amount of time manipulating or displaying strings, you might want to look at your usage of NSString methods. There might be better ways to do what you need to do.

Optimize Your Text Manipulations

NSString and its assorted subclasses provide tremendous flexibility in how you can manipulate text, but that flexibility comes at a performance cost. If your application manipulates strings frequently or in very intensive ways, you might want to carefully consider the NSString methods you use. You might also want to consider writing your own string utilities to optimize the manipulations you do.

If you want to iterate over the characters of a string, one of the things you should not do is use the `characterAtIndex:` method to retrieve each character separately. This method is not designed for repeated access. Instead, consider fetching the characters all at once using the `getCharacters:range:` method and iterating over the bytes directly.

If you want to search a string for specific characters or substrings, do not iterate through the characters one by one. Instead, use higher level methods such as `rangeOfString:`, `rangeOfCharacterFromSet:`, or `substringWithRange:`, which are optimized for searching the NSString characters. You might also consider using the methods of NSScanner to parse the string contents for substrings. NSScanner also lets you parse a string for numerical values and return those values as scalar types.

Understand the Cost of Drawing Strings

NSString provides convenience methods for rendering strings. Methods such as `drawAtPoint:withAttributes:` and `drawInRect:withAttributes:` let you draw the string content to a specific location in the current view. However, drawing strings in this manner can still incur a significant amount of overhead.

NSString uses the Cocoa text system for rendering string content. When you ask NSString to render itself for the first time, it must set up several text system objects and then lay out and draw the glyphs in the string. The Application Kit does try to cache the text system objects to avoid some of these costs in the future.

Preserve Your Text Objects

If you have used the Cocoa text system at all, you should understand the amount of work it takes to render text. Text rendering requires numerous calculations to make sure characters have the right font, spacing, position, and so on. Collecting these attributes and then locating the glyphs to be drawn can involve the creation of numerous objects. If you throw these objects away after each use, you incur a significant performance penalty each time you draw your code. Much of this penalty can be easily removed by caching the objects you create.

Many of the objects in the Cocoa text system can be set up once and retained for future use. Preserving these objects can significantly improve text rendering performance during subsequent drawing operations.

Notifications

The Cocoa notification mechanism gives you a way of communicating changes to any number of other objects. An important feature of notifications is that the code sending the notification does not have to know anything about what is done with that notification. The interested objects decide how they want to respond, if at all.

The Cocoa frameworks use notifications to communicate important status information to your application. You can use these notifications as hooks to perform important tasks. For example, Cocoa sends the `NSApplicationDidFinishLaunchingNotification` notification to give you a chance to perform any secondary initialization of your application data structures after your application is up and running.

As with the Cocoa frameworks, you define your own notifications and use them to communicate important status information to other objects within your program. However, you should be careful not to overuse the notification mechanism. The following sections describe the performance issues related to notifications and guide you towards using them properly.

Be Selective and Specific

An important rule with notifications is to be picky about which notifications you handle. All notifications, whether posted synchronously or asynchronously, are eventually dispatched to the registered observers synchronously by the local NSNotificationCenter object. If there are a large number of observers or each observer does a lot of work while handling the notification, your program could experience a significant delay.

When you register to receive notifications, you should always try specify a valid notification name and object. Although you can specify `nil` for the notification name, doing so causes you to receive all notifications, which can cause a noticeable drop in performance. Specifying a valid name reduces the number of notifications that are dispatched to your handler. Similarly, if you specify a valid object with the notification, you further filter the number of notifications you receive and reduce the performance penalty.

Optimize Your Notification Handlers

When you define your notification handler methods, be as efficient as possible at handling the notification and returning control to the notification center. Remember that notifications occur synchronously. If you initiate a lengthy operation in the middle of your notification handler, you delay the receipt of the notification by other handlers and further delay the event that triggered the notification.

If you must perform additional work upon receiving a notification, consider deferring that work until later using a timer or by passing it to a worker thread. For more information, see "Unblocking Your User Interface" (page 13).

Avoid Repeatedly Adding or Removing Observers

Adding and removing observers from a notification center involves updating the dispatch tables maintained by that object. If your application creates a number of short-lived objects, you should avoid handling notifications in these objects if possible. If you really need to dispatch notifications to short-lived objects, consider creating an intermediary observer object to act as a liaison between the notification center and your objects.

An intermediary observer object can be tuned to your particular data structures much more effectively than the default NSNotificationCenter object. Your intermediary would maintain a list of your short-lived objects and should be able to add those objects to its internal data structures quickly. Upon receiving a notification from the notification center, you could then asynchronously pass that notification to your short-lived objects using the `makeObjectsPerformSelector:withObject:` method.

Consider Alternatives to Notifications

As you design your application, do not simply assume that you should send a notification to communicate with interested parties. You should also consider alternatives such as key-value observing, key-value binding, and delegation.

Key-value binding and key-value observing were introduced in Mac OS X version 10.3 and provide a way of loosely coupling data. With key-value observing, you can request to be notified when the properties of another object change. Unlike regular notifications, there is no performance penalty for unobserved changes. There is also no need for the observed object to post a notification because the key-value observing system can do it for you automatically, although you can still choose do it manually.

Another technique that has existed in Cocoa for many years is the notion of delegation. If you have an object that generates notifications, you can assign a delegate to that object. Rather than post notifications to the notification center, you instead call the methods of the delegate, which can perform whatever tasks it requires. In the case where only one object is interested in the notifications anyway, this technique is preferred.

For more information about key-value observing, see *Key-Value Observing Programming Guide*. For information about delegates, see *Cocoa Fundamentals Guide*.

Document Revision History

This table describes the changes to *Cocoa Performance Guidelines*.

Date	Notes
2009-08-11	TBD
2005-08-11	Fixed minor typo.
2005-07-07	Added information related to writing data in binary formats.
2005-04-29	New document that offers tips on how to improve the performance of Cocoa applications.

Index

P

R

S

T

V

W