Code Speed Performance Guidelines

Performance

2005-07-07

Contents

Figures and Listings

Introduction to Code Speed Performance Guidelines

For most users, performance means speed. If an application performs its tasks quickly, the user is happy. If an application performs tasks slowly or is unresponsive to commands, the user is likely going to get frustrated and may possibly not want to use that application.

The focus of this programming topic is improving the speed of your code, both in the real and perceived sense. In the real sense, you should measure the time it takes to complete operations and modify your algorithms and loop code to be as efficient as possible. In the perceived sense, you should make your application appear fast to the user, even if an operation actually takes a long time to complete.

Organization of This Document

This programming topic contains the following articles:

- "Diagnosing Slow Operations" (page 9) describes techniques for finding which parts of your code are slow.
- "Check Your Algorithms" (page 15) provides some guidelines on how to approach speed improvements in your code.
- "Impedance Mismatches" (page 17) describes the performance impacts of translating between different data formats and tips on how to avoid such translations.
- "Perceived Responsiveness" (page 19) describes ways to make your application feel faster than it may actually be.
- "Detecting Polling Behavior" (page 21) describes a simple way to tell if your application is polling the system for information.
- "Accelerating Critical Code" (page 23) provides some practical tips on how to improve the performance of iterative code.
- "Tuning for Specific Hardware" (page 29) provides tips on how to tune your software for maximum performance on the G5 processor.

Diagnosing Slow Operations

Diagnosing slow behavior in an application requires a bit of detective work. In a few cases, poor performance may have nothing to do with your code, but in most cases your code is being inefficient in some way. When you detect a drop in your application's performance, follow the steps described in the following sections to isolate the problem.

Checklist for Diagnosing Problems

Before you start gathering data on exactly which parts of your code are slow, you should run through the following checklist to eliminate any obvious problems.

- **Are other processes slowing down the system?** Run `top` to see how much CPU time is being taken up by other processes.
- **Are specific operations slow?** Run Shark or `sample` to find out where your application is spending its time. See "Finding Time-Consuming Operations" (page 9) for more information.
- **Did your I/O patterns change significantly?** Run `fs_usage` to see if file operations are slowing down your system. For information on how to diagnose file performance issues, see *File-System Performance Guidelines*.
- **Is your application silently generating errors?** If your application is encountering errors, it may be spending much of its time handling those errors or working around them. Watch your code in the debugger or set up some error handling notifications to locate potential errors.
- **Are compiler optimizations enabled?** Build your application with compiler optimizations enabled to see if that improves performance. For information on the available compiler optimizations, see Managing Code Size in *Code Size Performance Guidelines*.
- **Is your application prebound?** If you are running a Mach-O executable on Mac OS X version 10.3.3 or earlier, prebinding can improve the performance of your application. If you are running on Mac OS X version 10.3.4 or later, prebinding might offer some gains but is less critical for performance. For information on how to enable prebinding, see Prebinding Your Application in *Launch Time Performance Guidelines*.

Running Shark or `sample` can help you quickly identify operations in your code that are taking too much time. once identified,

Finding Time-Consuming Operations

Apple provides several tools that let you sample your application at runtime to find out where it is spending its time. Sampling lets you gather information without recompiling your application. The sampling tools take a snapshot of your application's stack at regular intervals and then collect that information into a call graph of functions. This information can help you identify inefficient algorithms and slow functions.

The sections that follow describe how to use these tools and understand the data they generate.

Using Shark

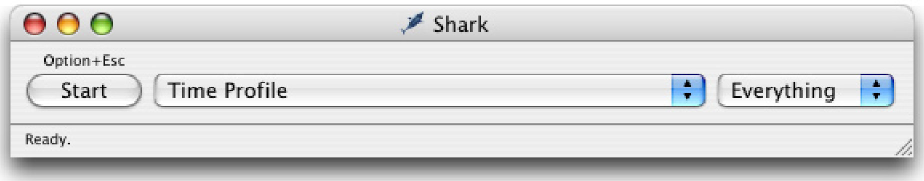
Shark is a powerful tool for finding hot spots and more subtle performance problems in your application. Shark samples either a single process or all system processes and records information about the call stacks for each process. It then displays the recorded information using tree views, charts, and other formats that can help reveal problems quickly.

Shark provides several different options for sampling processes. The most common option is the time profile, which gathers call stack data at a fixed interval and displays the most frequently called functions (the hot spots). You can also track specific function calls in your application, including malloc calls, file I/O calls. You can also gather information about specific hardware or software events, including cache misses, processor stalls, PCI requests, and page in requests.

Configuring Shark

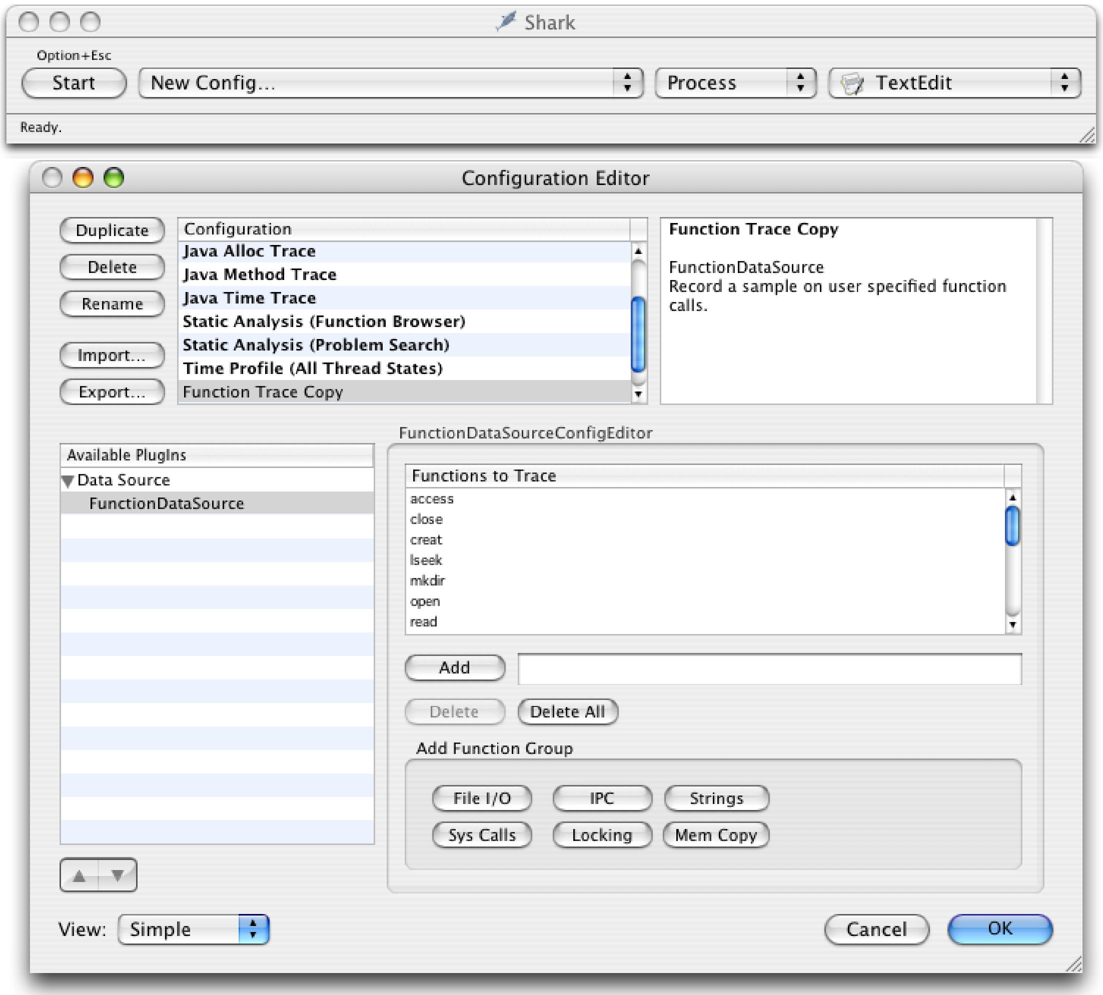
For most common operations, Shark requires little or no configuration. When you first launch Shark, the application is configured for a basic time profile, which gathers samples of all system processes at a fixed interval. You can select a different configuration preset from the sampling configuration popup menu, shown in Figure 1. When you are ready to sample, click the Start button or use the Option-Escape hot key.

Figure 1 Shark main window



If you do not want to use one of the existing configurations, you can create your own custom configurations by choosing New Config from the sampling configuration popup menu. This brings up the Configuration Editor window (Figure 2), from which you can choose the data you want to gather during sampling sessions. Configurations you create with this window are automatically added to the sampling configuration popup menu.

Figure 2 Configuration editor window



Function Trace Copy

FunctionDataSource
Record a sample on user specified function calls.

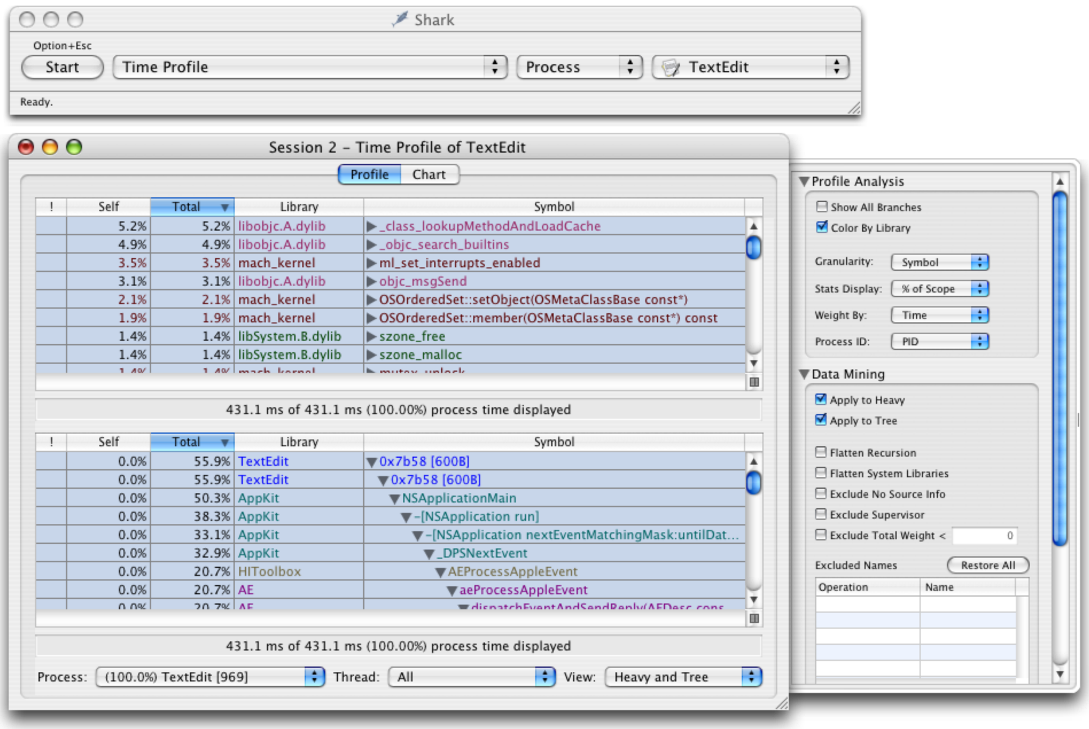
Functions to Trace: access, close, creat, lseek, mkdir, open, read

For more information about configuring the performance monitor counters, see the Shark User Manual.

Navigating Shark's Session Views

Shark provides several ways of viewing sample data and provides controls for managing the display granularity. Each session window has Profile and Chart buttons for displaying data textually or graphically. The Profile view is shown in Figure 3. In this view, you can view hot spots (heavy view), a tree view of your call stacks, or both simultaneously (as shown here). You can view call stack information for a specific process or thread or for all processes and threads. You can hide irrelevant call stack information using the Data Mining features found in the side drawer.

Figure 3 Data displayed in heavy and tree view



!	Self	Total	Library	Symbol
	5.2%	5.2%	libobjc.A.dylib	▶_class_lookupMethodAndLoadCache
	4.9%	4.9%	libobjc.A.dylib	▶_objc_search_builtins
	3.5%	3.5%	mach_kernel	▶ml_set_interrupts_enabled
	3.1%	3.1%	libobjc.A.dylib	▶objc_msgSend
	2.1%	2.1%	mach_kernel	▶OSOrderedSet::setObject(OSMetaClassBase const*)
	1.9%	1.9%	mach_kernel	▶OSOrderedSet::member(OSMetaClassBase const*) const
	1.4%	1.4%	libSystem.B.dylib	▶szone_free
	1.4%	1.4%	libSystem.B.dylib	▶szone_malloc

431.1 ms of 431.1 ms (100.00%) process time displayed

!	Self	Total	Library	Symbol
	0.0%	55.9%	TextEdit	▼0x7b58 [600B]
	0.0%	55.9%	TextEdit	▼0x7b58 [600B]
	0.0%	50.3%	AppKit	▼NSApplicationMain
	0.0%	38.3%	AppKit	▼-[NSApplication run]
	0.0%	33.1%	AppKit	▼-[NSApplication nextEventMatchingMask:untilDat...
	0.0%	32.9%	AppKit	▼_DPSNextEvent
	0.0%	20.7%	HIToolbox	▼AEProcessAppleEvent
	0.0%	20.7%	AE	▼aeProcessAppleEvent

431.1 ms of 431.1 ms (100.00%) process time displayed

Process: (100.0%) TextEdit [969] Thread: All View: Heavy and Tree

Profile Analysis
- [ ] Show All Branches
- [✓] Color By Library

Granularity: Symbol
Stats Display: % of Scope
Weight By: Time
Process ID: PID

Data Mining
- [✓] Apply to Heavy
- [✓] Apply to Tree
- [ ] Flatten Recursion
- [ ] Flatten System Libraries
- [ ] Exclude No Source Info
- [ ] Exclude Supervisor
- [ ] Exclude Total Weight < 0

Excluded Names

The heavy view shows you your program's hot spots, that is, it shows you the functions that were encountered most frequently. This view can point out places where your code is spending a lot of time. Hot spots tell only part of the story though. If a function appears to consume 50% of your program's processing time, there are two potential reasons why: it is slow or it is called too frequently by a different function. You can also use the data mining features to charge the cost of a given function to whoever called it. Doing so might point out a higher level function is the real culprit.

The tree view provides a top-down view of a process and is probably more familiar to users of the `sample` command-line tool. This view can be useful for finding high-level functions that are consuming too much CPU time. As with the heavy view, you can use the data mining features to charge the costs of a function to whoever called it.

The Chart tab of the data window shows data gathered by the performance monitor counters. For a basic time profile, the charts show call stack depth plotted over time. However, if you have additional performance counters set up, the charts display the values of those counters over time. For more information about setting up performance counters, see the Shark User Manual.

If you have source code, double-clicking a function will display a source code view for that function. The source code view provides you with a low-level performance analysis of the function code. This low-level analysis can show you how to tweak your code to get the best possible performance for of the current processor. For example, Shark can point out processor stalls or places that might benefit from parallelization through AltiVec. This analysis may not always yield big gains for your entire application but can be important in the final stages of tuning critical code.

Limitations of Shark

When gathering samples in time profile mode, it is important to remember that Shark's results are not comprehensive. Shark gathers samples only at predetermined intervals, gathering call stack information for the target threads during each interval. And while the sampling granularity in Shark is high, it is still possible for a function to be called more often than is actually reported.

To improve the data reported by Shark, you can change the sampling interval or vary the interval dynamically. Shark includes an advanced feature that automatically adds a random increment of time to the sample period to prevent harmonic phenomena, such as the same thread being active every 10 milliseconds.

Using the sample Command-Line Tool

The `sample` command-line tool provides another way to sample a process at regular intervals. The `sample` tool gathers sample data at regular intervals and creates a textual report of the call stack data, including the number of times each function was discovered. Because `sample` is a command-line tool, you can run it situations where you couldn't run Shark, such as from a remote machine.

To run `sample`, execute it from the command line, specifying the process ID of the program you want to sample along with the sample interval and duration. If you want to sample the launch of an application, specify the name of the application and the `-wait` option when calling `sample`. For more information on sampling the launch of an application, see Gathering Launch Time Metrics in *Launch Time Performance Guidelines*.

You should let `sample` complete its sampling period before killing the target process. If you think the process might die before sampling is complete, specify the `-mayDie` option when calling `sample`. With this option specified, `sample` gathers symbol information before it starts sampling to ensure that it can display that information in its report. Without this symbol information, you may be unable to decipher the call-graph data.

Doing a More Thorough Analysis

Statistical sampling can provide you with good insight into how much time an application is spending doing something. However, given the nature of statistical sampling, there is always a possibility that the data you receive is somewhat misleading. It happens rarely, but if you really want to know exactly which functions are called, how often they are called, and how long they take to run, you need to instrument your code. To do this, you must profile your code using `gprof`.

For instructions on how to profile your code with `gprof`, see Improving Locality of Reference in *Code Size Performance Guidelines* or the `gprof` man page.

Analyzing Sample Data

Once you've gathered some data from Shark or `sample`, how do you use it to find performance problems in your code? If the problem is really in your code, then you should be able to get enough information from either program to find the problem. One way to identify that information is to do the following:

1. If you are using Shark, look at the heavy view to see if your code is included in the hot spots. If your program needs only minor tuning, your code may not immediately appear in the hot spots. Try using Shark's data mining capabilities to hide system libraries and frameworks. That might reveal the hot spots in your own code.

2. If the heavy view does not reveal any clear hot spots, use the tree view of either Shark or `sample` to find the heaviest branches. Follow each branch down until you reach your own code so that you can determine what high-level operation was being performed. Use that as the starting point for tuning that particular operation.

3. Within the code of each heavy branch, walk down through any heavily called functions and examine the work you are doing:

   - Is your algorithm efficient for the amount of data you are processing? (See "Check Your Algorithms" (page 15)).
   - Are you calling a lot of other functions? If so, you might be trying to do too much work and might benefit from delaying that work or moving it to another thread. (See *Threading Programming Guide*.)
   - IAre you spending a lot of time converting from one data type to another? Perhaps you should modify your data structures to avoid the conversions altogether. (See "Impedance Mismatches" (page 17).)

4. After tuning each branch, run Shark or `sample` again to see if you successfully removed or reduced the problem. If problems persist, keep tuning other branches or start tuning the parent code that calls those branches.

Check Your Algorithms

Choosing the right algorithm for a task can have significant impacts on performance. If an algorithm doesn't scale to the amount of data in the system, your application can appear slow and unresponsive. The following sections help you identify potential problems with your algorithms and things you can do to fix those problems.

Measure First

While it is possible to choose the right algorithm right away, you'll never know it's the right one until you measure its performance under different load situations. You should always gather metrics for your code before you attempt to go back and tune any algorithms. Metrics tell you first and foremost *whether* you have a performance problem. Only after you've determined there is a problem should you try to figure out the best way to fix it.

When you gather performance metrics, remember that the apparent speed of the operation is not the only measurement. Memory usage is another measurement to consider. If an operation allocates a lot of memory, it may not perform as well under low-memory conditions or when the system has to do a lot of paging. You should try running your application under these conditions and gather more data.

Sampling is one way to gather data for your application. Sampling tells you where your application is spending its time. For information on the available sampling tools, see "Finding Time-Consuming Operations" (page 9).

Tuning at the Right Level

Whenever you analyze sample data from your application, you should always try to differentiate between the cost of the function being called and the usage of that function. Suppose you sample your executable and determine that it is spending too much time in one particular function. This tells you something about the general location of a performance problem but does not tell you exactly where that problem lies. In this situation, there could be several possible reasons for time being spent in that function, including the following:

- The function could be poorly optimized.
- The parent function could be calling the child function more times than it really needs to. Thus, the parent function needs optimization.
- The thread may be blocked and the statistical sampling tool is seeing the function many times when in fact no code is actually executing.
- The function may be very fast but called at a regular interval that happens to match the sampling interval.

Keep in mind that even if a function has a high cost, it's also possible that you can reduce its usage as well. Think about the design of your high-level algorithms and make sure that they are performing only those tasks that are absolutely required. Solving performance issues in your high-level algorithms can have a much greater impact than tuning individual functions. For example, eliminating a function call saves much more time than simply tuning that function.

The data mining features of Shark can help you view your data set in ways that might make it easier to see the real problems. Using the data mining features, you can remove symbols over which you have no control, such as those found in system libraries. Doing so applies the costs incurred by those symbols to the function that called them. This could point out places where your code is calling system routines too frequently. Reducing the number of system calls (or providing a different implementation) can significantly reduce the overall time spent in your own function.

For more information about Shark's data mining features, see the Shark User Manual.

Avoid Costly Algorithms

In operations involving anything other than small amounts of data, operations that involve quadratic or worse algorithms are generally a bad choice. Any time your algorithm speed scales at anything above a linear rate to the number of elements, you should reconsider the benefit of that algorithm.

When choosing an algorithm, it's important to know the intended data set for that algorithm. If you're dealing with a data set that contains anywhere from ten to ten million records, then it's worth the time to code an algorithm with a linear or logarithmic performance. The effort to do so is worth the resulting performance gains. However, if you know you'll always be dealing with a small number of records, the implementation time for a quadratic algorithm might make it more attractive than a more complex algorithm.

Avoid Calls to the Shell

Whenever possible, avoid using the `system` function to execute strings in the local shell. The `system` function sends a string to the shell's command-line interpreter and is an expensive operation to perform from your own code. Depending on the features you need, it might be better to implement them directly in your code or see if there is a more direct way to get what you need. For example, you might see if the target program accepts socket-based connections or has an API to do what you need.

Impedance Mismatches

In electronics, an impedance mismatch refers to a mismatch between the output signal from one component and the input signal expected by another component. In programming, this term is used in a similar way to refer to a mismatch between data structures in your code.

Each library and framework typically defines its own data structures for managing information. When the framework exposes a function, it may also expose any custom data structures that are parameters or return values for that function. If you call that function in your code, you must pass it the data structure it is expecting to see. If you do not store a copy of that data structure in your own code, then you have to create it and populate it with information before making the function call.

Converting between system-defined data structures and any custom data structure formats used by your code wastes CPU time that could be spent doing other things. Before you write any code for your algorithms, carefully consider what data you need to operate on and design your data structures accordingly.

Use Existing Data Structures

When you are getting ready to design your code and data structures, you should think carefully about how your code will interact with external code. If your algorithm calls for passing a particular data structure back and forth many times to an external library, you might want to design your algorithms to work directly with the data structures from that library.

As with any performance optimization, you should carefully consider whether matching the data structures of external frameworks is appropriate. Using the native data structure of an external library might give your code a slight speed boost in passing data back and forth, but if it slows down your algorithm it is a wasted gain. You should always measure the performance impact of any optimization you put in place and make sure it is an improvement rather than a regression.

Avoid Floating-Point to Integer Conversions

Converting back and forth between integer and floating-point values can slow down performance, particularly on the G5 processor. On the G5, type conversions of this sort can cause bubbles in the instruction pipelines as the processor hits the L1 cache to convert the data. If your code currently performs these types of conversions, you should consider the following options instead:

- Avoid the conversions altogether by staying in one domain (either integer or floating-point).
- Use Velocity Engine (AltiVec), where type conversions are done in registers rather than in memory.
- Try compiling with the GCC `-fast` option. (Note that this option optimizes for the G5 processor by default. To optimize for G4 processors, you must also pass the `-mcpu=7450` option to GCC.)

One way to avoid type conversions altogether is to use a shadow variable. This technique is useful in situations where you would otherwise have to cast back and forth between types. Instead of casting, you create a duplicate variable of the needed type and use it in the same way as the other variable.

Listing 1 shows the use of a shadow variable in a simplified example. The original code would cast integer `i` to a `double` and then add it to `sum`. Rather than add integer `i` to `sum` during each loop iteration, the code maintains a shadow copy of `i` and adds that value to `sum`. The change resulted in code that was three times faster than the original version on a G5 processor.

Listing 1 Using shadow variables

```
double calculateDoublePrecisionSum(int numIterations)
{
    double sum = 0.0;
    int i;
    double i_fp; // shadow variable for i

    for (i = 0, i_fp = 0.0; i < numIterations; i++, i_fp++)
    {
        sum += i_fp;
    }

    return sum;
}
```

Core Foundation Calls

If your application is implemented using Cocoa, you can take advantage of the Core Foundation toll-free bridged types to improve performance of repetitive operations. Many methods in the Foundation Kit framework have equivalent functions in the Core Foundation framework. These equivalent functions can take either a Core Foundation type or a Foundation Kit object. Because function calls have a slight performance advantage over message dispatches, you might see a measurable gain by calling the Core Foundation function instead.

When substituting Core Foundation function calls for Foundation Kit methods, make sure that you handle any exceptional cases. Many Core Foundation functions are faster because they do not perform as much error checking as their Foundation Kit equivalents. Passing a null object to a Foundation Kit method may cause the method to return a null value back. Passing a null object to a Core Foundation function may cause a crash.

Perceived Responsiveness

The reason for improving application performance is to make an application seem more responsive to the user. But sometimes there is only so much you can do to improve the actual performance of your code. In situations like that, improving the perceived responsiveness of your application can satisfy the user just as if you had made the actual improvements.

Important: Improving your application's perceived responsiveness is not a panacea for fixing slow code in your application. It is merely one tool for improving the end user experience. You should still make the effort to improve the actual efficiency of your application, as that will have longer-lasting effects.

Launch Time

Launch time is an important place to make your application seem fast, as it is the one time when the user typically waits for your application code to finish. The best way to make your application seem fast is to display your menu bar and main window as fast as possible.

When an application is launched, it is put into its initial state. In most cases the application need only make itself ready for the user. Rather than spend your time loading resources and initializing subsystems, find ways to defer your initialization code until the subsystems that need it are actually used. Not only does this reduce the amount of startup overhead for your application, it keeps your memory footprint low.

For information on improving launch-time performance, see *Launch Time Performance Guidelines* in Performance Documentation.

Lengthy Operations

If your application needs to perform a lengthy operation, try to do so in a way that does not restrict the user from performing other actions. Using multiple threads to perform tasks in the background is one way to make sure your user interface is responsive. Having multiple threads can also allow your application to take advantage of multiple processors to improve performance.

Using threads is not without its costs, however. Threads add to your application's memory footprint because of the space required for the thread's stack. Background threads need to communicate with your main thread or with other threads in situations where there might be resource contentions. You may need to use locks to ensure that each of your application's threads do not interfere with each other. These operations can be costly in their own right and should be used where there is a definite performance advantage.

An alternative to using threads is to use timers, which can call your code at fixed intervals to perform the operation. Timers have much less overhead than threads and are especially useful for operations that can be broken down into small chunks and executed incrementally over a longer period of time. Timers do suffer from the same resource restrictions as threads however. If the operation requires exclusive access to any resources, your code must use a lock to protect those resources until it is done with them.

For information on using threads, see *Threading Programming Guide*.

Avoid the Spinning Cursor

One way to tell if your application is unresponsive is to count how often you see the spinning cursor appear. Mac OS X displays the spinning cursor automatically when your application fails to process an event within a 5 seconds. The spinning cursor is a way to let the user know that your application is busy. It is also a way to let you know that your application may be taking too long to do something.

The reasons for seeing the spinning cursor vary and can range from processing large amounts of data to waiting for a response from the network. The best way to find out what's happening is to launch Spin Control and leave it running while you test your application. Spin Control samples your application whenever the spinning cursor appears. You can use the data gathered by Spin Control to find where your application was spending its time when it was unresponsive and correct the problem.

When processing large data sets, there are several techniques to avoid the spinning cursor. One technique is to do your processing on a separate thread of execution. This is the most general approach since it can be applied to most data sets. However, it does require extra overhead and communications to manage the thread. If the data can be factored into small chunks, you might have your application process a chunk at a time when no events are pending.

If your application is waiting for a response from a function call, you may be using the wrong function. Look through the API documentation for functions that perform the same task asynchronously.

Detecting Polling Behavior

Polling is an inefficient way to process events. Not only does it waste CPU time, it ties up memory with code that could otherwise be paged out and used by other programs.

If you're not sure your application is polling, you can find out very quickly. Put your application in an idle state and do one of the following:

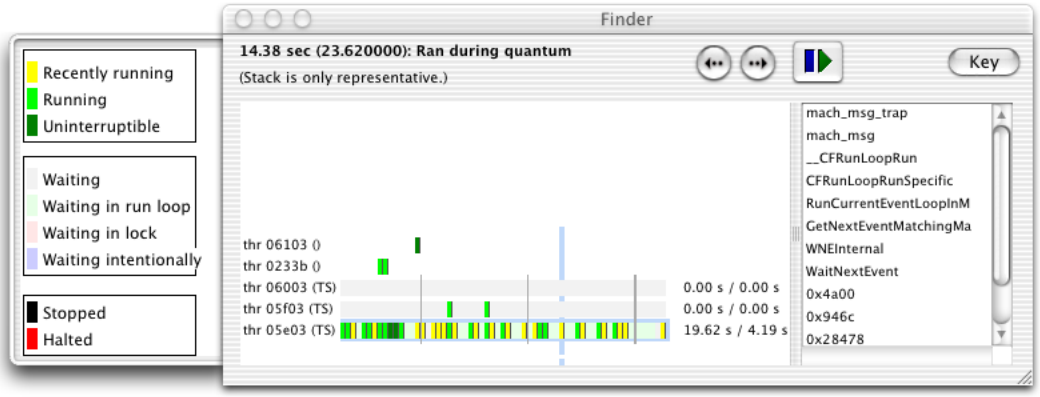
- Run `top` and make sure the `%CPU` field shows 0.
- Run Thread Viewer and see if any of your application's threads are active.

Checking the `%CPU` field in `top` is a quick way to determine *if* your application is polling. You can also run `top` with the `-d` option, which displays the number of context switches, messages, and system calls made by your application. If these numbers change over time, your application is polling the system in some way. Regardless of how you use it, `top` does not tell you which part of your application is responsible for polling. For that, you need to use Thread Viewer.

Thread Viewer graphically displays the activity of each of your application's threads along a color-coded time-line view. Clicking on the time line displays a backtrace of function calls made by the selected thread at that point in time. This backtrace is only a snapshot and may not reflect the current activity of the thread but it can help isolate the location of polling code.

Figure 1 shows the Thread Viewer display window. The drawer on the left side provides a key for interpreting the time-line information. Clicking in the time-line shows the stack trace in the right-side scroll box.

Figure 1 Thread Viewer display window



14.38 sec (23.620000): Ran during quantum
(Stack is only representative.)

```
mach_msg_trap
mach_msg
__CFRunLoopRun
CFRunLoopRunSpecific
RunCurrentEventLoopInM
GetNextEventMatchingMa
WNEInternal
WaitNextEvent
0x4a00
0x946c
0x28478
```

Accelerating Critical Code

For code that is called frequently by your application, even small optimizations can have a significant impact on performance. The following sections provide some simple ways to speed up repetitive operations.

As with any optimization, you should always measure the initial performance of code you plan to optimize. Taking an initial set of measurements helps you identify whether optimizing the code is warranted, and if it is, provides you with a set of baseline metrics against which to compare your changes. Without these metrics, there is no way to tell if your optimizations are an improvement or a regression from the original implementation.

Note: The following sections cover software-only tuning options. For hardware-specific tuning options, see "Tuning for Specific Hardware" (page 29).

Speeding Up Loop Operations

Because of their nature, loops are a good place to start looking for potential optimizations. The code in a loop operation is going to be performed multiple times in quick succession. If your operation is spending a lot of time inside of a single loop, you should look for ways to remove code from that loop.

The simplest improvement you can make is to remove invariant code from the body of a loop. In some special cases, you might even be able to remove the loop altogether and replace it with a more efficient implementation.

Removing Invariant Code

When you write loop code, try to remove any invariant operation, that is, operations whose outcome is the same each time. For example, if you have some mathematical equation in your loop, you might want to rearrange your equation so that you can precompute any constant values or perform those computations outside of the loop. Similarly, if you know that a particular function returns the same value each time it is called, move it outside the loop and use variables to store any needed values.

For example, suppose you have a loop that performs the same action on the items in an immutable array. You could write your code as follows to walk through the contents of the array and perform the action as follows:

```
for (i = 0; i < [myArray count]; i++)
{
    object = [myArray objectAtIndex: i];
    [object doSomething];
}
```

While this code works just fine, it is not the most efficient choice. In each loop iteration, it dispatches a message to get the number of items in the array, which is wasteful. If the number of items in the array never changes, you could assign that value to a variable and use that instead, as shown in the following code:

```
numItems = [myArray count];
for (i = 0; i < numItems; i++)
{
    object = [myArray objectAtIndex: i];
    [object doSomething];
}
```

Removing this one function call can improve performance in any loop, regardless of how many items are in the array. This technique also works for any technology as removing function calls means less code to execute during each loop iteration.

Unrolling Loops

The process of unrolling a loop is a delicate one and should be approached very carefully. The only time you should consider this option is when doing so simplifies your loop code significantly. Even in the best situations, make sure to go back and evaluate the real-time performance of your unrolled loop code. Unrolling loop code usually leads to more code, which increases the size of your application's memory footprint and can increase the possibility of paging.

One case where removing a loop can increase speed is in a Cocoa application where you have an array of objects and you want to send the same message to each object. NSArray implements the `makeObjectsPerformSelector:` and `makeObjectsPerformSelector:object:` methods for that exact purpose. In this case, the method performs the loop for you, using its knowledge of the array's internal data structures to optimize the loop performance.

Caching Method Implementations

Whenever you send a message to an Objective-C object, the runtime system must perform a lookup to determine which selector to use for that message. While the Objective-C runtime is very fast at performing this lookup, the operation still takes a small amount of time. If you want to call the same method on a collection of objects, you can eliminate that lookup cost altogether by caching the method's `IMP` pointer and calling it directly. Remember that the objects in the collection must be of the same type and have the same method implementation.

Important: Caching `IMP` pointers should be done only if you have measured a specific performance problem in a critical loop. In most situations, caching pointers is unnecessary and can make your application hard to maintain by inhibiting the dynamic nature of the Cocoa runtime system.

To cache an `IMP` pointer for an object derived from NSObject, call the `methodForSelector:` method of the object and store the returned value. The code in Listing 1 (page 25) shows you how to get the `IMP` pointer and use it to call the method for a specific object. In this example, the last two statements are equivalent to a method invocation. The first of these statements does the method lookup, obtaining a pointer to the implementation of the method. The second statement calls the method implementation with the desired search parameters.

Listing 1 Caching IMPs

```
#import <Foundation/Foundation.h>
#include <objc/objc-class.h>

static void DoSomethingWithString( NSString *string )
{
    typedef NSRange (*RangeOfStringImp)(id object, SEL selector,
                                NSString * string, long options, NSRange range);

    NSRange             foundRange;
    NSRange             searchRange;
    RangeOfStringImp    rangeOfStringImp;

    searchRange = NSMakeRange(0, [string length]);

    // The following two lines of code are equivalent to this method invocation:
    // foundRange = [string rangeOfString:@"search string"
    //                      options:0
    //                      range:searchRange];

    rangeOfStringImp = (RangeOfStringImp)
                [string methodForSelector:@selector(rangeOfString:options:range:)];

    foundRange = (*rangeOfStringImp)(string,
                                    @selector(rangeOfString:options:range:),
                                    @"search string", 0, searchRange);

}
```

Notifications

Notifications are a simple way to communicate changes within your application or to another application. However, you should carefully consider the performance implications of using notifications and avoid their overuse.

The fewer notifications you send, the smaller the impact on your application's performance. Depending on the implementation, the cost to dispatch a single notification could be very high. For example, in the case of Core Foundation and Cocoa notifications, the code that posts a notification must wait until all observers finish processing the notification. If there are numerous observers, or each performs a significant amount of work, the delay could be significant.

Another case where delivery cost for notifications is high is distributed notifications. If multiple processes register to receive a notification, the delivery of that notification might require bringing idle processes back into memory to handle it. This action has an effect both on CPU usage and on memory usage as processes are paged in to respond to the notification.

Note: For additional information related to tuning your notification code in a Cocoa application, see *Cocoa Performance Guidelines*.

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 most Core Foundation and Cocoa 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 might further delay the event that triggered the notification.

If you must perform additional work upon receiving a notification, consider deferring that work until later. Set a flag, use a timer, or do anything you can to return control back to the poster of the notification as quickly as possible.

Suspending Distributed Notifications

If your application is an observer for distributed notifications, and you do not want to receive those notifications when your application is not frontmost, be sure to specify that information when you register for the notification. Receiving notifications when your application is not frontmost can have a negative impact on performance because it might involve bringing your application back into memory to handle the notification. The distributed notification centers implemented by Core Foundation and Cocoa both give you the option to hold or drop notifications that come in while your application is inactive.

For more information about options for receiving distributed notifications, see the documentation for the `CFNotificationCenterAddObserver` method of Core Foundation or the `addObserver:selector:name:object:suspensionBehavior:` method of Cocoa's NSDistributedNotificationCenter class.

Use Darwin Notifications for Maximum Performance

If you find that the Cocoa or Core Foundation notification systems are inadequate for your performance needs, try using the Darwin notification system instead. The Darwin layer defines a basic set of notifications that allow fast communication among multiple processes. Notifications can be delivered automatically using a mach port, signal, or file descriptor. Although the system is much simpler than the ones offered by Core Foundation and Cocoa, it is also extremely lightweight and fast.

Another important feature of the Darwin notification system is the ability for clients to receive notifications manually. Unlike most notification mechanisms, which interrupt the observer to deliver the notification, your application can choose when it wants to receive Darwin notifications. If you are using notifications simply to communicate changes, this feature can offer tremendous performance advantages over the automatic delivery of notifications. For example, this is an excellent mechanism for notifying an application that a set of shared data has been modified and needs to be recached. You would not want to use this mechanism to respond to the occurrence of a specific event.

Important: When creating new Darwin notification tokens, be sure to include the current session ID to distinguish it from tokens in other user sessions. For more information about user sessions and fast user switching, see *Multiple User Environments*.

For more information about using Darwin notifications, see the `man 3 notify` page. For API reference information, see also *Darwin Notification API Reference* and the `notify.h` header file

Tuning for Specific Hardware

Not all processors are alike. Each new processor design brings with it a new way of thinking about your code and new techniques for improving performance. By taking advantage of the latest features on the newest processors, you can often see significant speed increases in your software. If you support the right features, you can also gain speed on the new processor without losing speed on older processor models.

The following sections offer tips primarily aimed at improving performance on the G5 processor. However, using these techniques should not hurt performance on older processors. Most of the techniques simply make it easier for the compiler and instruction scheduler to tune your code.

Avoid Instruction Scheduling Problems

The G5 processor uses a massively parallel execution core to perform multiple instructions simultaneously. In addition to Velocity Engine support, the processor includes two separate floating-point instruction units, two integer processing units, and several other units for managing the flow of instructions. Maximizing the performance of your software means keeping these instruction units busy as much as possible. This means you need to write your code with the following in mind:

- Do more work in parallel. Consider intermixing unrelated floating-point and integer-based operations to keep more instruction units busy.
- Manually unroll important loops, or use the `-funroll-loops` option with GCC. Partially unrolling a loop might let you do more work within each loop iteration.
- Enable instruction scheduling in your Xcode project, or pass the `-mtune=G5` option to GCC.

Bottlenecks in the execution of G5 instructions often occur because code was written with a serial flow in mind. If your code computes a number of similar (but independent) values, it is advantageous to arrange your code in a way that lets the instruction scheduler fill the instruction unit pipelines.

Note: Shark is an excellent tool for identifying and fixing instruction latency issues in your code. For more information about Shark, see the Shark User Guide.

Consider the simple function in Listing 1, which computes a sum and returns the value. This function takes advantage of only one instruction unit, which leaves other instruction units sitting idle.

Listing 1 Computing a sum the slow way

```
double ComputeSum_slow(int numIterations)
{
    int i;
    double sum = 0.0;

    for (i = 0; i < numIterations; i++)
    {
        sum += 1.0;
    }
    return sum;
}
```

If the number of iterations is guaranteed to be large enough, consider what happens if you take this code and partially unroll the loop. Listing 2 shows an updated version of this code, but in this revised edition the loop now performs eight floating-point operations through each iteration. The instruction scheduler sees this as a way to fill the pipelines of both floating-point instruction units. Although the same amount of work is being done, the distributed nature of the work results in code that is up to 10 times faster than the original.

Listing 2 Computing a sum in parallel

```
double ComputeSum_fast(int numIterations)
{
    double sum0, sum1, sum2, sum3, sum4, sum5, sum6, sum7;
    int i;

    sum0 = sum1 = sum2 = sum3 = sum4 = sum5 = sum6 = sum7 = 0.0;

    for (i = 0; (i+7) < numIterations; i += 8)
    {
        sum0 += 1.0;
        sum1 += 1.0;
        sum2 += 1.0;
        sum3 += 1.0;
        sum4 += 1.0;
        sum5 += 1.0;
        sum6 += 1.0;
        sum7 += 1.0;
    }
    return (sum0 + sum1 + sum2 + sum3 +sum4 +sum5 + sum6 + sum7);
}
```

Although the preceding example shows a simple case, it hopefully demonstrates the effect of doing more work in parallel. Applied to your own code, you should be able to find similar improvements by breaking out parallel calculations. Especially for critical operations, such as large scientific calculations, this kind of optimization can lead to tremendous performance gains.

Fix Floating-Point Alignment Issues

To process floating-point values efficiently, processors typically require that they be aligned along certain memory boundaries. Floating-point alignment is especially important for the G5 processor, where misaligned values can cause a processor exception. Given that Carbon and Cocoa both use floating-point numbers extensively for working with graphical elements, it is important (and relatively easy) to ensure correct alignment of floating-point values in your compiled code.

To ensure that floating-point values are aligned properly, add the GCC compiler option `-malign-natural` to your project's build settings. This option causes the compiler to align floating-point values along their natural boundaries. Although there are other options for doing floating-point alignment, the `-malign-natural` option is preferred because it handles all of the important types, including `double` floating-point values. For more information about this option, see the `gcc` man page.

Access Memory Contiguously

As processor speeds increase, so does the latency for accessing memory. To help alleviate this problem, the G5 processor includes a hardware prefetch engine to get data into the processor caches before it is needed. However, taking advantage of this prefetch engine requires you to do the following:

- Pack your data structures together to improve their locality.
- Walk through your data structures contiguously so that the hardware prefetch engine can stream data in just before you need it.

G5 cache lines are 128 bytes long. If your data structures are tightly packed, the prefetch engine can load the entire structure into the fewest number of cache lines. This improves both the latency in loading the cache and your cache usage, since more useful memory is in the cache at the same time.

You also need to be careful about accessing memory in a contiguous manner. For example, if you need to iterate over the entries in a two-dimensional array of data, there are two ways to do it. You can walk the columns of the first row, followed by the columns of the second row; or, you can walk the first element of each row, followed by the second element of each row. Because of the organization of memory, walking the columns of the first row, followed by the columns of the second row is much more efficient because the column data is contiguous. Walking an array in this order is often many times faster than walking down a single column of data.

Tuning With Velocity Engine

The Velocity Engine (also known as AltiVec) is a 128-bit vector execution unit embedded in the G4 and G5 processors. This unit lets you perform highly parallel operations, such as high-bandwidth data processing (for streaming video) and algorithmically intensive computations used in graphics, audio, and mathematical operations. If you perform any operations of this nature, you should incorporate Velocity Engine support into your application.

In many cases, all you need to do to take advantage of Velocity Engine is link with the right frameworks and libraries. Mac OS X uses Velocity Engine to implement accelerated support for the following types of operations:

- Digital signal processing—Fast Fourier Transforms, convolutions, squares, and more
- Vector Image Processing—resize, distort, convolution, morphing, alpha compositing, format conversion, and other operations on images
- Basic Linear Algebra Subprograms—vector-scaling linear algebra, matrix-vector linear algebra, and matrix operations
- Linear Algebra operations—linear equation computations, find least-square solutions of linear systems of equations, solve eigenvalue problems, and perform many other operations from the LAPACK library
- Vector Mathematics—computational functions such as divides, square roots, and exponential functions.
- Basic Algebraic operations—basic algebraic operations on operands up to 128 bits in size
- Big Number operations—basic math, shift, and rotate operations on operands ranging in size from 256 bits to 1024 bits.

The Accelerate framework (introduced in Mac OS X version 10.3) coalesces support for these operations in a single framework. If your software supports versions of Mac OS X earlier than 10.3, you might need to include several separate libraries and frameworks instead.

Determining if Velocity Engine Is Available

If you choose to write your own custom code using Velocity Engine instructions, you should always check to make sure the feature is available on the current hardware. Although most newer computers support Velocity Engine, some older computers based on the G3 processor might not. If you execute Velocity Engine instructions on one of these older computers, your program will crash.

To check whether Velocity Engine is available, you can either use the Gestalt feature in Core Services or use the `sysctl` function. To use the Gestalt feature, query the system using the `gestaltPowerPCProcessorFeatures` selector, which is defined in `Gestalt.h`. To use the `sysctl` function, you would write a function similar to the one in Listing 3.

Listing 3 Checking for Velocity Engine availability

```
Boolean HasVelocityEngine(void)
{
    int mib[2], hasVE;
    size_t len;

    mib[0] = CTL_HW;
    mib[1] = HW_VECTORUNIT;
    len = sizeof(hasVE);
    sysctl(mib, 2, &hasVE, &len, NULL, 0);
    return (hasVE != 0);
}
```

Although checking for the availability of vector instructions is sufficient for most developers, if you do any data streaming in your application, you should also check to see if the `dcba` instruction is available as well. Gestalt and `sysctl` both offer ways to tell if this instruction is available. For more information, see the *Gestalt Manager Reference* or the `sysctlbyname` man page.

Note: The functions of the Accelerate framework automatically check for the availability of Velocity Engine and execute code appropriate for the target processor. You do not need to check for the availability of this feature before calling these functions.

Conditionalizing Velocity Engine Code

If you intend to support computers that run on the G3 processor, but still want to include Velocity Engine support for more modern computers, you must write your code in a way that supports both platforms. Supporting both platforms is relatively easy as long as you remember not to include Velocity Engine code in any function that might execute on the G3 processor. For example, the following code would cause an illegal instruction exception on a G3 processor:

```
void MyUnsafeFunction()
{
    if (HasVelocityEngine())
    {
        // Velocity Engine instructions
    }
    else
    {
        // non-VE implementation
    }
}
```

The reason this code causes an exception is that the compiler generates some implicit instructions at the beginning of the function to modify some Velocity Engine registers. A better way to rewrite this code is as follows:

```
void MySafeFunction()
{
    if (HasVelocityEngine())
    {
        MyVelocityEngineFunction();
    }
    else
    {
        // non-VE implementation
    }
}
```

In this revised implementation, you can guarantee your Velocity Engine instructions are executed only if the vector unit is present.

Tuning Your Java Code

If you are developing Java applications for Mac OS X, there are several things you can do in your programs to reduce performance problems. The following sections list some of the basic things you can do in your code. For additional tips, see the performance documentation on the Sun website (http://java.sun.com/docs/performance/).

Eliminate Synchronization Issues

In large, threaded programs, synchronization is often unavoidable but can be a significant performance penalty if you are not careful. In earlier Java Virtual Machines (JVMs), synchronization used to be an extremely expensive operation. In most modern JVMs, including the HotSpot (TM) Java VM in Mac OS X, only synchronized methods that lead to contention are expensive.

It is a good idea to measure your program's performance and while doing so try to identify any highly contended objects. Wherever you find such objects, consider redesigning or re-implementing your code to avoid that contention. If you manage your data structures carefully, either by restricting that data to a single thread or using `java.lang.ThreadLocal` to maintain per-thread data, you can avoid many contention issues and increase performance.

Allocate Small Objects Efficiently

In earlier JVMs, object allocation was very expensive for a very simple reason. The garbage collection algorithms employed in these virtual machines operated conservatively by walking the entire heap to look for object references. Because all Java objects are allocated on the heap, each new allocation linearly increased the workload of the garbage collector.

The HotSpot (TM) Java VM in Mac OS X now uses a generational garbage collection algorithm. This algorithm is fast. During each garbage collection pass, objects without references are cleaned up at no cost because they are simply never copied. Object allocation in the new JVM is also much faster because it uses an atomic pointer increment.

For your own programs, you should pick a garbage collection algorithm that works best with the allocation patterns your program uses. Information about tuning your program's garbage collection algorithm, as well as other performance-related Java information, is available on the Sun website at http://java.sun.com/docs/performance/.

Avoid the Overuse of Exceptions

Exception handling in Java is very slow. Unnecessary exception handlers, in particular, make code slightly slower and much larger. Even in places where exception handlers are necessary, handling those exceptions is a very expensive operation.

As you write Java code, use exceptions only for truly exceptional cases. Do not use exceptions to indicate simple errors from which your code could otherwise recover. Instead, use them only to indicate abnormal conditions that your code does not know how to handle.

Document Revision History

This table describes the changes to *Code Speed Performance Guidelines*.

Date	Notes
2005-07-07	Added an article containing Java tuning tips. Added guidance on calling out to the shell.
2005-04-29	Updated tool descriptions. Added an article that covers tuning tips for specific hardware. Added an article covering the performance of notifications.
	Document name changed. Old title was *Optimizing Your Code For Speed*.
2003-07-25	Added information about the CHUD tools.
2003-05-15	First revision of this programming topic. Some of the information appeared in the document *Inside Mac OS X: Performance*.

Index