
Performance Overview

Performance



2010-05-14



Apple Inc.
© 2004, 2010 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

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

Apple, the Apple logo, Carbon, Cocoa, Cocoa Touch, Finder, Instruments, iPhone, Mac, Mac OS, Macintosh, Objective-C, Pages, Quartz, QuickDraw, Velocity Engine, and Xcode are trademarks of Apple Inc., registered in the United States and other countries.

IOS is a trademark or registered trademark of Cisco in the U.S. and other countries and is used under license.

Intel and Intel Core are registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks are trademarks or registered trademarks of Sun

Microsystems, Inc. in the U.S. and other countries.

OpenGL is a registered trademark of Silicon Graphics, Inc.

PowerPC and the PowerPC logo are trademarks of International Business Machines Corporation, used under license therefrom.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Introduction **Introduction 7**

- Who Should Read This Document 7
- Organization of This Document 7
- Providing Feedback 7
- See Also 8

Chapter 1 **Developing for Performance 9**

- What Is Performance? 9
 - The Efficient Use of Resources 9
 - The Perception of Speed 11
- Tracking Performance 12
 - Establish Your Baseline Metrics 12
 - Measure Early, Measure Often 12
 - Analyze Your Results 13

Chapter 2 **Basic Performance Tips 15**

- Common Areas to Monitor 15
 - Code for Your Program's Key Tasks 15
 - Drawing Code 15
 - Launch Time Initialization Code 16
 - File Access Code 16
 - Application Footprint 16
 - Memory Allocation Code 17
- Fundamental Optimization Tips 17
 - Be Lazy 17
 - Take Advantage of Perceived Performance 18
 - Use Event-Based Handlers 18
 - Improve the Concurrency of Your Program's Tasks 19
 - Use the Accelerate Framework 19
 - Modernize Your Application 19

Chapter 3 **Performance Tools 21**

- Installing the Apple Tools 21
- Key Applications 21
 - Instruments 21
 - Shark 22
- Analysis Tools 23
- Monitoring Tools 23

Hardware Analysis Tools 24
Additional Command-Line Tools 25

Chapter 4 **Doing an Initial Performance Evaluation 27**

Using top 27
Using Instruments 30
Using Shark 33
Using Quartz Debug 36
Using Spin Control 36

Document Revision History 39

Index 41

Figures, Tables, and Listings

Chapter 3 **Performance Tools 21**

Table 3-1	Analysis tools	23
Table 3-2	Monitoring tools	24
Table 3-3	CHUD tools	24
Table 3-4	Command-line tools	25

Chapter 4 **Doing an Initial Performance Evaluation 27**

Figure 4-1	Choosing an Instruments template	31
Figure 4-2	Configuring the target executable	32
Figure 4-3	Examining the recorded data	33
Figure 4-4	Shark launch window	34
Figure 4-5	Shark profile window	34
Figure 4-6	Shark code display	35
Figure 4-7	Quartz Debug options	36
Figure 4-8	Spin Control main window	37
Figure 4-9	Spin Control sample window	37
Table 4-1	Output from top using the -w option	28
Table 4-2	Output from top using the -d option	29
Listing 4-1	Typical output of top	28

Introduction

Performance is an important design factor in all software products. If a program runs slowly or displays a spinning cursor, users are likely to become frustrated with the program and look for alternatives. Maintaining a reasonable level of performance requires some diligence on your part, but the earlier you start considering it, the easier it is to catch and fix problems.

Who Should Read This Document

Performance Overview is an essential guide for developers who are new to the area of software performance analysis. This document gives an overview of the factors that govern performance and offers an approach for identifying and fixing common performance problems. It also introduces you to the specific tools and documentation you can use to identify and fix performance problems.

Organization of This Document

This document has the following chapters:

- [“Developing for Performance”](#) (page 9) describes the factors that constitute performance and the approaches to achieving the best performance in your software.
- [“Basic Performance Tips”](#) (page 15) describes the common areas of your code to analyze and offers some fundamental performance techniques.
- [“Performance Tools”](#) (page 21) describes the available tools for doing a performance analysis of your program.
- [“Doing an Initial Performance Evaluation”](#) (page 27) walks you through the basics of some key tools and shows you how to use them to find performance problems.

Providing Feedback

If you have feedback about the documentation, you can provide it using the built-in feedback form at the bottom of every page.

If you encounter bugs in Apple software or documentation, you are encouraged to report them to Apple. You can also file enhancement requests to indicate features you would like to see in future revisions of a product or document. To file bugs or enhancement requests, go to the Bug Reporting page of the [Apple Developer website](#):

<http://developer.apple.com/bugreporter/>

To file bugs, you must be registered as an Apple Developer. You can obtain a login name for free by following the instructions on the [Apple Developer Registration page](#).

See Also

In addition to this document, there are several documents that cover more specific aspects of performance. You should investigate these documents for detailed tips on how to analyze and solve performance problems.

- *Code Size Performance Guidelines* offers advice on how to improve the memory footprint of your program.
- *Code Speed Performance Guidelines* offers advice on how to tune your algorithms and find performance bottlenecks.
- *Drawing Performance Guidelines* offers advice on how to optimize your program's drawing-related code.
- *File-System Performance Guidelines* offers advice on how to access files more efficiently.
- *Launch Time Performance Guidelines* offers advice on how to speed up the launch time of your application.
- *Memory Usage Performance Guidelines* offers advice on how to use memory more efficiently and on how to analyze your current memory usage.
- *Concurrency Programming Guide* provides detailed information and examples about how to execute tasks in parallel.
- *64-Bit Transition Guide* discusses the performance impacts of 64-bit binaries and provides guidance on when creating such binaries is appropriate.

Developing for Performance

Performance is an aspect of software design that is often overlooked until it becomes a serious problem. If you wait until the end of your development cycle to do performance tuning, it may be too late to achieve any significant improvements. Performance is something to include early in the design phase and continue improving all throughout the development cycle.

Of course, in order to design for performance, it helps to understand what performance is. The sections in this chapter provide background information about the factors that influence performance, how those factors manifest themselves in Mac OS X and iOS, and how you can approach the monitoring of those factors.

What Is Performance?

The term “performance” may mean different things to different people. So before embarking on a quest to improve the performance of your application, now is a good time to consider what this term means.

Many people equate performance with speed. Indeed, if a program performs a complex operation in one second, you might think the program has good performance. Taken by itself, though, speed can be a misleading measurement. In complex software systems, the speed of an operation is not a fixed value. If you perform the same operation several times under different conditions, the time it takes to complete that operation could vary widely. That is because the program is only one of many processes sharing resources on the local system, and the use (or abuse) of those resources affects all other processes.

The following sections explain performance in terms of two different concepts: efficient resource usage and perceived performance. Both of these concepts have an important impact on how you design and implement your application, and understanding how to use both can lead to better overall performance.

The Efficient Use of Resources

A computer shares a limited number of resources among all the running processes. At the lowest level, these resources break down to the following categories:

- CPU time
- Memory space
- Mass storage space

All of your data resides either in memory or on some sort of mass storage device and must be operated on by the CPU. An efficient application uses all of these resources carefully. The following sections provide more detail about each type of resource and its effects on your programs.

CPU Time

CPU time is doled out by the system so you must make the best possible use of what time you have. Because both Mac OS X and iOS implement symmetric multiprocessing, each thread on the system is assigned a slice of time (maximum of 10 milliseconds) in which to run. At the end of that time (or before in many cases) the system takes back control of the CPU and gives it to a different thread.

On a typical system with many active threads, if every thread used its full allotment of time, performance would be terrible. This leads to one of the most important goals for writing an application:

Goal:*If your program has nothing to do, it should not consume CPU time.*

The best way to accomplish this goal is to use an event-based model. Using modern event-handling systems, such as the ones found in Cocoa and iOS, means your program's threads are run only when there is work to be done.

When your application does have work to do, it should use CPU time as effectively as it can. This means choosing algorithms that are appropriate for the amount of data you expect to handle. It also means using other system resources, such as an available vector unit (AltiVec or SSE in Mac OS X) or a graphics processor, to perform specialized operations, which leads to the following goal:

Goal:*Move work out of the CPU whenever you can.*

For basic information about how to use CPU time effectively, see [“Fundamental Optimization Tips”](#) (page 17). For tips specifically related to improving the speed of drawing operations, see [“Drawing Code”](#) (page 15).

Memory Space

Memory on modern computing hardware is typically composed of progressively slower (but larger) types of memory. The fastest memory available to the CPU is the CPU's own registers. The next fastest is the L1 cache, followed by the L2 and L3 caches when they are available. The next fastest memory is the main memory. The slowest memory of all consists of virtual memory pages in Mac OS X that reside on disk and must be paged in before they can be used.

In an ideal world, every application would be small enough to fit into the system's fastest cache memory. Unfortunately, most of an application's code and data resides either in main memory or on the disk. Therefore, it is important that the application's code and data is organized in a way that minimizes the time spent in these slower media, which leads to the following goal:

Goal:*Reduce the memory footprint of your program.*

Reducing the memory footprint of your program can significantly improve its performance. A small memory footprint usually has two advantages. First, the smaller your program, the fewer memory pages it occupies. Fewer memory pages, typically means less paging. Second, code is usually smaller as a result of being more heavily optimized and better organized. Thus, fewer instructions are needed to perform a given task and all of the code for that task is gathered on the same set of memory pages.

In addition to reducing your application's memory footprint, you should also try to reduce the footprint of writable memory pages in your application. Writable memory pages store global or allocated data for your application. In Mac OS X, such pages can be written to disk if needed, but doing so is slow. In iOS, the contents of these pages must be purged manually by the application itself, which might later require the application to recreate the data that was on those pages. In both cases, the system's efforts to free up memory take time—time that could be better spent executing your application code.

For basic information about how to reduce the footprint of your program, see [“Application Footprint”](#) (page 16). For tips specifically related to using memory more efficiently, see [“Memory Allocation Code”](#) (page 17).

Mass Storage Space

File-system performance on any computer is important because nearly everything resides in a file somewhere. Your applications, data, and even the operating system itself all reside in files that must be loaded into memory from a device that is incredibly slow compared to other parts of the system. File systems, whether they are local or network-based, are one of the biggest bottlenecks to performance. This leads to yet another goal:

Goal:*Eliminate unnecessary file operations and delay others until the information is actually needed.*

Removing this bottleneck, by eliminating or delaying your file operations, is important to improving the overall performance of your application. Tens of millions of CPU cycles can pass between the time you request data from a file and the time your program actually sees that data. If your program accesses a large number of files, it may wait many seconds before it receives all of the requested data.

Another important thing to remember is that your application and any files it creates may be on the network instead of on a local hard disk. Mac OS X, in particular, makes the network as invisible as possible, so you should never make assumptions about the locality of files.

For basic information about how to improve the file-based performance of your program, see [“File Access Code”](#) (page 16).

The Perception of Speed

Even if you tune your application code for optimal performance, it's entirely possible for your application to appear slow to the user. The problem is unavoidable: if you have a lot of work to do, you need the CPU time and resources to do that work. There are things that you can do though to give your application the appearance of speed, which leads to the following goal:

Goal:*Make your program responsive to the user.*

Responsiveness is usually a more important factor to users than raw speed. As long as a program responds to commands in a timely manner, the user is often willing to accept the fact that some tasks take longer to perform. The perception of speed is achieved by letting the user continue to work while your program processes data in the background. Improving the number of concurrent tasks performed by your application is a good way to make it responsive to the user. Concurrency is typically implemented using Grand Central Dispatch or threads. While your application's main thread responds to the user, dispatch queues or background threads perform calculations or handle other time-consuming tasks.

Another common way to make your application appear fast is to improve its launch time. An application that takes more than a second or two to launch is probably doing too much. Not only is it unresponsive to the user during that time but it may also be loading resources that are not needed right away or might not be used at all, which is wasteful.

For information about how to improve launch times, see [“Launch Time Initialization Code”](#) (page 16). For information about improving the perceived performance of your program, see [“Take Advantage of Perceived Performance”](#) (page 18).

Tracking Performance

The only way to ensure high performance is to include performance goals in your product design and measure your product against those goals throughout the development process. High performance is not a feature that you can graft onto your code at the end of the development cycle; it is intimately tied to that cycle. As code is written, it is important to know the impact it has on your program's overall performance. If you detect performance problems early, you have a good chance to fix them before it is too late.

The way to determine if you are meeting or exceeding a specific goal is to gather metrics. Apple provides several tools for monitoring and analyzing the performance of a program. You can also build measurement tools directly into your code to help automate the process of gathering data. Whichever approach you choose, you need to exercise those tools regularly and analyze the results.

Establish Your Baseline Metrics

The first thing you need to do is decide on a set of baseline metrics you want to measure. Choose the tasks you think are most important to your users and identify a set of constraints for performing those tasks. For example, you might want your application to load and display its initial window in less than 1 second, or you might want to keep your total memory usage within a given target range.

The tasks you choose to measure should reflect the needs of your users. Your marketing department should be able to help you choose a set of tasks that users find relevant. If you have an established product, talk to your users and find out what features they consider slow and consider improving the performance of those features as part of your planned update.

Once you have a list of tasks you want to track, you need to determine the performance targets for each task. For existing products, you might simply be trying to improve on the performance of the previous version. You might also try to measure the performance of competing products and set goals that meet or exceed their performance. If you have a new product, you might have to experiment with numbers to find reasonable values. Alternatively, you might want to establish aggressive baseline values and try to come as close to them as possible.

As with any performance measurement, consistency is important. Your process for establishing baseline metrics should include information about the system on which you are gathering those metrics. Record the hardware and software configuration of your system in some detail and always run your tests against the same configuration. Try to use the slowest possible hardware configuration for establishing your baselines. Measurements on a fast machine might lead you to believe that your software performs well, but many users will be running computers with slower processors and less memory.

Measure Early, Measure Often

Performance data is not something you can gather once and hope to find all of the performance bottlenecks in your program. It's easier to find problems if you maintain a history of your program's performance. Maintaining a history makes it easy to see whether your application's performance is improving or declining. If it's declining, you can take action to correct the problem before your product ships.

Another reason for measuring performance regularly is that you can correlate those results with code checkins. If performance at a particular milestone declines, you can review the code checked in during that period and try to find out why. Similarly, if performance improves, you can use the recent code checkins as a model for good programming practices and encourage your team to use similar techniques.

You should start making performance measurements as soon as you have a partially functional program. As new features are added, you can add measurements for those features. Incorporating a set of automated diagnostic routines directly into your program makes it easier for the members of your team to see the results immediately. Having this information readily available makes it easier for them to fix performance problems before checking in their code.

Analyze Your Results

Gathering data is the most important step in identifying performance bottlenecks. But once you have the data, it's also important that you use it to find problems. Analyzing performance data is not as simple as looking at the output and seeing the problem right away. You might get lucky and see the problem quickly, but some problems are subtler and require more careful analysis.

One way to help analyze results is to plot them graphically. Visualizing performance data can help you see trends much more quickly than if they are in a spreadsheet or other text-based medium. For example, you could plot the time to complete a single operation against a particular build to determine if performance is improving or declining from build to build. Plotting multiple data sets against the same set of milestones might also reveal trends and provide insight as to why performance is increasing or decreasing.

Analyze Higher-Level Algorithms

As you analyze performance data, keep an open mind towards the abstraction level at which the problem resides. Suppose the data you have indicates that a lot of time is spent inside a particular function. It may be that the code in the function itself can be optimized so that it performs faster, but is that the real cause of the problem? Run your program again but this time check the number of times that function is called. If the function is called one million times, the problem might be in the higher-level algorithm that is calling it in the first place. If the function is called once, the body of the function is likely the problem.

Note: Instruments and Shark are both powerful tools for analyzing your program's runtime behavior. Instruments lets you record multiple metrics and display them side by side, making it much easier to spot trends. The data mining capabilities of Shark are another excellent way to identify problems in higher-level algorithms quickly. For more information about Instruments, Shark and the other Apple-provided tools, see ["Performance Tools"](#) (page 21).

The performance tools themselves have limitations that you need to understand and take into account when analyzing data. For example, sampling programs may point out places where your application is spending a lot of time, but you should understand how those tools gather their data before drawing too many conclusions. Sampling tools do not track every function call. Instead, they offer a statistical analysis of your program based on samples taken at fixed intervals. Use the output from these tools as a guide, but be sure to correlate it with other data you record.

Other Analysis Techniques

If you are ever in doubt as to the true cause of a performance problem, avoid making assumptions about the cause of the problem. Instead, refine your analysis by focusing your data gathering efforts on the relevant code. Try using different tools to gather new types of information. A different tool might provide a unique perspective that reveals more about the actual problem.

Some additional ways you can analyze your program include the following:

- Watch the code in the debugger. Walking through code in the debugger might reveal logic errors that are slowing the code down.
- Add checkpoints to the code to log information about when that code was executed. For an example of using checkpoints to track initialization code, see the *Launch Time Performance Guidelines*.
- Try coding alternate solutions to the problem and see if they run into similar problems.

Basic Performance Tips

This chapter offers practical advice for how to tune your programs. It offers suggestions of areas you should monitor with the performance tools and also provides a list of practical tips for improving performance.

Common Areas to Monitor

Many performance problems can be traced to specific parts of your program. As you design and implement your code, you should monitor those areas to make sure they meet the performance targets you set.

Code for Your Program's Key Tasks

As you design your program, consider the tasks or workflows that users will encounter the most. During your implementation phase, be sure to monitor the code for those tasks and make sure their performance does not drop below acceptable levels. If it does, you should take immediate actions to correct the problems.

The key tasks performed by a program varies from program to program. For example, a word processor might need to be fast during text input and display, while a file utility program would need to be fast at scanning files and directories on a hard disk. It is up to you to decide which tasks your users are most likely to perform.

For information on how to identify and fix slow operations in your program, see *Code Speed Performance Guidelines*.

Drawing Code

Most programs do some amount of drawing. If your program uses only standard windows and controls, then you probably do not need to worry too much about drawing performance. However, if you do any custom drawing, you need to monitor your drawing code and make sure it is performing at acceptable levels. In particular, if you support any of the following, you should investigate ways to optimize your drawing code.

- Live resizing
- Custom view drawing code, especially if portions of the view can be updated without updating the whole view
- Textured graphics
- Entirely opaque views

For information on how to optimize drawing performance, see *Drawing Performance Guidelines*.

Launch Time Initialization Code

Launch time is where you initialize your program's data structures and prepare your application to receive user input. However, many programs do much more work at launch time than is necessary. In many cases, tasks performed at launch time can be deferred until after the application has posted its user interface and started processing events. This deferral gives the user the perception that your application is fast, which is a good first impression to make.

For applications that need to run in Mac OS X version 10.3.3 and earlier, another way to improve launch times is to prebind your application. Prebinding involves precalculating library address ranges and storing those values in your application binary. This step eliminates the need for the dynamic loader (`dyld`) to calculate those address ranges at launch time. Improvements in `dyld` for Mac OS X version 10.3.4 make prebinding unnecessary in that and later releases. Similarly, prebinding is unnecessary in iOS.

For information on how to improve launch-time performance, see *Launch Time Performance Guidelines*.

File Access Code

The file system is a bottleneck for getting information into memory and the CPU. In the time it takes to access a file, tens of millions of instructions may be executed. It is therefore imperative that you examine the way your program uses files and to make sure those files are actually needed and used properly.

Minimizing the number of files you use is one way to improve file-related performance. When you do access files, keep the following in mind:

- Understand how the system caches file data and know how to optimize the use of those caches. Avoid caching data yourself unless you plan to refer to it more than once.
- Read and write data sequentially whenever possible. Jumping around a file takes extra time to seek to the new location.
- Read larger blocks of data from files whenever possible, keeping in mind that reading too much data at once might cause different problems. For example, reading the entire contents of a 32 MB file might trigger paging of those contents before the operation is complete.
- Avoid closing and reopening files unnecessarily. If caching is enabled, doing so may cause the cache to be refreshed even if the data did not change.

For information on how to identify and fix file-related performance problems, see *File-System Performance Guidelines*.

Application Footprint

The size of your code can have a tremendous effect on system performance. The more memory pages used by your program, the fewer there are available for the system and other programs. This memory pressure can eventually lead to paging and an overall system slowdown.

Managing your code footprint is all about organizing your code and data structures. You need to make sure you have the right pieces in memory and that you are not causing any memory pages to be read or written unnecessarily. Some of the problems that cause a large memory footprint are as follows:

- Code pages contain unused code. The compiler typically organizes code by compilation module, which is not always the best technique. Organizing your modules based on what code executes together might be a better option.
- Static or constant data is stored on writable pages. During paging, this data is written to disk unnecessarily. Do what you can to move this data to non-writable pages that can be purged quickly.
- A program exports more symbols than are actually needed. Symbols take up space and are only needed by external code modules that must call into your program. Remove any symbols that should not be used externally.
- Code is not properly optimized by the compiler and linker. Be sure to try the compiler's optimization settings and see which one works best for your program.
- Too many frameworks are included by the program. Load only the frameworks your program actually uses.

For information on how to find and fix code footprint problems, see *Code Size Performance Guidelines*.

Memory Allocation Code

Programs allocate memory for storing both permanent and temporary data structures. Each memory allocation has a cost associated with it, both in CPU time and in memory consumption. Understanding when your program allocates memory and how that memory is used can help you reduce both of those costs.

Understanding your program's memory usage can help determine ways to reduce that usage. You can use the available performance tools to find out if autoreleased Objective-C objects are being deallocated before they cause too much paging. You can also use these tools to find memory leaks caused by bugs in your code. Watching the number of times you call `malloc` might also point out places where you can reuse existing memory blocks rather than create new ones.

One important rule to follow when allocating memory is to be lazy. Defer memory allocations until you actually need the memory being used. For some additional ways you can be lazy with memory allocations, see "Be Lazy" (page 17).

For information about optimizing your memory allocation patterns, see *Memory Usage Performance Guidelines*.

Fundamental Optimization Tips

Before you begin implementing a new program, there are several performance enhancements you should consider adding. Although you might not be able to take advantage of all of these enhancements in every case, you should at least consider them during your design phase.

Be Lazy

A very simple way to improve performance is to make sure your application does not perform any unnecessary work. Each moment of an application's time should be spent responding to the user's current request, not predicting future requests. If you do not need a resource right away, such as a nib file containing a preferences window, don't load it. Such an action takes time to execute because it accesses the file system, and if the user never opens that preference window, the process of loading its nib file is a waste of time.

The basic rule to follow is to wait until the user requests something from your application, then use the necessary resources to fulfill the request. You should cache data only in situations where there is a measurable performance benefit. Preloading caches on the assumption that the rest of the application will run faster can actually degrade performance in low-memory situations. In such a situation, your cached data might be paged to disk before it can be used. Thus, any savings you gained by caching the data turns into a loss because that data must now be read from disk twice before it is ever used. If you really want to cache data, do so after the operation has been performed once.

Some other things to be lazy about include the following:

- Defer memory allocations until you actually need the memory.
- Don't zero-initialize blocks of memory. Call the `calloc` function to do it for you lazily.
- Give the system the chance to load your code lazily. Profile and organize your code so that the system loads only the code needed for the current operation.
- Defer reading the contents of a file until you actually need the information.

Take Advantage of Perceived Performance

The perception of performance is just as effective as actual performance in many cases. Many program tasks can be performed in the background, using a dispatch queue, or at idle time. Doing this keeps the application's main thread free to handle user interactions and makes the program interface feel more responsive to the user. As you design your program, think about which tasks can be moved to the background effectively. For example, if your program needs to scan a number of files or perform lengthy calculations, do so using a dispatch queue.

Another way to improve perceived performance is to make sure your application launches quickly. At launch time, defer any tasks that do not contribute to the immediate presentation of your application interface. For example, defer the creation of large data structures until after your application has finished launching and displayed its main window. If you use your main window to display some data that must be calculated or retrieved at launch time, show the window first along with a progress indicator or other status message indicating that the data is being loaded. For applications that use plug-ins, avoid loading plug-ins until their code is actually needed.

Use Event-Based Handlers

All modern Mac OS X applications should be using the Cocoa event system or Carbon Event Manager. (Similarly, iPhone applications must use the touch-based event system provided by the UIKit framework.) Your application should never retrieve events by polling the system. Doing so is highly inefficient. In fact, when there are no events to process, polling code is a 100 percent waste of CPU time. The modern event-based APIs are designed to provide the following benefits:

- They make your program more responsive to the user.
- They reduce your application's CPU usage.
- They minimize your application's working set—the number of code pages loaded in memory at any given time.
- They allow the system to manage power aggressively.

Besides user events, you should avoid polling in other situations as well. Threads in Mac OS X and iOS use run loops to provide on-demand responses to timers, network events, and other incoming data. Many frameworks also use an asynchronous programming model for certain tasks, notifying a designated handler function or method when the task completes. In Mac OS X 10.6 and later, dispatch sources also provide a way for you to receive important events asynchronously and execute them on a dispatch queue.

Improve the Concurrency of Your Program's Tasks

On computers with multiple cores, concurrency is another way to improve both the perceived and actual performance of your program. A program that is able to perform multiple tasks at the same time can execute those tasks in parallel on a multicore computer. Even when a computer has only one core, factoring your code into multiple, asynchronous tasks can provide a perceived speed boost when done correctly. Specifically, you should perform custom tasks using a dispatch queue and leave your main thread free to handle user events and update your user interface primarily.

Before you begin adding support for concurrency, though, be sure to put some thought into how your program might implement the corresponding tasks effectively. Factoring your code into different tasks requires some consideration of your program data structures and code paths. Tasks that share data structures may require the use of a serial dispatch queue to synchronize access to those structures.

For information on how to implement concurrency in your program, see *Concurrency Programming Guide*.

Use the Accelerate Framework

If your Mac OS X application performs a lot of mathematical computations on scalar data, you should consider using the Accelerate framework (`Accelerate.framework`) to accelerate those calculations. The Accelerate framework takes advantage of any available vector processing units (such as the PowerPC AltiVec extensions, also known as Velocity Engine, or the Intel x86 SSE extensions) to perform multiple calculations in parallel. By coding to the framework, instead of to the vector unit, you can avoid having to create separate code paths for each platform architecture. The Accelerate framework is highly tuned for all of the architectures Mac OS X supports.

Note: The Accelerate framework is not available in iOS.

Tools such as Instruments and Shark can help point out portions of your program that might benefit from using the Accelerate framework. For more information about using these and other tools, see [“Performance Tools”](#) (page 21).

Modernize Your Application

If your program was designed to run on an older version of Mac OS, you should update your code to support the conventions of Mac OS X. In particular, you should avoid using older technologies such as Carbon or legacy technologies such as QuickDraw. Instead, you should build your applications using Cocoa or Cocoa Touch. You should also update your binary format to Mach-O. The Mach-O format is the only format supported on Intel-based Macintosh computers and iOS-based devices.

For a list of the technologies available for use in Mac OS X applications, see *Mac OS X Technology Overview*. For a list of technologies available for use in iOS, see *iOS Technology Overview*.

Performance Tools

The Xcode tools includes several graphical applications and command-line tools for gathering performance metrics. The following sections introduce the available tools and tell you when you might use them.

Installing the Apple Tools

The Apple performance tools are installed as part of the Xcode tools. This package ships on a CD that comes with retail copies of Mac OS X. You can also download the Xcode tools for free from the [Apple Developer website](#).

To install Xcode Tools, double-click the installer package found on the Xcode Tools CD or that you downloaded from the web. By default, the installer creates a `/Developer` directory on the boot volume of your hard drive and installs applications, documentation, examples, and other files in this directory. In Xcode 3.0 and later, the installer lets you specify a custom installation directory so that you can have multiple versions of the tools on your computer.

Most of the applications with a graphical user interface have online help available through the Help menu. For information about these tools, see *Mac OS X Man Pages*.

Key Applications

Although Xcode comes with numerous applications for gathering performance data, there are a few that you will use more frequently than others. These tools can be used to gather data for both Mac OS X applications and iPhone applications. The following sections describe these tools and the features they provide.

Instruments

Introduced in Mac OS X v10.5, the Instruments application combines a set of powerful analysis tools with an easy-to-use graphical user interface. Instruments provides unprecedented information about the runtime behavior of your application. Rather than show only one aspect of your program at a time, you configure each analysis session with one or more “instruments,” each of which gathers information about a particular performance metric. The data from all instruments is shown side-by-side, making it easier to correlate the data from one instrument to another and detect trends in your application’s behavior.

Among the types of metrics you can gather with the Instruments application are the following:

- The performance of Core Data–based applications
- Information about file system reads, writes, and other operations
- Statistics relating to garbage-collected code

- Information about graphics operations and performance
- Statistics about object and other memory-related allocations
- Information about memory leaks
- Statistical samples of your application at runtime
- Information about process-specific and system-level activity
- Information about Java thread activity
- Information about events dispatched by Cocoa

For a quick example of how to use the Instruments application, see [“Using Instruments”](#) (page 30). For detailed information about the Instruments application, see *Instruments User Guide*.

Shark

Shark is an advanced tool for sampling or tracing a single application or all running applications. Like Instruments, Shark provides a set of powerful analysis tools with an easy-to-use interface to help you gather performance data. Shark helps you isolate problems quickly by providing a rich set of data-mining features and is an indispensable tool for finding performance bottlenecks. It provides optimization tips and help with analyzing assembly-level operations. Because it traces operations down at the kernel-level of the system, Shark can also provide you with tremendous insight about the behavior of your application relative to the rest of the system.

Among the types of operations you can perform in Shark are the following:

- Statistical sampling of your application over a period of time
- System-level tracing
- Malloc tracing
- Static analysis
- L2 Cache profiling
- Processor bandwidth analysis
- Java code analysis

In addition to basic sampling, Shark also offers the windowed time facility feature for several of its sampling options. The windowed time facility tells Shark to record and process a finite buffer of the most recently acquired samples. This feature lets Shark record data continuously and for long periods of time in the background. You can then direct Shark to stop sampling only after something interesting occurs in your code and to display the samples related to that interesting behavior. Thus, you can use this feature to determine what part of your program’s execution is interesting after it occurs.

For a quick example of how to use Shark, see [“Using Shark”](#) (page 33). For detailed information about Shark, see *Shark User Guide*.

Analysis Tools

Instruments and Shark are not the only analysis tools that you can use to gather data about the performance of your program. Most of the analysis tools are geared towards finding specific types of performance problems.

Table 3-1 lists the analysis tools installed with the Xcode Tools. (These tools are not able to analyze applications running in iOS.) Applications such as MallocDebug are installed in the `<Xcode>/Applications/Performance Tools` directory, where `<Xcode>` is the root directory of your Xcode installation. Command-line tools, such as `heap` and `leaks`, are installed in `/usr/bin`.

Table 3-1 Analysis tools

Tool	Description
MallocDebug	Tracks and analyzes memory allocated in an application. You can use this tool to find memory leaks or analyze memory allocation patterns.
OpenGL Driver Monitor	Gathers GPU-related performance data, including data related to VRAM usage, video bus traffic, and hardware stalls among others. You can use this information to identify the cause of temporary slowdowns or sporadic hesitations in your OpenGL application.
OpenGL Profiler	Creates a runtime profile of your OpenGL-based application. You can view function statistics and the call-trace history of your application's OpenGL calls.
Saturn	Instruments your code to provide function-level profiling and displays the resulting data graphically. You can use this tool to count events, such as how many times a function is called or an event is sent.
heap	Lists all <code>malloc</code> -allocated buffers in the heap of a specified process
leaks	Searches the memory space of a process for any allocated but unreferenced blocks of memory.
vmmmap	Displays the virtual memory regions allocated to a specified process. You can use this tool to analyze the memory usage of your process.

Monitoring Tools

Monitoring tools are passive tools that gather data automatically. To use these tools, leave them running while you exercise the features of your program. You can then analyze the data generated by these tools to gain a better understanding of your program's performance characteristics. Some programs, like Spin Control, should be left running all the time. Most others can be launched and terminated as needed to gather performance information.

Table 3-2 lists the monitoring tools installed with the Xcode Tools in Mac OS X. (These tools are not able to monitor applications running in iOS.) Applications such as BigTop and Spin Control are installed in the `<Xcode>/Applications/Performance Tools` directory hierarchy, where `<Xcode>` is the root directory of your Xcode installation. The Activity Monitor tool is installed in the `/Applications/Utilities` directory. Command-line tools, such as `fs_usage` and `top`, are installed in `/usr/bin`.

Table 3-2 Monitoring tools

Tool	Description
Activity Monitor	Displays common usage statistics relating to memory and CPU usage for the currently running processes. You can also initiate the sampling of a process from this application. This tool provides information that is similar to that of the <code>top</code> tool.
BigTop	Displays system statistics, such as CPU, disk, network and memory usage graphically over time. You can monitor a single process or all processes. This tool provides information that is similar to that of the <code>top</code> and <code>vm_stat</code> tools.
Quartz Debug	Shows screen updates in real time by briefly flashing the areas being redrawn. You can use this tool to analyze your application's drawing behavior.
Spin Control	Samples programs that cause the spinning cursor to appear. Leave this program running in the background to catch unresponsive applications at critical moments.
Thread Viewer	Graphically displays activity across a range of threads. It provides color-coded timeline views of thread activity and can display backtraces of activity at specific points in time. (Available in Mac OS X v10.5 and earlier.)
<code>fs_usage</code>	Displays an ongoing list of file-system activity, as generated by page faults and calls to file-system functions. You can use this tool to understand the file access patterns of your program.
<code>sc_usage</code>	Displays an ongoing list of system call and page fault statistics.
<code>top</code>	Displays common system usage statistics relating to memory and CPU usage for the currently running processes. This tool updates the information dynamically so that you can see trends at runtime.

Hardware Analysis Tools

The CHUD Tools include additional applications for doing hardware and low-level software analysis in Mac OS X. (These tools are not able to analyze hardware running iOS.) Table 3-3 lists the tools that are part of this package and provides a brief overview of their purpose. The graphical applications are installed in `<Xcode>/Applications/Performance Tools/CHUD` directory, where `<Xcode>` is the root directory of your Xcode installation. The command-line tools are installed in the `/usr/bin` directory. All of the tools are included with the Xcode Tools.

Table 3-3 CHUD tools

Name	Description
Reggie SE	Lets you examine and modify CPU and PCI configuration registers.
<code>acid</code>	A command-line tool that analyzes TT6E instruction traces and presents detailed analyses and histograms. You can use this tool to detect bad instruction sequences, such as misaligned operands, data dependency stalls, and spilled loads.

Name	Description
<code>simg4</code>	A command-line tool that is a cycle-accurate simulator of the Motorola 7400 processor. This tool takes TT6 traces as input. (Available in Mac OS X v10.5 and earlier.)
<code>simg5</code>	A command-line tool that is a cycle-accurate simulator of the IBM 970 processor. This tool takes TT6 traces as input. (Available in Mac OS X v10.5 and earlier.)

For information about using the CHUD tools, see online help for a particular tool. Documentation for some CHUD tools is also installed in the `<Xcode>/Documentation/CHUD` directory, where `<Xcode>` is the root directory of your Xcode installation.

Additional Command-Line Tools

Table 3-4 lists some additional command-line tools that you can use to monitor and analyze performance in Mac OS X. (These tools are not able to monitor and analyze applications running in iOS.) These tools are located in the `/usr/bin/` directory and must be run from a command-line prompt. Most are installed along with the Xcode Tools. For information about these tools, see *Mac OS X Man Pages*.

Table 3-4 Command-line tools

Name	Description
<code>atos</code>	Converts back and forth between a symbol name and the numeric address of that symbol in a running executable.
<code>c2ph</code>	Displays the C-structures from an object file along with their member offset values.
<code>gprof</code>	Produces execution profiles based on an execution analysis of a program.
<code>malloc_history</code>	Shows the <code>malloc</code> allocations performed by a specified process.
<code>nm</code>	Displays the symbol table information for one or more object files.
<code>otool</code>	Displays the contents of a Mach-O executable in a more human-readable form
<code>pagestuff</code>	Displays information about the logical pages of a Mach-O executable file.
<code>pstruct</code>	Parses the C structures from an object file and displays them along with their member offset values.
<code>sample</code>	Produces an execution profile based on the execution analysis of a program.
<code>vm_stat</code>	Displays Mach virtual memory statistics, including the number of active, inactive, wired, and free pages. This tool also displays page fault and other activity information.

Doing an Initial Performance Evaluation

So, you have some code and you want to see if it is suffering from performance problems. Where do you start? Not all problems are immediately visible. You might notice that an operation took several seconds to perform, but you might not notice an operation that consumed too many CPU cycles or allocated too much memory. This is where Apple's performance tools come into play. They can help you see aspects of your program that are easily overlooked.

The following sections provide a brief overview of how to use some key tools when starting to analyze your program. These tools are good for identifying potential problems and can provide a significant amount of performance data. Remember, though, that there may be other tools that provide more specific information related to the problem. Running your application with several other tools can help you confirm whether a particular area is a problem.

Important: The performance tools are there to assist you in investigating performance problems. Make sure you gather as much data as you can during your analysis. Performance analysis is somewhat of an art and requires careful consideration of all available data to find the real problem.

For more information about the performance tools in general, including where to get them, see [“Performance Tools”](#) (page 21).

Using `top`

The `top` tool is an important tool for identifying potential problem areas in a process. This tool displays a periodically sampled set of statistics on system usage. Using `top` and understanding its output are an excellent way to identify potential performance problems.

The `top` tool displays periodically updated statistics on CPU usage, memory usage (in various categories), resource usage (such as threads and ports), and paging events. In the default mode, `top` displays CPU and memory utilization of all system processes. You can use this information to see how much memory your program is using and what percentage of the CPU time it is using. An idle program should not use any CPU time and an active one should consume a proportionate amount of CPU time based on the complexity of the task.

Note: If you want to track CPU usage and other statistics over time, use BigTop instead. BigTop graphs performance trends over time, providing a real-time display of memory usage, page faults, CPU usage, and other data.

Listing 4-1 shows a typical statistical output from `top`. For application developers, the statistics you should be most interested in are the CPU usage, resident private memory usage (`RPRVT`), and pagein/pageout rates. These values tell you some key things about your application's resource usage. High CPU usage may mean that your application's tasks are not tuned appropriately. Increased memory usage and page-in/page-out rates may indicate a need to reduce your application's memory footprint.

Listing 4-1 Typical output of top

```

Processes: 36 total, 2 running, 34 sleeping... 81 threads
Load Avg: 0.24, 0.27, 0.23      CPU usage: 12.5% user, 87.5% sys, 0.0% idle
SharedLibs: num = 77, resident = 10.6M code, 1.1M data, 4.75M LinkEdit
MemRegions: num = 1207, resident = 16.4M + 4.94M private, 22.2M shared
PhysMem: 16.0M wired, 25.8M active, 48.9M inactive, 90.7M used, 37.2M free
VM: 476M + 39.8M 6494(6494) pageins, 0(0) pageouts

  PID COMMAND      %CPU  TIME      #TH #PRTS #MREGS RPRVT  RSHRD  RSIZE  VSIZE
  318 top           0.0%  0:00.36   1   23   13   172K   232K   380K   1.31M
  316 zsh           0.0%  0:00.08   1   18   12   168K   516K   628K   1.67M
  315 Terminal     0.0%  0:02.25   4  112   50   1.32M   3.55M   4.88M   31.7M
  314 CPU Monito   0.0%  0:02.08   1   63   35   896K   1.34M   2.14M   27.9M
  313 Clock        0.0%  0:01.51   1   57   38   1.02M   2.01M   2.69M   29.0M
  312 Dock         0.0%  0:03.72   2   77   78   2.18M   2.28M   3.64M   30.0M
  311 Finder       0.0%  0:07.68   4   86  171   7.96M   9.15M   15.1M   52.1M
  308 pbs          0.0%  0:01.37   4   76   40   928K   684K   1.77M   15.4M
  285 loginwindow  0.0%  0:07.19   2   70   58   1.64M   1.93M   3.45M   29.6M
  282 cron         0.0%  0:00.00   1   11   14    88K   228K   116K   1.50M
  245 sshd         0.0%  0:02.48   1   10   15   176K   312K   356K   1.41M
  222 SecuritySe   0.0%  0:00.14   2   21   24   476K   828K   1.29M   3.95M
  209 automount    0.0%  0:00.03   2   13   20   336K   748K   324K   4.36M
  200 nfsiod       0.0%  0:00.00   1   10   12    4K   224K   52K   1.22M
  199 nfsiod       0.0%  0:00.00   1   10   12    4K   224K   52K   1.2
[...]
```

In its header area, `top` displays statistics on the global state of the system. This information includes load averages; total process and thread counts; and total memory, broken down into various categories such as private, shared, wired, and free. It also includes global information concerning the system frameworks. At regular intervals, `top` updates these statistics to account for recent system activity.

Table 4-1 describes the columnar data that appears in the CPU and memory utilization mode using the `-w` parameter. For detailed information about how `top` reports information, see the `top` man page.

Table 4-1 Output from top using the `-w` option

Column	Description
PID	The BSD process ID.
COMMAND	The name of the executable or application package. (Note that Code Fragment Manager applications are named after the native process that launches them, <code>LaunchCFMApp</code> .)
%CPU	The percentage of CPU cycles consumed during the interval on behalf of this process (both kernel and user space).
TIME	The amount of CPU time (<i>minute:seconds.hundredths</i>) consumed by this process since it was launched.
#TH	The number of threads owned by this process.
#PRTS (delta)	The number of Mach port objects owned by this process. (To display the delta value relative to the value first displayed when <code>top</code> was launched, use the <code>-w</code> parameter.)
#MREG	The number of memory regions.

Column	Description
VPRVT	The private address space currently allocated. (This value is displayed only with the <code>-w</code> parameter.)
RPRVT (delta)	The total amount of resident private memory. (To display the delta value relative to the previous sample, use the <code>-w</code> parameter when running <code>top</code> .)
RSHRD (delta)	The resident shared memory. (To display the delta value relative to the previous sample, use the <code>-w</code> parameter when running <code>top</code> .)
RSIZE (delta)	The total resident memory as real pages that this process currently has associated with it. Some may be shared by other processes. (To display the delta value relative to the previous sample, use the <code>-w</code> parameter when running <code>top</code> .)
VSIZE (delta)	The total address space currently allocated, including shared memory. (To display the delta value relative to the previous sample, use the <code>-w</code> parameter when running <code>top</code> .) This value is mostly irrelevant for Mac OS X processes. Every application has a large virtual size because of the shared region used to hold framework and library code.

The RPRVT data (for resident private pages) is a good measure of how much real memory an application is using. The RSHRD column (for resident shared pages) shows the resident pages of all the shared mapped files or memory objects that are shared with other processes.

Note: The `top` tool does not provide a separate count of the number of pages in shared libraries that are mapped into the process.

The `top` tool reports memory usage of windows in the “shared memory” category because window buffers are shared with the window server.

Table 4-2 shows the columns displayed in the event-counting mode, which is enabled with either the `-e`, `-d`, or `-a` option on the command line. You can use these options to gain additional insight about specific behaviors of your application. For example, you can correlate the number of page faults with the amount of memory your application is using to determine if your application’s memory footprint might be too big.

Table 4-2 Output from `top` using the `-d` option

Column	Description
PID	The BSD process ID.
COMMAND	The name of the executable or application package. (Note that Code Fragment Manager applications are named after the native process that launches them, <code>LaunchCFMApp</code> .)
%CPU	The percentage of CPU cycles consumed during the interval on behalf of this process (both kernel and user space).
TIME	The amount of CPU time consumed by this process (<i>minute:seconds.hundredths</i>) since it was launched.
FAULTS	The total number of page faults.

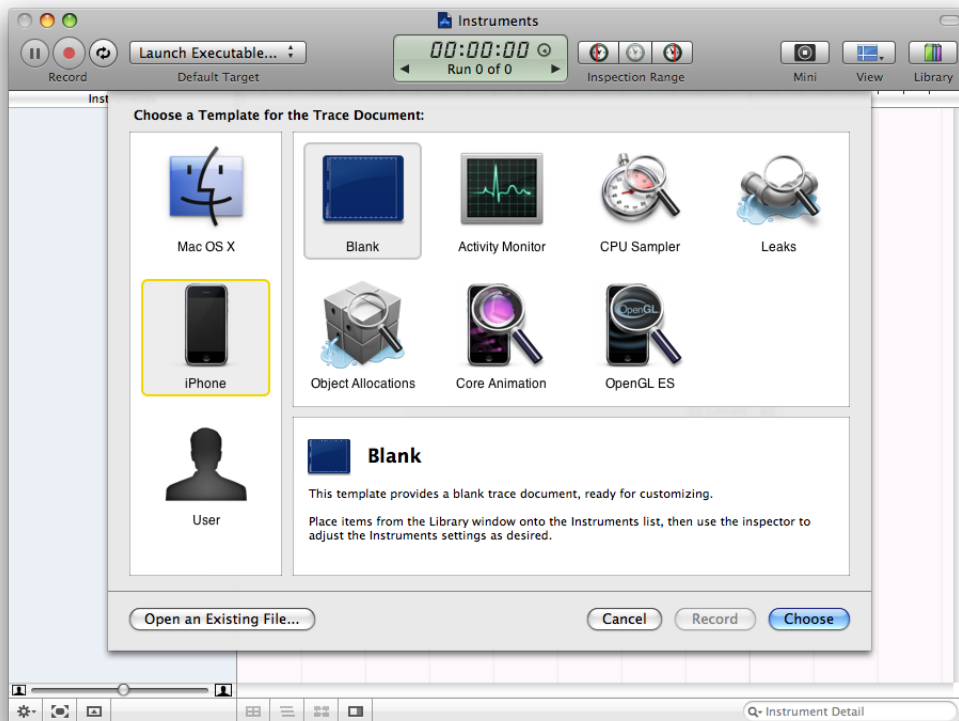
Column	Description
PAGEINS	The number of page-ins, requests for pages from a pager (each page-in represents a 4 kilobyte I/O operation).
COW_FAULTS	The number of faults that caused a page to be copied (generally caused by copy-on-write faults).
MSGSENT	The number of Mach messages sent by the process.
MSGSRCD	The number of Mach messages received by the process.
BSDSYSCALL	The number of BSD system calls made by the process.
MACHSYSCALL	The number of Mach system calls made by the process.
CSWITCH	The number of context switches to the process (the number of times the process has been given time to run by the kernel's scheduler).

Using Instruments

Instruments is an incredibly powerful tool that you can use to gather performance data and analyze your application's overall behavior. The Instruments application supports applications running in either Mac OS X or iOS. When in doubt, you should start your performance analysis using Instruments because it is simple to use and gives you a wide variety of options for gathering data. One of the more useful features of the Instruments application is the ability to see different types of performance information graphed side by side. Seeing information in this way makes it easier to identify trends and relationships between seemingly different metrics.

When you first launch Instruments, you are asked to select a starting template for your document (Figure 4-1). The templates come preconfigured with one or more instruments that are designed to gather data for specific situations. For example, the Leaks template includes both the ObjectAlloc instrument and the Leaks instrument, letting you see both the total number of memory blocks that were allocated and the subset of those memory blocks that are considered leaks. You can add more instruments to a document at any time but the common configurations provided by the templates are usually sufficient for typical tasks.

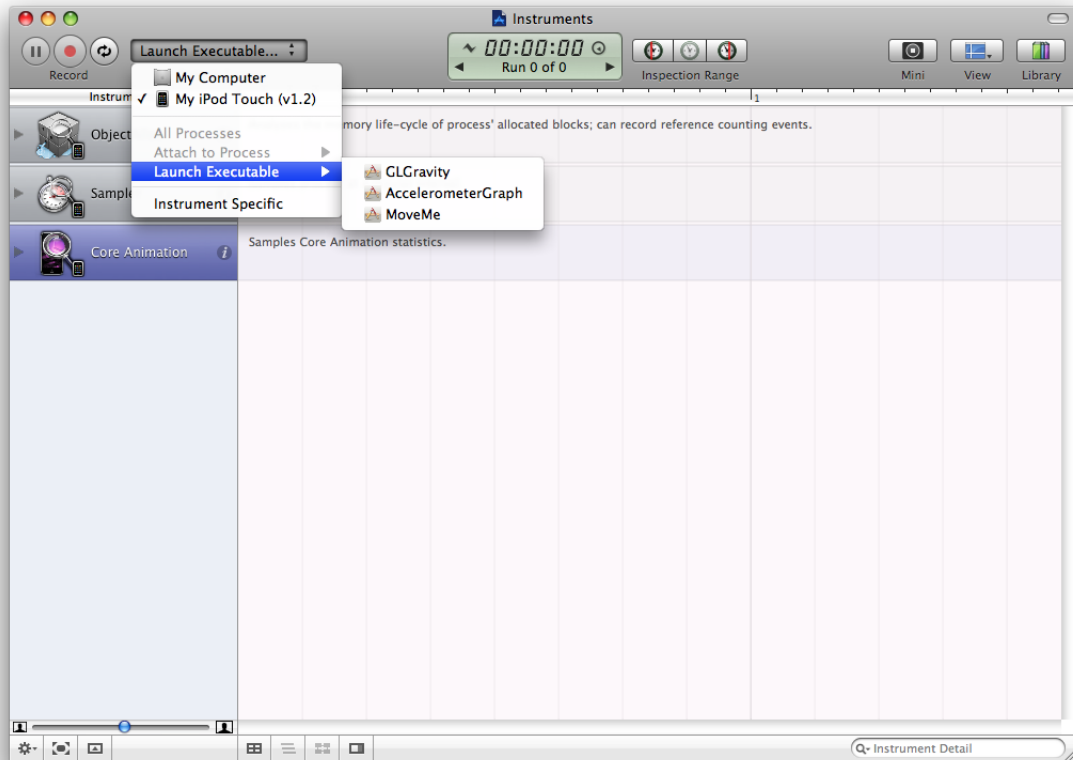
Figure 4-1 Choosing an Instruments template



When creating a document, you can also choose to gather data for an application running in either Mac OS X or iOS. Choosing a particular platform configures the document for that platform by default, but you can change the platform later if needed.

After you create your document, you must select which application (or applications) you want Instruments to monitor. You do this using the Default Target control in the document's toolbar. Figure 4-2 shows an instruments document being configured to launch and monitor an application on an iOS-based device. When tracing iPhone applications, you launch the executable as part of the overall tracing process. For Mac OS X-based applications, some instruments give you the option of attaching to an already running process.

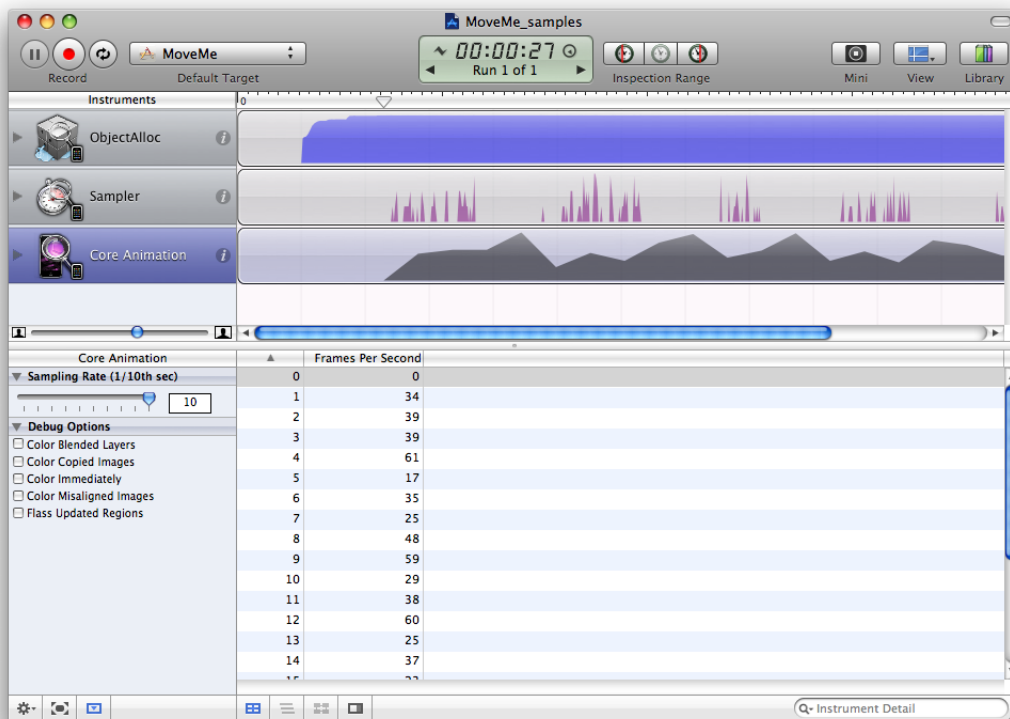
Figure 4-2 Configuring the target executable



After selecting the process you want to monitor, you click the Record button and Instruments starts recording data for the selected application. Data recording takes place in real time, so you can interact with your application and initiate any tasks you want to check for performance problems. As you manipulate your application, Instruments shows the resulting data in the trace document window. To stop recording, press the Record button again, which should now be titled Stop.

Figure 4-3 shows the data gathered for the MoveMe sample application after a recording run. The graphical view for most instruments is configurable so that you can look at the data that is most interesting to you. In the figure, the ObjectAlloc instrument shows the total amount of allocated memory, but you can also change the display to show the density of memory allocations at any given time. Below the graphical view, you can view the detailed data that was gathered by Instruments from the running application.

Figure 4-3 Examining the recorded data



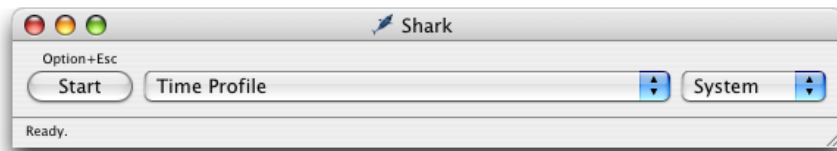
For detailed information about how to use Instruments and for information about the types of performance data you can gather, see *Instruments User Guide*.

Using Shark

Shark is one of the most powerful analysis tools you can use to find performance problems in your code. Shark can generate a time-based profile of your program's execution, trace function calls, and graph memory allocations over time. You can use Shark to track information for your program or for the entire system. Shark even captures execution information for kernel entities such as drivers and kernel extensions.

Despite the power of Shark, using the tool is very simple. Upon launch, Shark presents you with the window shown in Figure 4-4. Click Start (or use the Option-Escape global shortcut) to start sampling all system processes. Click the button again (or use the same global shortcut) to stop gathering data and display the samples (Figure 4-5).

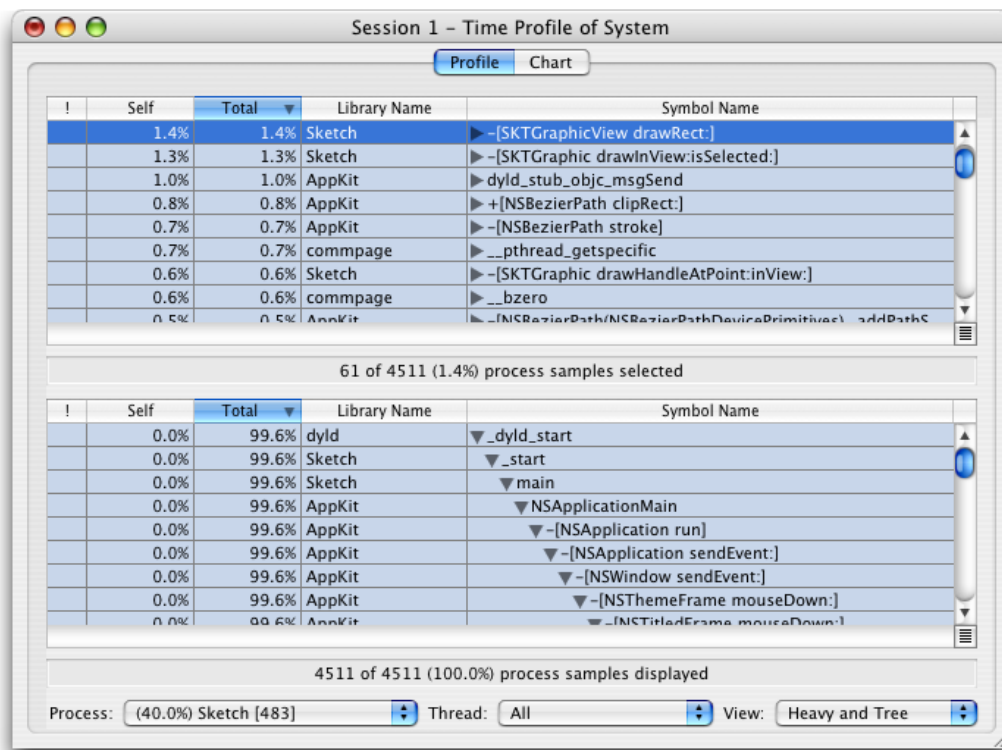
Figure 4-4 Shark launch window



If you want to limit sampling to only one process, you can use the pop-up menu on the right side of the window to select that process. Also, if you want to do something other than a statistical time profile, choose the appropriate option from the configuration pop-up menu before you click Start.

When you stop sampling, Shark displays the profile window (Figure 4-5) with the gathered data. This is the main window you use to identify potential problems. You can configure this window to display a heavy view, tree view, or both.

Figure 4-5 Shark profile window



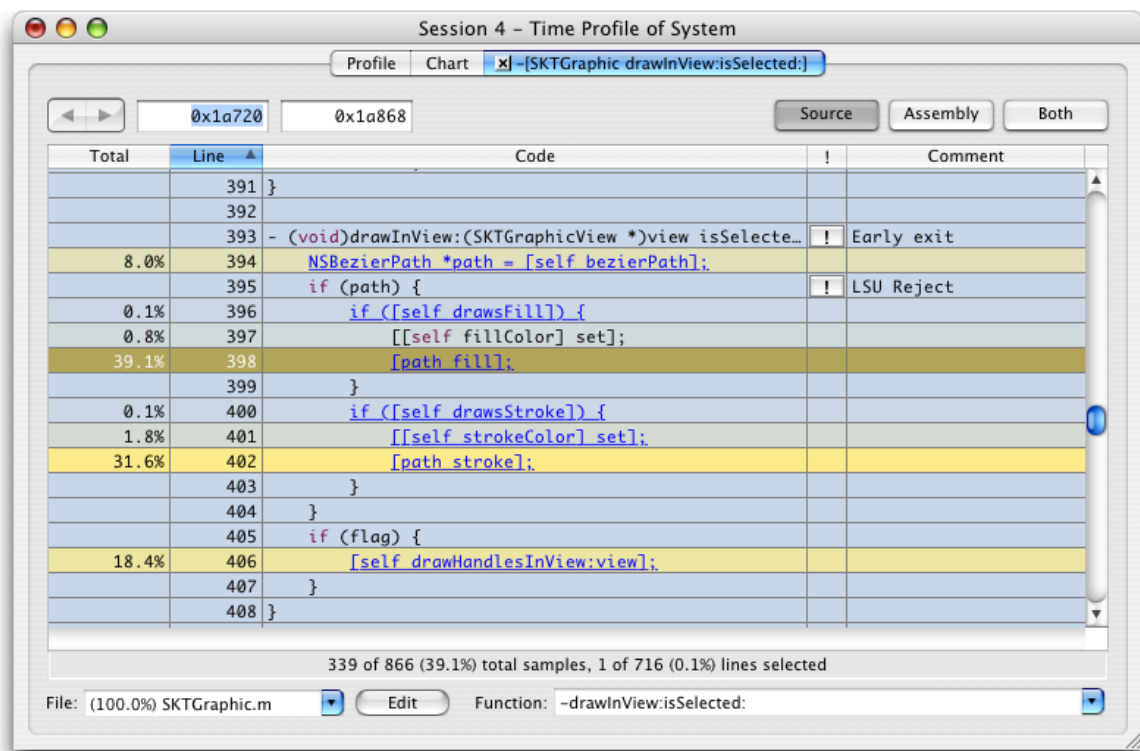
The heavy view sorts function calls based on the frequency by which they appeared. This view identifies your program's hot spots. If you see one of your program's functions near the top of this view, you should investigate it. Functions tend to have higher weights if they are poorly optimized but a more likely scenario is that the function is being called too frequently from some other place. This can indicate an inefficient higher-level algorithm.

The tree view shows the same data organized by calling hierarchy. This view offers a convenient way to understand the context in which a particular function is called. This is the more traditional way to view call stack data and can be used in conjunction with the heavy view to track down hot spots and to see the surrounding context quickly.

The Shark profile window makes it easy to filter out irrelevant code and libraries through a feature called data mining. The Data Mining menu has several options for excluding or flattening symbols and libraries. If you apply these commands, Shark rolls the costs of calling those symbols and libraries into the function that called them. For example, if you know your code makes several calls to Core Foundation and you exclude the Core Foundation library, the time spent in Core Foundation now appears as time spent in your code. If the amount of time spent in your code jumps significantly afterward, you might want to investigate ways to call fewer Core Foundation functions.

Shark also can help you identify performance issues within a given function through the code view. When you double-click a function, Shark displays the source code for that function if it is available (see Figure 4-6). It applies color to lines of code based on the amount of time spent in that code. For each line of source, brighter shades of yellow indicate where more time was spent during the profile.

Figure 4-6 Shark code display



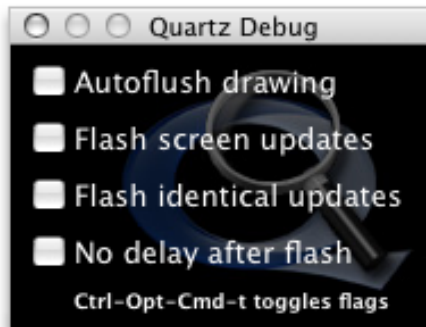
Shark occasionally offers specific tuning tips and comments in the margin. Clicking the exclamation-point icon displays a tip that you can use to improve your code. The comment column displays a summary of the tip.

For more information about Shark and its features, see *Shark User Guide*.

Using Quartz Debug

Quartz Debug is an important tool for determining the efficiency of your drawing code. The tool collects information from your program's drawing calls to find out where your program is drawing and whether it is redrawing content unnecessarily. Figure 4-7 shows the Quartz Debug options window.

Figure 4-7 Quartz Debug options



With the Flash screen updates option enabled, Quartz Debug shows you visually where your code is drawing. It places a yellow rectangle over an area where a redraw operation is about to occur and then pauses briefly before redrawing the content. This flickering yellow pattern can point out places where you are drawing more than is necessary. For example, if you update only a small portion of a custom view, you probably do not want to be forced to redraw the entire view. Alternatively, if you see a system control being redrawn several times in succession, it might point out the need to hide that control before changing its attributes.

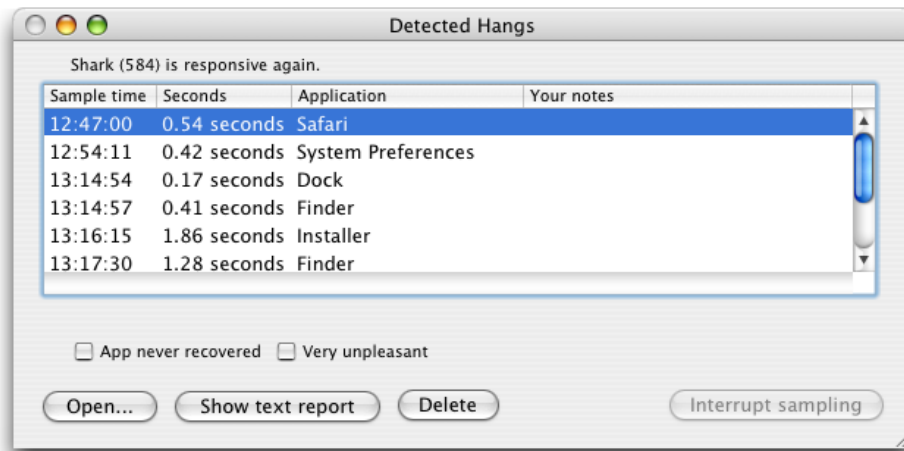
The “Flash identical updates” option displays red over any areas whose resulting bits are identical to the current content. You can use this option to detect redundant drawing operations in your code.

Using Spin Control

If an application becomes unresponsive at any time, the window server notifies the user of this situation by changing the cursor to a spinning multi-color wheel. If your application is the one that is unresponsive, sampling it during that time can help you determine why it is unresponsive. However, even if you have Instruments or another tool ready to go, you might not be able to start them fast enough to gather a set of samples during the unresponsive period. This is where Spin Control provides a helpful solution.

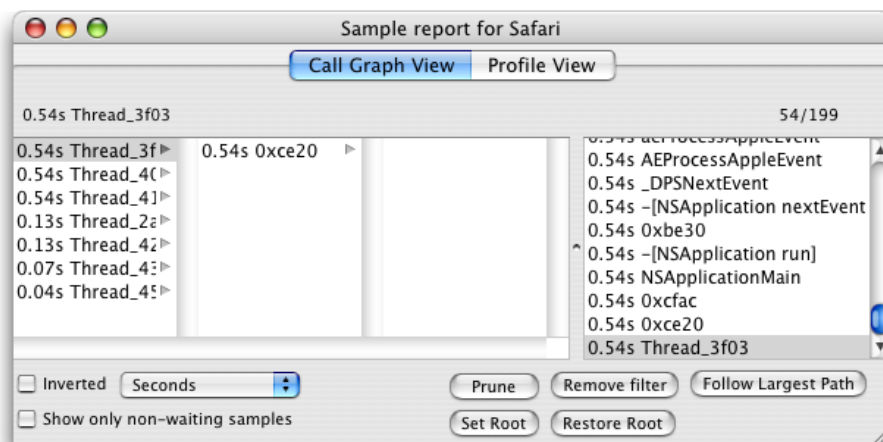
Spin Control is a monitoring tool that automatically samples while they are unresponsive. Simply launch Spin Control and leave it running on your computer whenever you are testing your application. When the spinning cursor appears, Spin Control automatically starts sampling your application, making the results available from its main window, as shown in Figure 4-8.

Figure 4-8 Spin Control main window



To view the backtrace for a particular session, select that session and click Open. Spin Control displays a browser window containing the sample data, as shown in Figure 4-9. You can navigate the call stacks for your threads and use the data to find out which function or methods were executing while your application was unresponsive. The controls in the bottom-left corner of the window let you invert the call stack information and change the way you view the samples. The buttons at the bottom right let you prune the call stacks and focus on the most relevant entries.

Figure 4-9 Spin Control sample window



If you want to view a complete listing of the call stacks, click the “Show text report” button on the main window. This format shows a formatted version of the entire data set that you can copy and paste into other documents.

Document Revision History

This table describes the changes to *Performance Overview*.

Date	Notes
2010-05-14	Updated the Apple developer information.
2009-07-07	Updated the thread guidance to include Grand Central Dispatch.
2008-03-28	Updated for Mac OS X v10.5 and iOS.
2006-10-03	Updated the advice about using Velocity Engine to reflect using the Accelerate framework, which supports multiple platform architectures.
2005-04-29	New document that introduces the factors that determine performance.
	Some information appeared previously in <i>Performance Fundamentals</i> and <i>Finding Performance Problems</i> .

REVISION HISTORY

Document Revision History

Index

A

Accelerate framework [19](#)
acid tool [24](#)
Activity Monitor [24](#)
algorithms, analyzing [13](#)
AltiVec. *See* Velocity Engine
analysis techniques [13–14](#)
atos tool [25](#)

B

BigTop [24](#)
BSD system calls [30](#)
bugs, reporting [7](#)

C

c2ph tool [25](#)
cache files [16](#)
Carbon Event Manager [18](#)
checkpoints [14](#)
code checkins [13](#)
code optimizations [17](#)
constant data [17](#)
context switches [30](#)
copy-on-write faults [30](#)
CPU registers [24](#)
CPU time [10, 28](#)

D

data mining [35](#)
drawing code [15](#)
drivers [33](#)

E

enhancements, requesting [7](#)
event handling [18](#)

F

file systems
 monitoring [16](#)
 performance of [11](#)
files
 closing [16](#)
 reading [16, 18](#)
 writing [16](#)
frameworks [17](#)
fs_usage tool [24](#)

G

global data [10](#)
gprof tool [25](#)

H

heap tool [23](#)
heavy view, in Shark [34](#)

I

IBM 970 processor [25](#)
Instruments [30](#)
Instruments application [21](#)

K

kernel extensions [33](#)

L

laziness, in coding [17](#)

leaks tool [23](#)

M

Mach messages [30](#)

Mach ports [28](#)

Mach system calls [30](#)

Mach-O format [19](#)

MallocDebug [23](#)

malloc_history tool [25](#)

memory footprint [16](#)

memory

allocation patterns [23](#)

deferring allocations [17, 18](#)

finding leaks [17, 23](#)

footprint [10, 16](#)

graphing allocations [33](#)

initializing [18](#)

monitoring allocations [17](#)

overview [10](#)

metrics

analyzing results [13](#)

correlating with code checkins [13](#)

CPU time [10](#)

establishing baseline values [12](#)

file usage [11](#)

gathering [12](#)

memory usage [10](#)

Motorola 7400 processor [25](#)

N

network events [19](#)

network speed [11](#)

nm tool [25](#)

O

OpenGL Profiler [23](#)

optimization tips [17](#)

otool tool [25](#)

P

page faults [29](#)

page-ins [30](#)

pagestuff tool [25](#)

PCI configuration registers [24](#)

perceived performance [11, 18, 19](#)

performance tools

acid tool [24](#)

Activity Monitor [24](#)

atos tool [25](#)

BigTop [24](#)

c2ph tool [25](#)

fs_usage tool [24](#)

gprof tool [25](#)

heap tool [23](#)

installing [21](#)

Instruments [21, 30](#)

leaks tool [23](#)

limitations [13](#)

MallocDebug [23](#)

malloc_history tool [25](#)

nm tool [25](#)

OpenGL Driver Monitor [23](#)

OpenGL Profiler [23](#)

otool tool [25](#)

pagestuff tool [25](#)

pstruct tool [25](#)

Quartz Debug [24, 36](#)

Reggie SE [24](#)

sample tool [25](#)

Saturn [23](#)

sc_usage tool [24](#)

Shark [22, 33](#)

simg4 tool [25](#)

simg5 tool [25](#)

Spin Control [24, 36](#)

Thread Viewer [24](#)

top tool [24, 27](#)

vmmmap tool [23](#)

vm_stat tool [25](#)

performance

analyzing [23, 24](#)

and lazy coding [17](#)

areas to monitor [15–17](#)

definition [9–11](#)

evaluating [27–35](#)

monitoring [23](#)

of key tasks [15](#)

optimization tips [17](#)

tracking [12–14](#)
polling for events [18](#)
private address space [29](#)
process ID [28](#)
profiling code [18, 23](#)
pstruct tool [25](#)

Q

Quartz Debug [24, 36](#)

R

Reggie SE [24](#)
resident private memory [29](#)
resident shared memory [29](#)

S

sample tool [25](#)
Saturn [23](#)
sc_usage tool [24](#)
Shark [22, 33](#)
simg4 tool [25](#)
simg5 tool [25](#)
Spin Control [24, 36](#)
spinning cursor [36](#)
static data [17](#)
symbols [17](#)

T

Thread Viewer [24](#)
threads [19, 28](#)
timers [19](#)
top tool [24, 27](#)
tree view, in Shark [35](#)
tuning tips [35](#)

U

unresponsive applications [36](#)

V

Velocity Engine [19](#)
virtual memory [23](#)
vmmmap tool [23](#)
vm_stat tool [25](#)

W

writable memory pages [10](#)

Z

zero-initialized memory [18](#)