Performance Overview

Performance

2006-10-03

Contents

Figures, Tables, and Listings

Introduction to Performance Overview

Performance is an important design factor in all software products. If a program runs slowly or displays the spinning cursor too frequently, 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.

Filing Bug Reports

If you encounter bugs in Apple software or documentation, you are encouraged to report them to Apple. In addition to reporting bugs, you can also file enhancement requests to indicate features you would like to see in future revisions of a product or document. To file bug reports or enhancement requests, go to the Bug Reporting page of the Apple Developer Connection website, which is at the following URL:

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

You must have a valid Apple Developer Connection login name and password to file bugs. You can obtain a login name for free by following the instructions found on the Bug Reporting 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.
- *Threading Programming Guide* provides detailed conceptual and task-based information about how to use threads in your program.
- *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 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 user perception. 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 Mac OS X implements 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 Mac OS X system (with more than 100 concurrently running 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 Carbon Event Manager, 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 (Velocity Engine or SSE) or the 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 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 paged out to 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. If a page fault occurs in a low-memory situation, the virtual memory system may need to reclaim some of your application's memory pages to make room for another process. If a page has not been modified, the system can reclaim it and immediately overwrite it with new data. However, if the page has been modified, the system must first write the modifications to disk. Writing data to disk adds a significant amount of time to the processing time of a page fault.

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 in Mac OS X is that your application and any files it creates may be on the network instead of on a local hard disk. Mac OS X makes the network as invisible as possible, so you should never make any 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 for optimal performance, it's entirely possible that your application still appears 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. In this situation, you need to give your application the appearance of speed. You can do that with 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. Thus, the perception of speed is achieved by letting the user continue to work while your program processes data in the background. Threading your application is a good way to make it responsive to the user. While your main thread responds to the user, worker threads perform calculations or handle other time-consuming tasks.

Another common way to make your application appears fast is to improve its launch time. An application that takes more than a few seconds 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 19).

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 a document to load in less than 1 second or cause no more than 100 kilobytes of memory to be allocated.

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 will find relevant. If you have an established product, talk to you users and find out what features they consider slow and consider adding those features to your list of tasks.

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 or 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 were in a spreadsheet or other text-based medium. For example, you could plot the time to complete an operation against a particular build to determine if performance is improving or declining from build to build.

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 sample on calls to that function. Look at how many times the function is called and see if there are any patterns there. 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: Shark is one of the more powerful tools you can use to analyze your program. The data mining capabilities of Shark are an excellent way to detect problems in higher-level algorithms. For more information about 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 Shark and other tools gather their data before drawing too many conclusions. These 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 the 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 the time when you initialize your program's data structures and prepare 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 started processing user 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 largely unnecessary in that and later releases.

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 to be sure that the files you use are needed and are used properly.

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

- Understand how the system caches work and know how to optimize the use of those caches. Avoid caching data 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 *Launch Time 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 way to organize code. Alternatively, a function might have been excised from the active code path but remains in the code module.
- Static or constant data is stored on writable pages. During paging, this data is written to disk unnecessarily.
- A program exports more symbols than are actually needed.
- Code is not properly optimized by the compiler and linker.
- Too many frameworks are included by the program. Load only the code you need.

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 find out if autoreleased Objective-C objects are being deallocated before they cause too much paging. You can find memory leaks caused by bugs in your code. You can also watch the number of times you call `malloc`, which might 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 19).

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.

Use Event-Based Handlers

All modern Mac OS X applications should be using the Carbon Event Manager or other event-based model for responding to system events. The old way of retrieving events by polling the system is highly inefficient. In fact, when there are no events to process, polling code is a 100 percent waste of CPU time. Using more modern event-based APIs can lead to the following benefits:

- It makes your program more responsive to the user.
- It reduces your application's CPU usage.
- It minimizes your application's working set—the number of code pages loaded in memory at any given time.

- It allows the system to manage power aggressively.

The Cocoa framework incorporates Carbon Event Manager calls into its classes and methods to implement an event-driven model for you. Applications written in Cocoa automatically take advantage of this behavior and require no additional modifications. Carbon applications must support the Carbon Event Manager calls explicitly.

Event-based handlers are not limited to supporting user events, such as mouse and keyboard events. Each thread has its own run loop to provide on-demand responses to timers, network events, and other incoming data. Applications support run loops using either the Core Foundation (CFRunLoop) or Cocoa (NSRunLoop) interfaces.

Thread Your Program

Supporting multiple threads is a good way to improve both the perceived and actual performance of your program. On hardware containing multiple processors, a multithreaded program often has significantly better performance than a single-threaded program. By distributing tasks across all available processors, an application can perform multiple operations simultaneously. Even on a single-processor machine, the use of additional threads can provide a perceived speed boost by leaving your main thread free to handle user events.

Before you begin adding support for multiple threads, though, be sure to put some thought into how your program might use those threads effectively. Because threads require a fair amount of overhead to create, you should carefully choose which tasks you want to assign to separate threads. If all of your program's tasks are small and performed at different times, you would probably not want to create separate threads for each one. Instead, creating a single long-lived worker thread might be more appropriate.

Another consideration with threading is how to protect your data structures. Problems can occur when multiple threads modify the same data without first checking to see if it is safe to do so. Your code needs to use locks rigorously to protect its data structures. You might also need to synchronize specific blocks of code to prevent them from being executed by multiple threads at once.

For information on how to support additional threads in your program, see *Threading Programming Guide*.

Use the Accelerate Framework

If your 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, 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.

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

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 is 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 may be paged to disk before it can be used. Thus, any savings you gained by caching the data turn 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, wait until a given operation has been performed once before you cache any data from it.

Some other things to be lazy about include the following:

- Defer memory allocation until the point where 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, on a separate thread, or at idle time. Doing this makes the program interface feel more responsive to the user. Of course, creating the perception of performance does not work in every case. For example, the perception may be lost if the data being processed in the background is needed by the user immediately.

As you design your program, think about which tasks can be moved to the background effectively. For example, if your program needed to scan a number of files, do it on a background thread. Similarly, if you need to perform lengthy calculations, do it in the background so that the user may continue to manipulate your program's user interface.

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 you do not need immediately until after your application has finished launching. You should also avoid loading plug-ins until the moment their code is actually needed.

Use the Mach-O Binary Format

If you have a Carbon application that is based on the Code Fragment Manager Preferred Executable Format (PEF), you should consider switching to the Mach-O executable format for several reasons. Foremost among them is that Mach-O is designed and optimized for use with the Mac OS X virtual memory system. Other reasons include the following:

- PEF executables are not supported on Intel-based Macintosh computers.
- In Mac OS X, the libraries that implement the Carbon environment use the Mach-O executable format. Mach-O executables use a calling convention different from that used by PEF executables. Calls made to or from PEF code fragments must be translated at runtime. While the translation overhead is small, it is unnecessary if you are using Mach-O.
- Apple's Mac OS X development environment supports only Mach-O. Whether or not you use Apple's development environment for Mac OS X, the Mac OS X performance tools are significantly easier to use with Mach-O executables than with PEF executables.
- Mach-O executables can directly call other Mach-O shared libraries and BSD API routines in the kernel.
- Mach-O supports just-in-time binding, where a link to a function is resolved when that function is first called. All links in a PEF-based application (and all PEF libraries it links to) must be resolved when the application is launched.

Although Mach-O is not supported in Mac OS 9, using Mach-O does not require you to abandon Mac OS 9 as a delivery platform. You can build an application package that runs a PEF binary in Mac OS 9 and a Mach-O binary in Mac OS X. This allows you to optimize your executable for each operating system that you wish to support. For more information, see *Bundle Programming Guide*.

For an overview of the Mach-O format and how you can take advantage of that format for performance tuning, see "Overview of the Mach-O Executable Format" in *Code Size Performance Guidelines*.

Performance Tools

Mac OS X comes with tools for gathering several different types of performance metrics for your application. Some of these tools can be launched from the Finder and some must be run from the command line. 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 package. This package ships on a CD that comes with retail copies of Mac OS X. You can also download it for free or order a CD from the Apple Developer Connection section of the Apple website.

To install Xcode Tools, double-click the installer package found on the Xcode Tools CD or that you downloaded from the web. The installer creates a `/Developer` directory on the boot volume of your hard drive. Inside this directory are subdirectories containing the applications, documentation, examples, and other files.

The primary performance tools are located in the `/Developer/Applications/Performance Tools` directory but some tools are located in other subdirectories of `/Developer/Applications`. Several command-line performance tools are also installed in the `/usr/bin` directory and are available from Terminal.

Most of the applications with a graphical user interface have online help available through the Help menu. All of the command-line tools have **man pages**, accessed on the command line by typing “`man`*toolname*”.

Analysis Tools

Analysis tools let you actively gather data about the performance of your program. You can view these tools in a way similar to debugging tools. You use them to investigate problems and gather information needed to go back and revise your code. Unlike debugging tools, most analysis tools provide a way to save data from a session so that you can view it later, which is very useful for charting the progress of your application.

With the exception of Shark, most of the analysis tools are geared towards finding specific types of performance problems. While any one tool might give you useful information, it is important to run several tools on the same code to view problems from several different angles. For example, using ObjectAlloc, you might find that your program creates a number of objects, but running MallocDebug you might find that many of those objects are actually being leaked. Shark provides many ways to gather and view information and is indispensable for finding performance problems in your code.

Table 3-1 lists the analysis tools installed with the Xcode Tools. Applications such as Shark are installed in the `/Developer/Applications/Performance Tools` directory. 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.
ObjectAlloc	Tracks Objective-C and Core Foundation object allocations and deallocations in real time. The tool also lets you view the retention history for an object, which can be useful in recovering memory held by over-retained objects.
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.
Sampler	Analyzes your application's behavior at runtime. It can identify where your program spends its time and summarize how often allocation routines, system calls, or arbitrary functions were called.
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.
Shark	Does statistical sampling of all processes and threads in the system. You can also use Shark to trace function calls, including `malloc` calls, and to chart information graphically. Shark helps you to isolate problems quickly by providing a rich set of data-mining features and is an indispensable tool for finding performance bottlenecks.
`heap`	Lists all `malloc`-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.
`vmmap`	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. Applications such as BigTop and Spin Control are installed in the `/Developer/Applications/Performance Tools` directory hierarchy. 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. This tool provides information that is similar to that of the `top` 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 i similar to that of the `top` and `vm_stat` 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.
`fs_usage`	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.
`sc_usage`	Displays an ongoing list of system call and page fault statistics.
`top`	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. The graphical applications are installed in `/Developer/Applications/Performance Tools/CHUD` and the command-line tools are installed in `/usr/bin`. All of the tools are included with the Xcode Tools. Table 3-3 lists the tools that are part of this package and provides a brief overview of their purpose.

Table 3-3 CHUD tools

Name	Description
CacheBasher	Measures cache performance under a wide range of conditions.
MONster	Collects sampling data at a hardware level. The tool can collect samples at a systemwide or process-specific level and display the metrics for the collected data. You can use this tool to gather metrics such as utilized bandwidth, cycles per instruction, and cache miss rates.
PMC Index	Lets you search for available performance counter events. When you select multiple events, the tool notifies you if those events cannot be recorded simultaneously.
Reggie SE	Lets you examine and modify CPU and PCI configuration registers.

Name	Description
Skidmarks GT	Measures integer, floating-point, and vector performance.
`acid`	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.
`amber`	A command-line tool that traces all threads of execution in a process, recording every instruction and data access to a trace file. This tool can generate traces in TT6, TT6E, or FULL format.
`simg4`	A command-line tool that is a cycle-accurate simulator of the Motorola 7400 processor. This tool takes TT6 traces as input.
`simg5`	A command-line tool that is a cycle-accurate simulator of the IBM 970 processor. This tool takes TT6 traces as input.

For information about using the CHUD tools, see online help for a particular tool. Documentation for some CHUD tools is also installed in `/Developer/ADC Reference Library/CHUD`.

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 located in the `/usr/bin/` directory and must be run from a command-line prompt. Most are installed along with the Xcode Tools.

Table 3-4 Command-line tools

Name	Description
`atos`	Converts back and forth between a symbol name and the numeric address of that symbol in a running executable.
`c2ph`	Displays the C-structures from an object file along with their member offset values.
`gprof`	Produces execution profiles based on an execution analysis of a program.
`kdump`	Displays kernel trace data.
`malloc_history`	Shows the `malloc` allocations performed by a specified process.
`nm`	Displays the symbol table information for one or more object files.
`otool`	Displays the contents of a Mach-O executable in a more human-readable form
`pagestuff`	Displays information about the logical pages of a Mach-O executable file.
`pstruct`	Parses the C structures from an object file and displays them along with their member offset values.

Name	Description
`sample`	Produces an execution profile based on the execution analysis of a program.
`vm_stat`	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. Many of the Apple performance tools are designed for specific tasks, such as tracking memory allocations or finding leaks. 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.

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 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, 1.11M 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(1)` 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, `LaunchCFMApp`.)
`%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 (*minute:seconds.hundreths*) 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. The delta value, which is enabled by the `-w` parameter, is relative to the value first displayed when `top` was launched.

Column	Description
`#MREG`	The number of memory regions.
`VPRVT`	The private address space currently allocated (with `-w` parameter only).
`RPRVT (delta)`	The resident private memory. The delta value, which is enabled by the `-w` parameter, is relative to the previous sample.
`RSHRD (delta)`	The resident shared memory. The delta value, which is enabled by the `-w` parameter, is relative to the previous sample
`RSIZE (delta)`	The total resident memory as real pages that this process currently has associated with it. Some may be shared by other processes. The delta value, which is enabled by the `-w` parameter, is relative to the previous sample.
`VSIZE (delta)`	The total address space currently allocated, including shared memory. The delta value, which is enabled by the `-w` parameter, is relative to the value first displayed when `top` was launched. 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 tasks.

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

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.

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

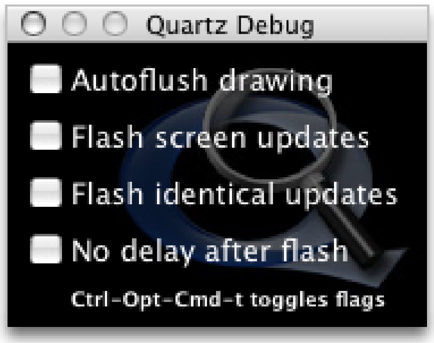
Column	Description
`PID`	The BSD process ID.
`COMMAND`	The name of the executable or application package. (Note that Code Fragment Manager applications are named after the native process that launches them, `LaunchCFMApp`.)
`%CPU`	The percentage of CPU cycles consumed during the interval on behalf of this process (both kernel and user space).
`TIME`	The amount of CPU time consumed by this process (*minute:seconds.hundreths*) 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).
MSGS_SENT	The number of Mach messages sent by the process.
MSGS_RCVD	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 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 4-1 shows the Quartz Debug options window.

Figure 4-1 Quartz Debug options



In its primary mode, 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.

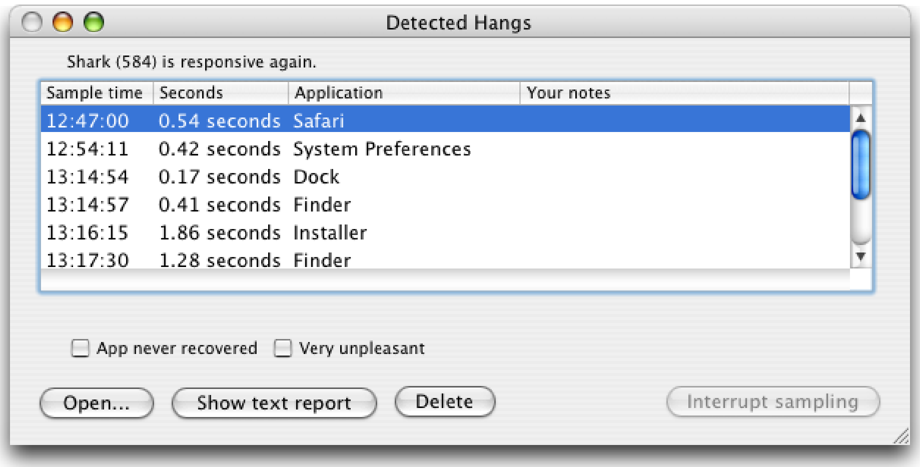
The tool also has a mode that identifies parts of the screen that are being redrawn with the exact same content. If you select the "Flash identical updates" option, the tool displays red over any areas whose resulting bits are identical to the current content.

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 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 Shark or another tool ready to go, you might not be able 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 unresponsive applications. You leave Spin Control running on your computer whenever you are testing your application. When the spinning cursor appears, Spin Control gathers the backtrace information and makes it available from the application's main window, as shown in Figure 4-2.

Figure 4-2 Spin Control main window

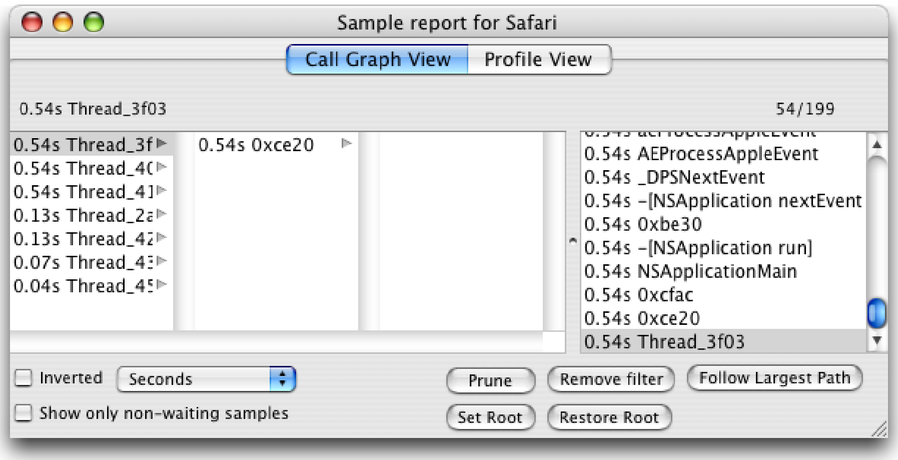


Shark (584) is responsive again.

Sample time	Seconds	Application	Your notes
12:47:00	0.54 seconds	Safari	
12:54:11	0.42 seconds	System Preferences	
13:14:54	0.17 seconds	Dock	
13:14:57	0.41 seconds	Finder	
13:16:15	1.86 seconds	Installer	
13:17:30	1.28 seconds	Finder	

To view the backtrace for a particular session, select that session and click Open. Spin Control displays the browser window (Figure 4-3) along with the sample data. You can use the data in this window to identify the code that was executing when your application became unresponsive. The controls in the bottom-left corner of the window let you 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 call stack entries.

Figure 4-3 Spin Control sample window



```
0.54s Thread_3f03                                                54/199

0.54s Thread_3f   0.54s 0xce20        0.54s AEProcessAppleEvent
0.54s Thread_40                       0.54s _DPSNextEvent
0.54s Thread_41                       0.54s -[NSApplication nextEvent
0.13s Thread_2a                       0.54s 0xbe30
0.13s Thread_42                       0.54s -[NSApplication run]
0.07s Thread_43                       0.54s NSApplicationMain
0.04s Thread_45                       0.54s 0xcfac
                                      0.54s 0xce20
                                      0.54s Thread_3f03
```

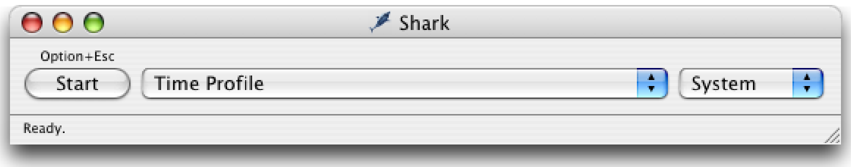
If you want to view a complete listing of the call stacks, click the “Show text report” button on the main window. This format shows a formatted version of the entire data set that you can copy and paste into other documents.

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).

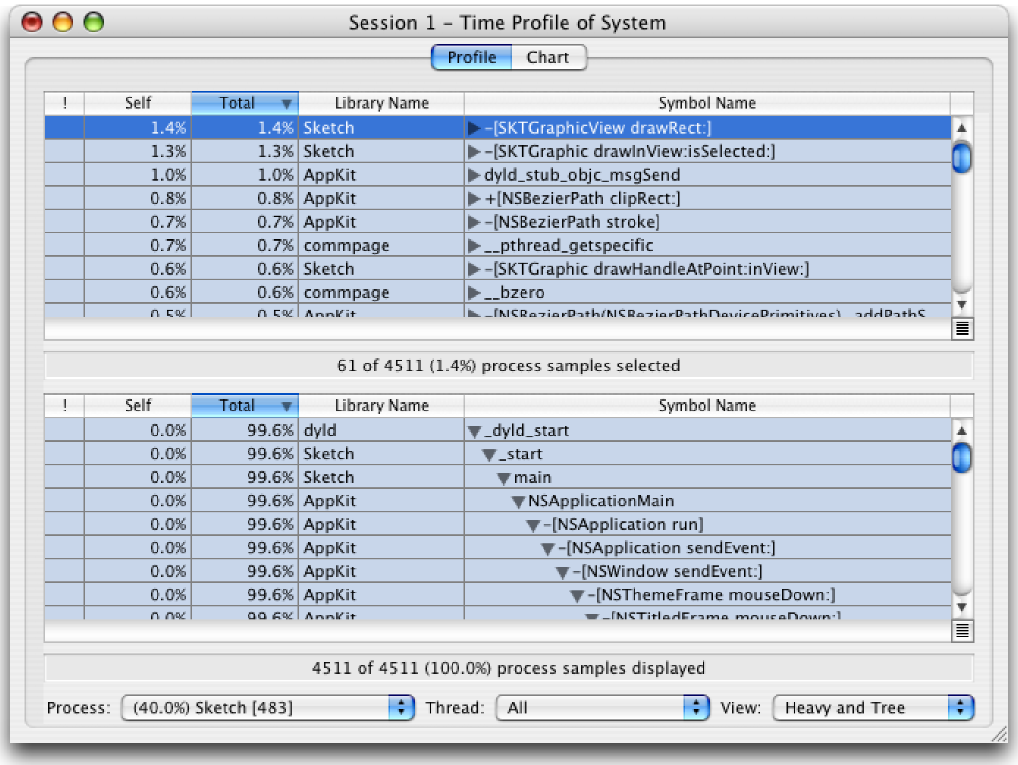
Figure 4-4 Shark launch window



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



!	Self	Total	Library Name	Symbol Name
	1.4%	1.4%	Sketch	▶ -[SKTGraphicView drawRect:]
	1.3%	1.3%	Sketch	▶ -[SKTGraphic drawInView:isSelected:]
	1.0%	1.0%	AppKit	▶ dyld_stub_objc_msgSend
	0.8%	0.8%	AppKit	▶ +[NSBezierPath clipRect:]
	0.7%	0.7%	AppKit	▶ -[NSBezierPath stroke]
	0.7%	0.7%	commpage	▶ __pthread_getspecific
	0.6%	0.6%	Sketch	▶ -[SKTGraphic drawHandleAtPoint:inView:]
	0.6%	0.6%	commpage	▶ __bzero
	0.5%	0.5%	AppKit	▶ -[NSBezierPath(NSBezierPathDevicePrimitives) _addPathS

61 of 4511 (1.4%) process samples selected

!	Self	Total	Library Name	Symbol Name
	0.0%	99.6%	dyld	▼ _dyld_start
	0.0%	99.6%	Sketch	▼ _start
	0.0%	99.6%	Sketch	▼ main
	0.0%	99.6%	AppKit	▼ NSApplicationMain
	0.0%	99.6%	AppKit	▼ -[NSApplication run]
	0.0%	99.6%	AppKit	▼ -[NSApplication sendEvent:]
	0.0%	99.6%	AppKit	▼ -[NSWindow sendEvent:]
	0.0%	99.6%	AppKit	▼ -[NSThemeFrame mouseDown:]
	0.0%	99.6%	AppKit	▼ -[NSTitledFrame mouseDown:]

4511 of 4511 (100.0%) process samples displayed

Process: (40.0%) Sketch [483] Thread: All View: Heavy and Tree

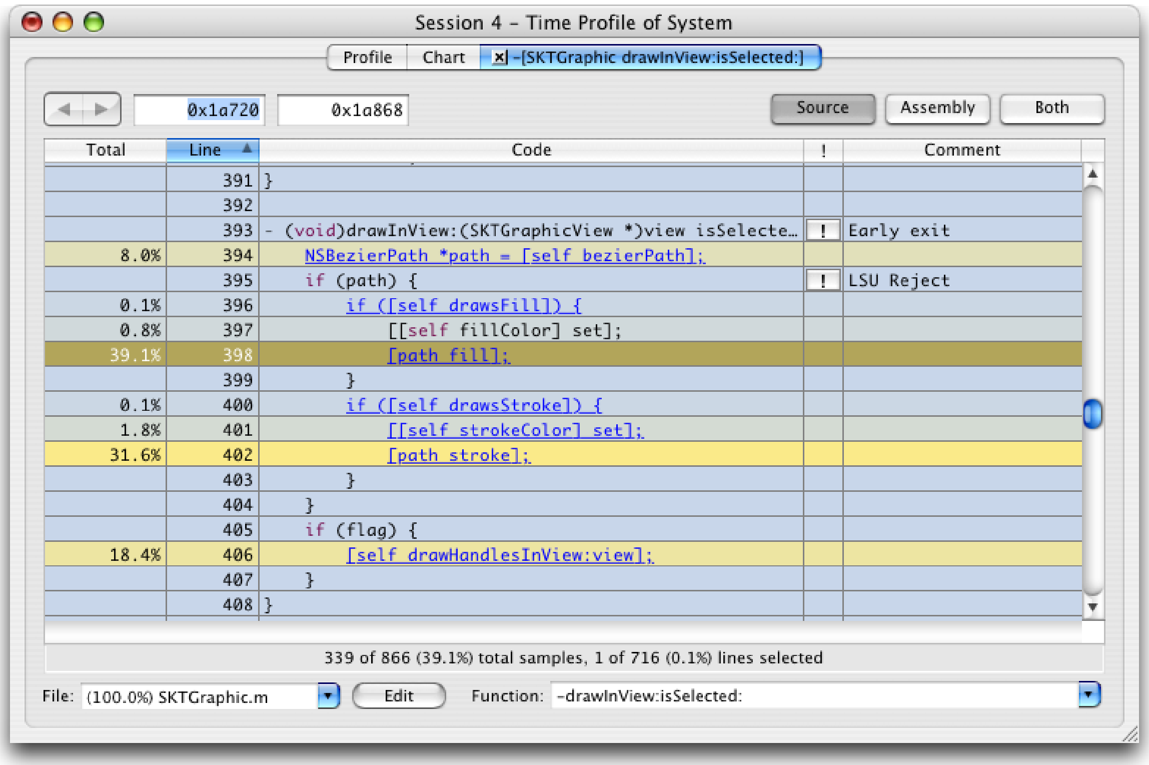
The heavy view sorts function calls based on the frequency by which they appeared. This view identifies your program's hot spots. If you see one of your program's functions near the top of this view, you should investigate. Functions tend to have higher weights if they are poorly optimized but a more 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


Session 4 – Time Profile of System
Profile
Chart
-[SKTGraphic drawInView:isSelected:]
0x1a720
0x1a868
Source
Assembly
Both

Total	Line	Code	!	Comment
	391	}		
	392			
	393	- (void)drawInView:(SKTGraphicView *)view isSelecte…	!	Early exit
8.0%	394	NSBezierPath *path = [self bezierPath];		
	395	if (path) {	!	LSU Reject
0.1%	396	if ([self drawsFill]) {		
0.8%	397	[[self fillColor] set];		
39.1%	398	[path fill];		
	399	}		
0.1%	400	if ([self drawsStroke]) {		
1.8%	401	[[self strokeColor] set];		
31.6%	402	[path stroke];		
	403	}		
	404	}		
	405	if (flag) {		
18.4%	406	[self drawHandlesInView:view];		
	407	}		
	408	}		

339 of 866 (39.1%) total samples, 1 of 716 (0.1%) lines selected

File: (100.0%) SKTGraphic.m Edit Function: -drawInView:isSelected:

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, choose Help > Shark Help to display the Shark User Guide.

Document Revision History

This table describes the changes to *Performance Overview*.

Date	Notes
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 *Performance Fundamentals* and *Finding Performance Problems*.

Index