Optimizing Your Application with Shark 4

If your application could benefit from improved performance—and let's be honest, most can—you need to know about Shark 4, the latest update to Apple's remarkable performance optimization tool. Shark enables you to very quickly identify where your application's performance problems lie, down to the specific functions on which you should concentrate your optimization efforts. You can then focus on the fixes that will yield the maximum benefits. Developers using earlier versions of Shark have found and fixed problems that improved performance dramatically, in a matter of hours.

Shark 4 has added many features that enhance the gathering and analysis of performance data. You will find a much improved user interface that presents you with more powerful options. You will find data mining features that can be of tremendous assistance to you in analyzing high-level performance problems. Low-level analysis has also been improved with the ability to view source and assembly side-by-side. Java developers can now use Shark to optimize their applications, and all developers can take advantage of the Network profiling capabilities.

Getting Started with Shark

The prime directive in optimization is to measure first. Before you start optimizing code, use Shark to identify the best candidates for optimization. Much of your code *could* be optimized, but you need to make sure that your effort is time well spent. For example, you don't want to begin by putting a lot of effort into tweaking code that runs on a background thread and improving the performance a fraction of a percent. Concentrate your efforts on improvements that an end user will perceive in their normal use of the application.

The current version of the CHUD (Computer Hardware Understanding Developer) tools is available from the ADC Member site— you can register as an ADC member for free. When you download the Shark application, be sure you download version 4. If you have a previous version of Shark, you can find, download, and install the latest version by running the CHUD Updater on your Macintosh and it will download the newest version of Shark for you. You can do this from within Shark by selecting Help> Check for Update or by double-clicking on the Updater located in the `/Developer/Applications/Performance Tools/CHUD` directory. Shark 4 runs on Mac OS X v.10.3 Panther and beyond.

To start working with Shark 4, you also need an application to profile. In this article, we are going to look at two sample applications that illustrate Shark's features and power.

Let's begin by running a time profile for the Celestia application. You should download the latest version of the Celestia.DMG file, also at the ADC Member site. This version of Celestia includes changes added by the Shark team at Apple that demonstrate the performance improvements they were able to make with the assistance of Shark 4. Download the CelestiaDemo.dmg disk image and mount it. Go to the `CelestiaDemo/macosx` folder on your Macintosh and double-click the celestia.xcode project file to open it.

In Xcode, make sure that you are creating a deployment build by selecting the Build > Detailed Build Results menu itemm and using the pull-down menu labeled Active Build Style. Do this to make sure that you are profiling a build that takes advantage of the compiler optimizations that are often turned off in developer builds.

Run the Celestia application, and you will see the application, similar to what is shown in Figure 1.

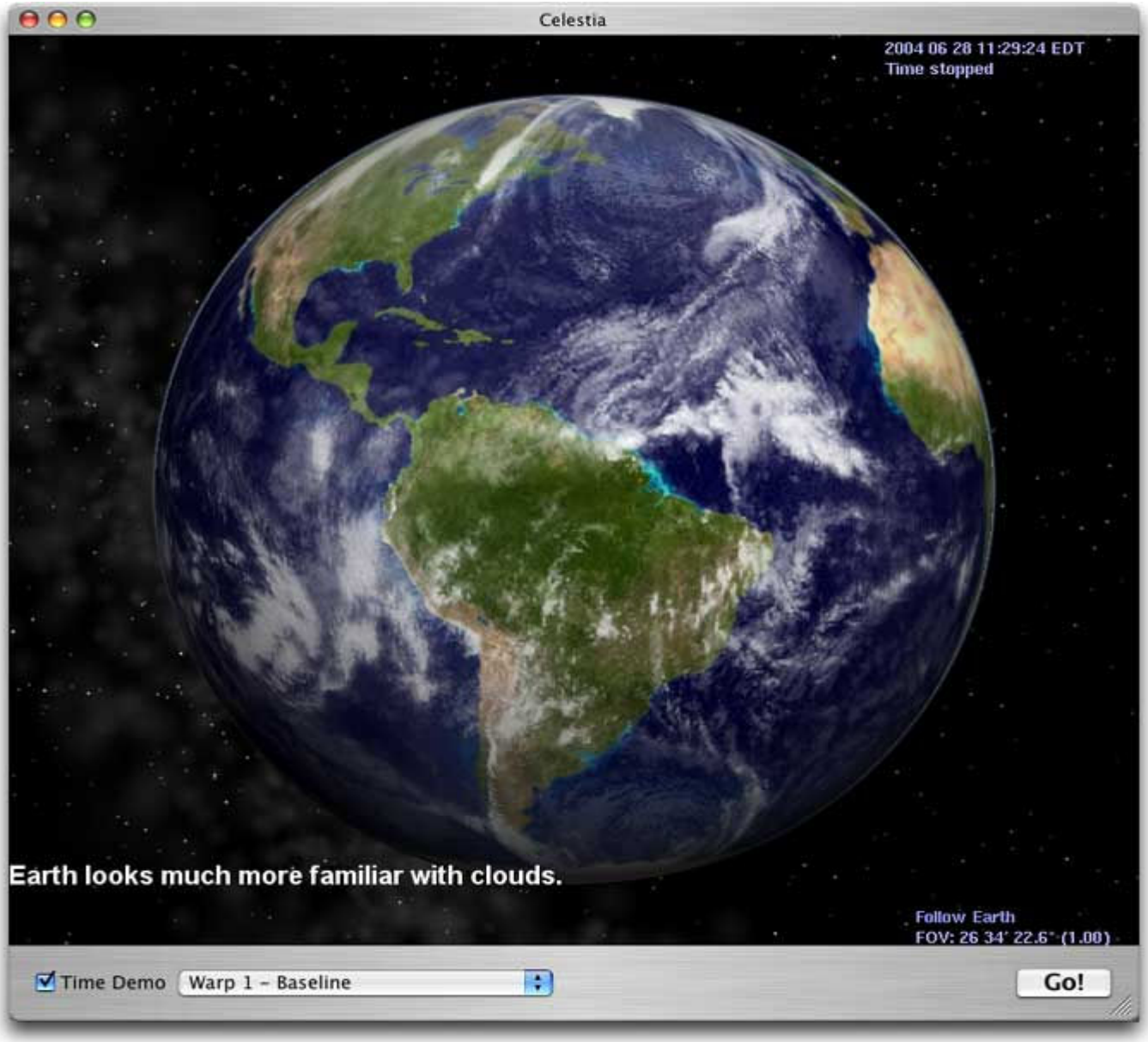

Figure 1: The Celestia Demo Application.

Note that at the bottom of this window, a number of things have been added for the demo that are not part of the Celestia application—a checkbox, a pull-down menu and a button. You can play with the different optimizations that were performed by the Apple engineers which, in the tradition of space travel adventures, they have labeled Warp 1 through Warp 9. The button to begin the demo is labeled Go! These have been added to make the application more useful for working with Shark 4.

Identifying the Hot Spots

A good place to start investigating an application for potential optimizations is with a Time profile. This will help you determine whether you are spending a lot of time in a particular hot spot in your code. There are often more gains to be had by slightly improving the performance in a function that is frequently called than in dramatically optimizing a function that is almost never called.

With Celestia running, start up Shark 4. By default, Shark is configured to take a Time Profile of Everything. This is the option that most developers will use most of the time. After the snapshot is taken, you can filter the view to show the results for a particular thread or process. It is often useful to view what percent of total processor time is spent in the process you are profiling. For those with different needs, there are fourteen available configurations. You can also specify your preferences within many of the configurations. For example, if you change the drop down list from Everything to Process the Shark window expands horizontally to display a pull-down list of all of the running processes. For the reasons given above, it is best to use the default values. Ensure that Shark is configured to take a time profile of everything as shown in Figure 2.

In Celestia press the Go! button to restart the demo and then return to Shark and press its Start button. Shark begins taking snapshots of the state of all running applications every millisecond. The snapshot includes the thread and instruction address as well as the function call stack for the currently running process. When you collect this information one thousand times each second, you can begin to see patterns in how the application is working. For our purposes, after ten seconds or so press the Stop button to end the capture of information.

Figure 2: Running Shark 4.

Shark then processes the samples for a few seconds, caching relevant symbols, source files and program text. It then displays the results in a Profile window as shown in Figure 3. At the bottom of the window you will notice that Celestia did not require all of the processor's attention. On a dual CPU system, the Process pop-up list should report that Celestia required about half of the processor's time. Now, use the thread pop-up list to select the thread on which Celestia ran. In these results, it is a single-threaded application.

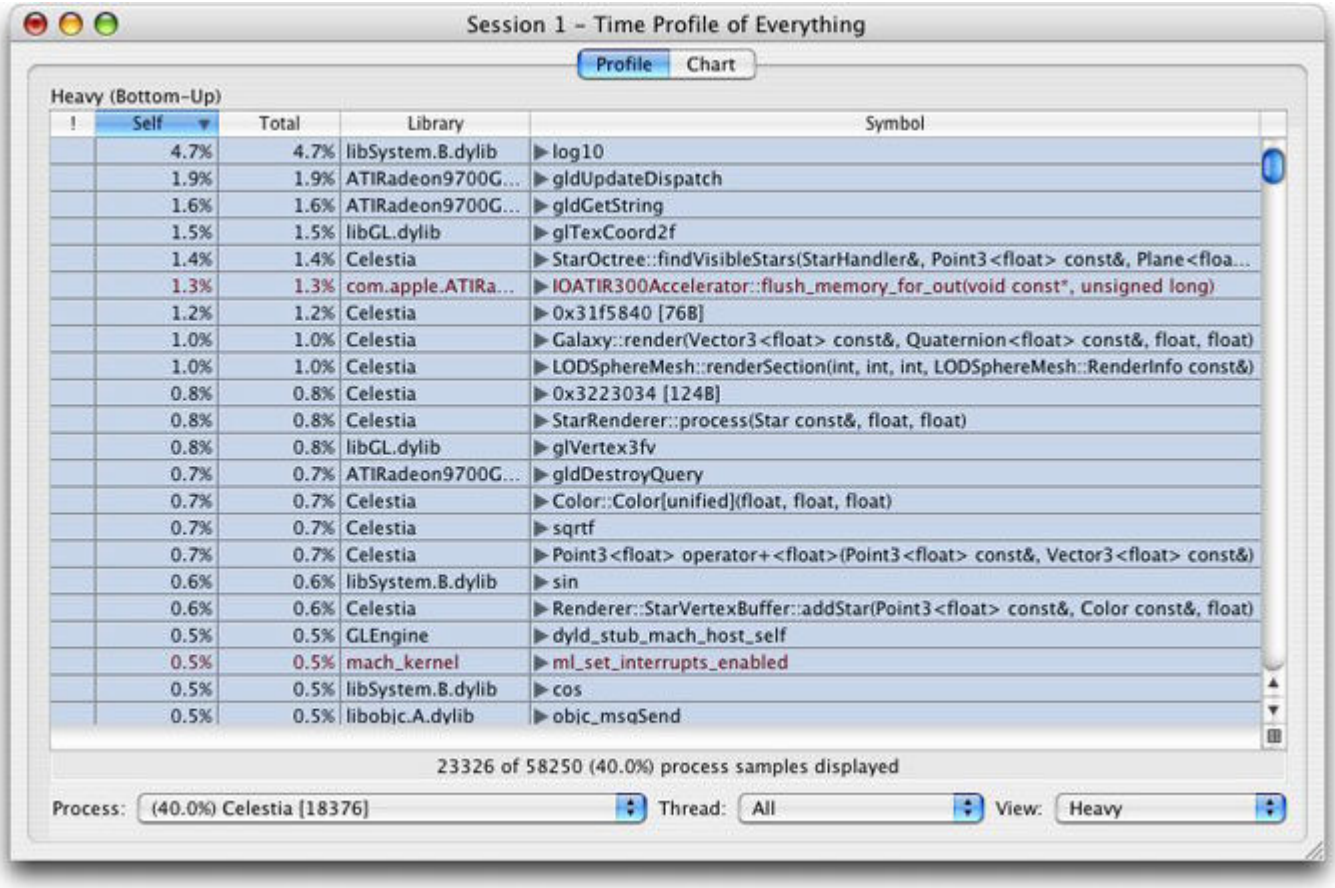

Figure 3: Initial Profile View.

At the bottom of the Profile window, you have a choice of a Heavy view, a Tree view, or both. Select the Tree view and look into that first tree. You can see that the bulk of the time is spent calling in to the `draw()` method in CelestialCore. For the most part, you will want to isolate where the greatest time is being spent in the code you write, as there is little you can do about the code in AppKit, Core Foundation, and HIToolbox. As most of the time is spent in `CelestiaCore::draw()`, this is a good starting place for further investigation.

Data Mining

One thing that quickly becomes clear as you start using Shark is that you are gathering a lot of data. Shark 4 includes Data Mining filtering facilities to allow you to hide some of the information that you are not currently interested in—so you can focus on what really is important.

In this section you will see how to focus on the most relevant data. The information that you choose to hide is not deleted in any way; you are only filtering what will be displayed. Look at the Library column in the expanded tree that is shown in Figure 3. The only items that you have much control over are colored green. It might be useful to hide the interior of some of the framework libraries such as AppKit, CoreFoundation, and HIToolbox. Click on the line containing `_handleWindowNeedsDisplay` and then select the Data Mining menu. Your choices should look like those shown in Figure 4.

Figure 4: The Data Mining Menu.

Your choices include the ability to charge the selected symbol or its library to callers and to flatten the library. (We will guide you through an example of using the Charge option in the Java section below.) The choice to flatten a library results in the details of what happens inside of that library being obscured. This makes it easier to locate problems with the code that you can address. In this example, when you flatten the AppKit library each grouping that is currently colored in blue is collapsed into its top line. So in the first grouping you know that you called into AppKit in the class `NSApplicationMain` and the next non-AppKit call was the HIToolbox `BlockUntilNextEventMatchingListInMode`. Similarly, you entered the AppKit Libraries with `_handleWindowNeedsDisplay` and the next non-AppKit call was in the Celestia library. The result is shown in Figure 5.

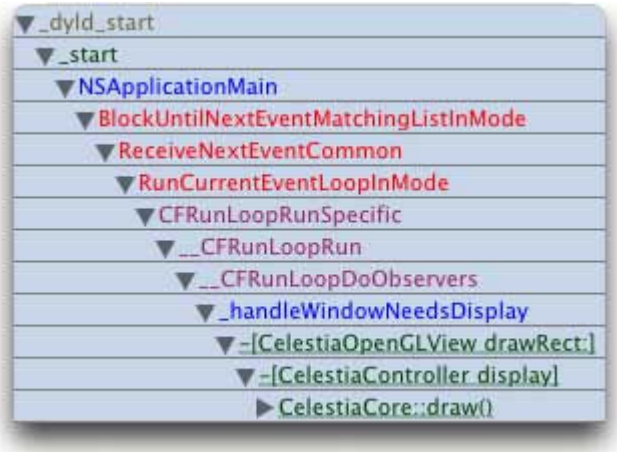

Figure 5: Flattening the AppKit Library.

For heavy duty data mining, you want to use the drawer containing the Profile Analysis and Data Mining options to set some general settings. Select View > Show Advanced Settings or use the keyboard shortcut Shift-Command-M. You see a drawer appear next to the window as shown in Figure 6.

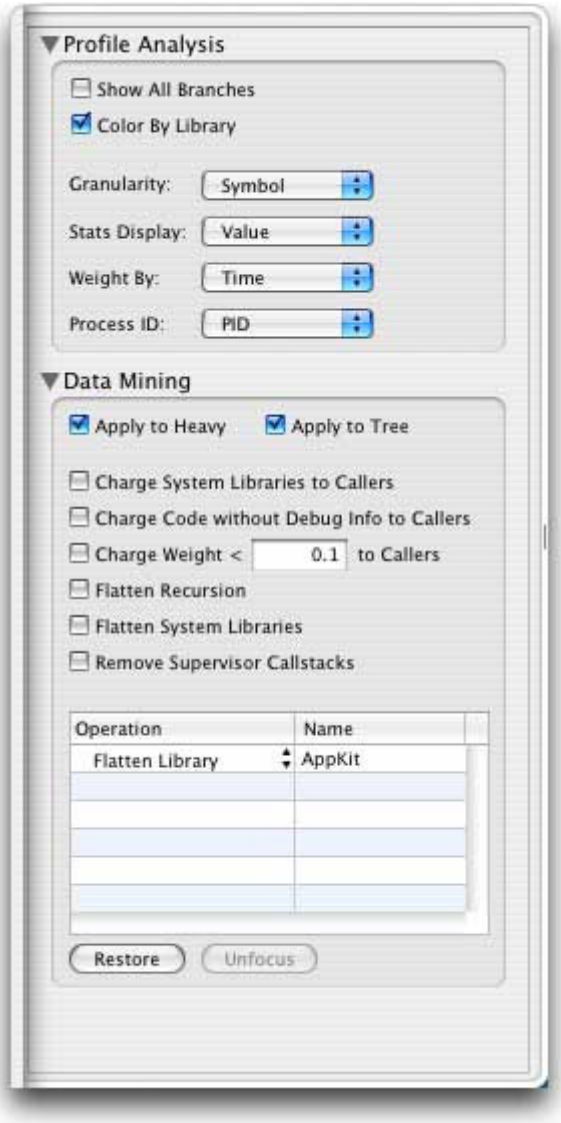

Figure 6: Advanced Settings for Data Mining.

At the bottom of the drawer, you can see the indication of the last action you performed in flattening the AppKit Library and see that you can restore it; or click on Flatten Library if you want to change it to Charge Library. In the Profile Analysis section of the drawer, checking the second checkbox enabled the color coding of the tree by Library. If you change the Stats Display to Value, and you change the Weight By to Time, you will see the time spent in each part of the tree as opposed to viewing the percent of the total time.

Under data mining there are options to charge code for which you have no debug information, which is in a system library, or which has a weight below a certain threshold which you can set. Selecting any of these helps keep the noise down in the output and allows you to quickly locate problems that are likely to be both important and in code you can access. For example, restore your previous change and instead check the checkbox labeled Charge System Libraries to Callers. Now the tree should just display lines from the non-System Libraries as is shown in Figure 7.

Figure 7: No System Calls in the Tree.

As you choose different options in the data mining menu or in the advanced settings view, remember that you are not discarding any of the underlying data; you are only changing how that data is represented. You can always restore the view to study the data in a different way. Another nice feature in Shark 4 is that when you end a session, your data mining settings are saved so that when you start up again you do not have to go through the steps of massaging your data again.

The Chart View

Before heading to the chart view, uncheck Charge System Libraries to Callers and click on the line in the tree labeled `CelestiaCore::draw()`. Now click on the Chart tab and you get a visual display of the time profile. The horizontal axis is time and the vertical axis is the call stack with the longer running processes at the bottom.

Because you selected draw(), the items highlighted in yellow show you when draw() was called. You can use the slider at the bottom left to expand or contract the time axis and the slider in the middle to display a particular time in the window. In Figure 8, you can see a particular slice of user time (blue) that is bracketed by kernel calls (maroon).

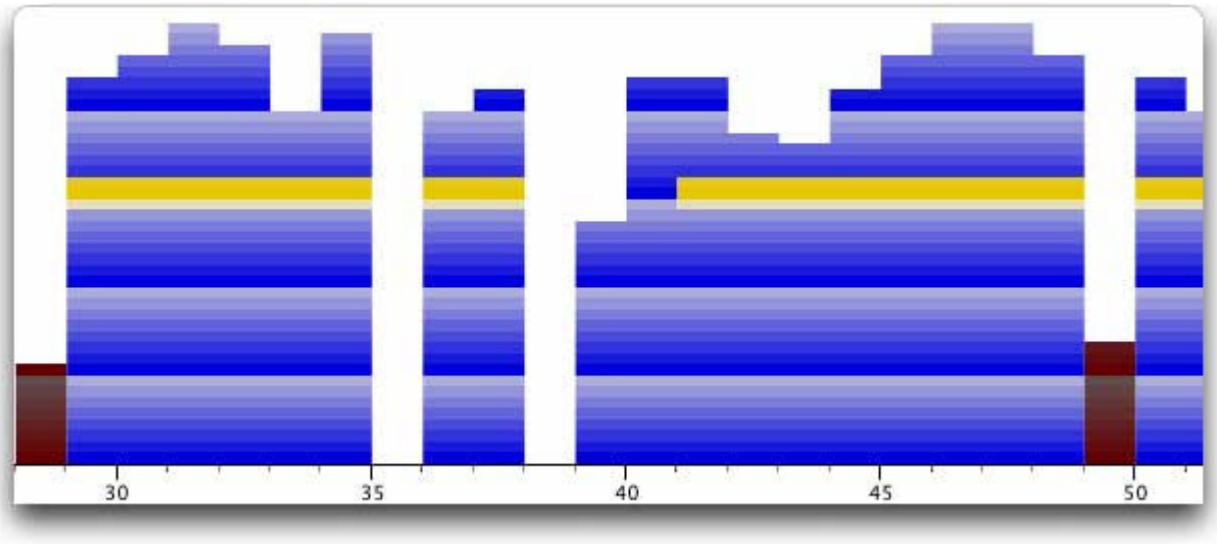

Figure 8: Overly frequent calls to draw().

You can see that the code is making overly frequent calls to the draw() method. In fact, click on the kernel calls and you see that these include calls to the graphics card. There is a classic comedy scene from American television where characters named Lucy and Ethyl are shown in a factory wrapping pieces of chocolate as they come down a conveyor belt. As the conveyor belt speeds up, the chocolate comes too fast for the women to keep up and chaos ensues, which can be amusing, but is costly; and optimizing the conveyor belt will not help; rather, the benefit comes from not providing chocolates faster than the women can wrap them. In the case of Celestia, as in many video games, the call to redraw the screen is made many more times a second than the screen can possibly be refreshed.

If you return to your demo of Celestia and change from Warp 1 to Warp 2, you can see the effect of reducing the number of calls to draw. In fact, by requesting to redraw the screen less often, it appears that you are redrawing the screen more often. You are not actually redrawing the screen at a significantly different rate but you are freeing up the application to do more calculations in between screen redraws so that the application is much more responsive. As you play with different Warp speeds you will notice that the application does get faster as the programmers made Celestia multithreaded and addressed issues with cosine. Even given these other improvements, the most dramatic improvement comes from limiting the frequency of the calls to draw().

Digging into the Code

Once you have located a problem, it may be time to dig into the relevant code a little bit and explore. In the case of the draw() method identified above, the issue is not what is happening within the call so much as with the frequency with which it is called. Set Celestia to Warp 5 and take another Time Profile. On the Profile Tab double-click on the line containing `BigFix::BigFix[unified](double)`. A tab will open up labeled `BigFix::BigFix[unified]`. In the top right corner you will have the options Source, Assembly, and Both. Select Both and you should see something like what is displayed in Figure 9.

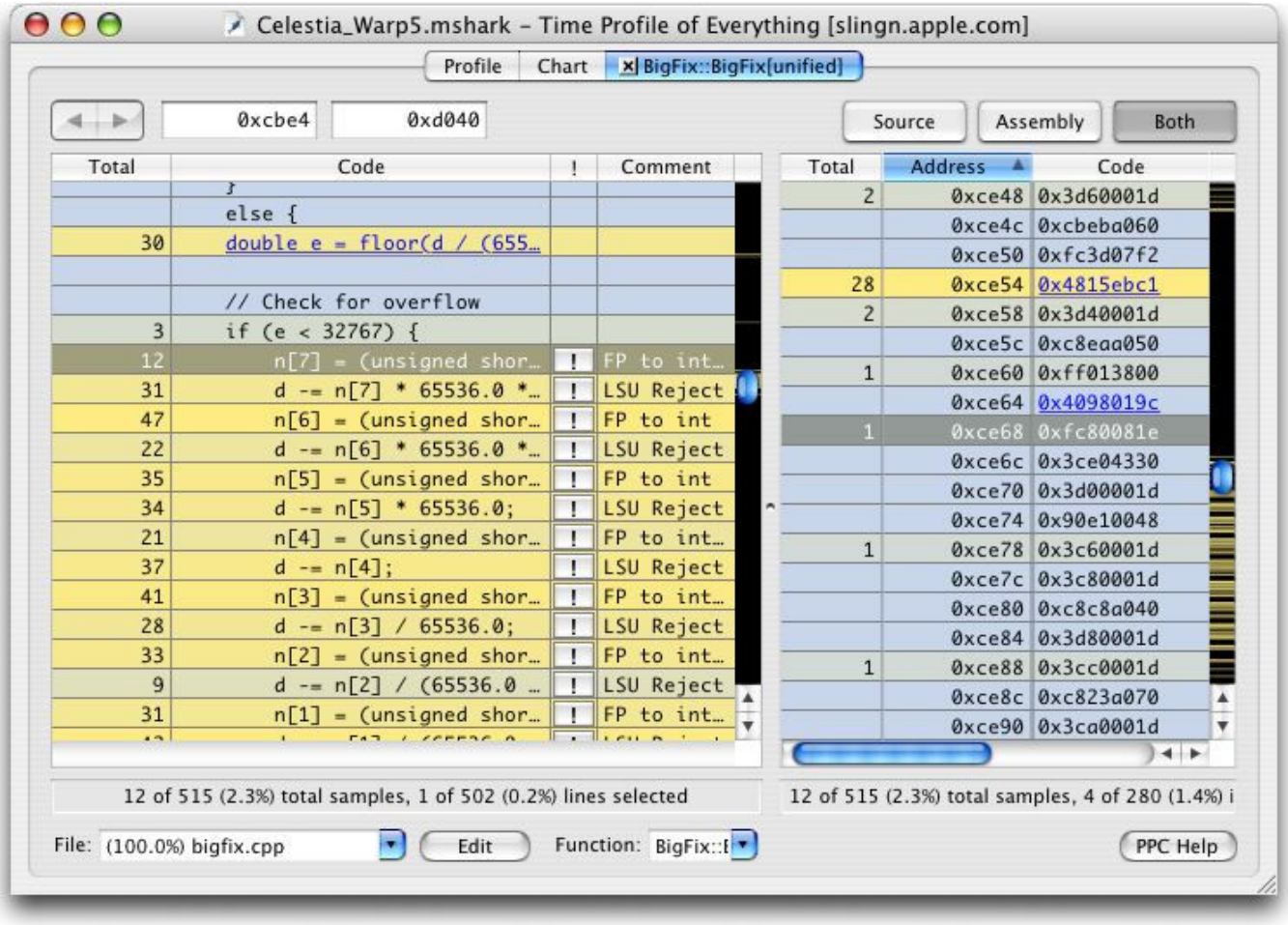

Figure 9: Code Profile.

You can see that a lot of time is spent converting between a floating-point and an integer. The comment in many of the highlighted lines is "FP to int". Press the PPC Help button and click on any line of assembly to show the help information for the assembly code. For example, in Figure 10 you can see the information displayed for Floating Convert to Integer Word with Round toward Zero.

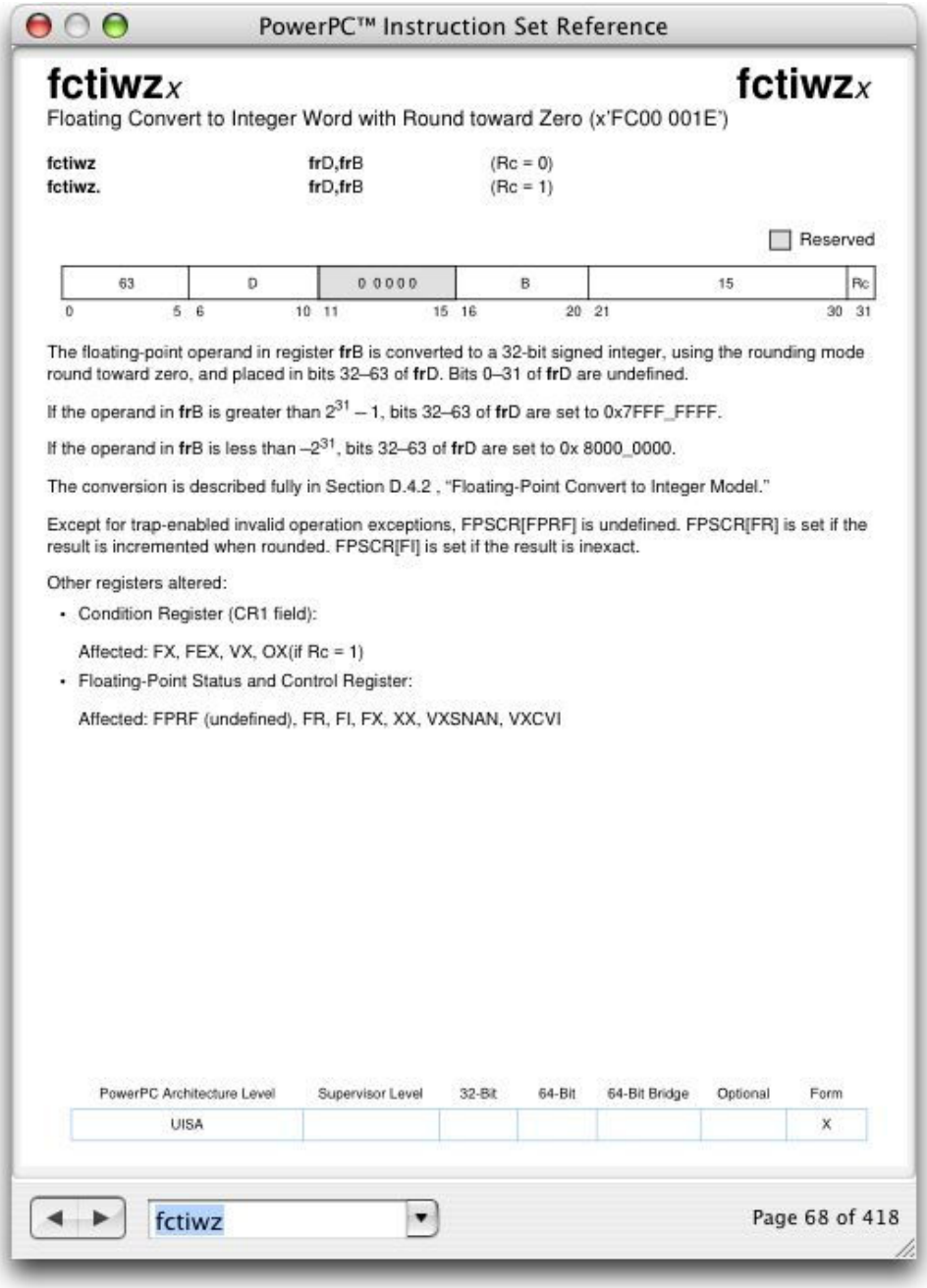

Figure 10: Assembly Instruction Reference.

The PPC Help window is live. If you click on another line of assembly code, the window will refresh with the reference information about that assembly instruction. You can also search or navigate through the PPC instruction reference to find information on other instructions.

Profiling from Afar

Network Profiling has been introduced in Shark 4. Shark has been network enabled so that Shark running on one machine can trigger the Shark instance running on another machine to start and stop gathering information. On the target machine open the Network Manager by selecting the menu item Shark > Network Profiling.. or by using the keyboard shortcut Shift-Command-N. You will see a window similar to the one shown in Figure 11 with the Profile this computer radio button selected.

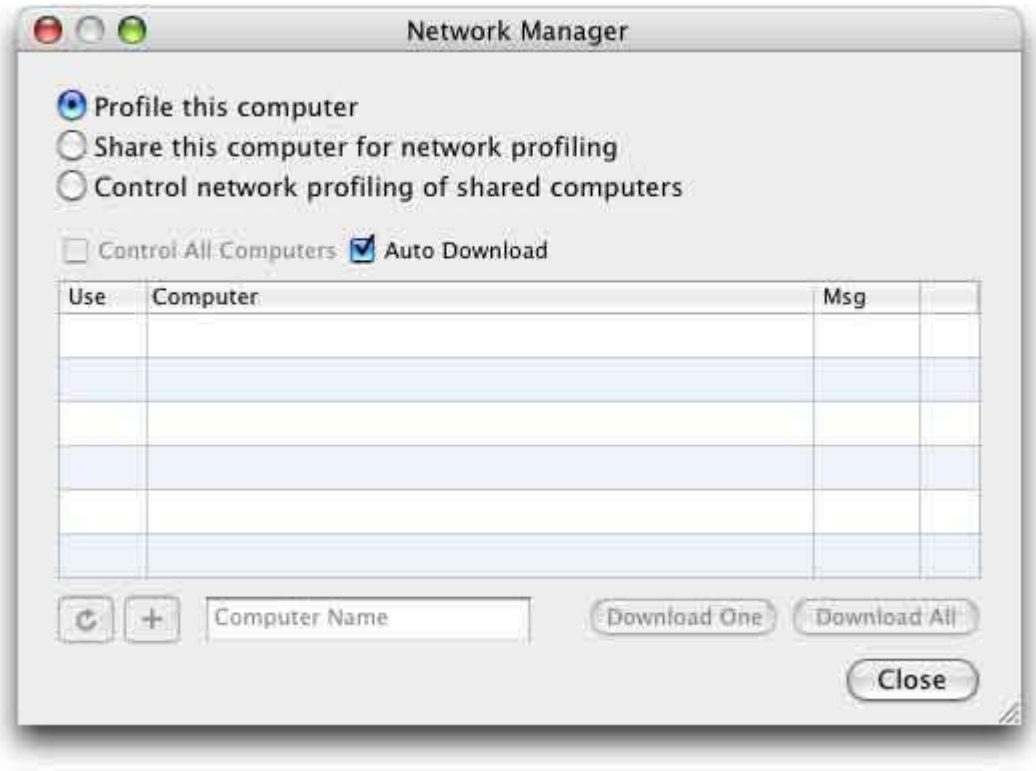

Figure 11: Configuring the Target Machine.

To enable this instance of Shark to serve as a target, select the radio button labeled Share this computer for network profiling. If you are running a firewall, you will be prompted to use the System Preferences > Sharing > Firewall panel to open up the ports in the range 7475-7480. If you click on the "Sharing" button in the warning window you will be taken to this panel and can use New... to create a group named Shark with this range. Return to the Network Manager and select that radio button to "Share this computer for network profiling" and the port will be displayed on the same line.

On the client side, select Control network profiling of shared computers. Add computers on the local link using their .local name. In Figure 12 you can see that two local machines can be remotely profiled.

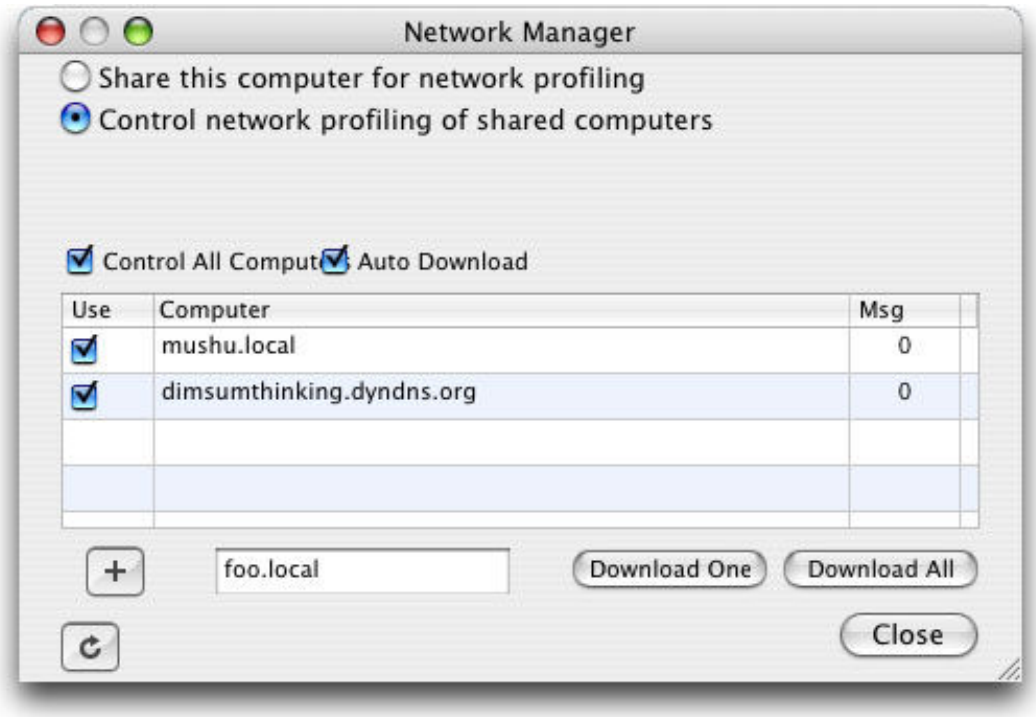

Figure 12: Configuring the Client Machine.

Shark needs to be running on the target machine and sharing must be enabled. You then select the target machine from the client machine and decide which profile you are creating and press Start. After you have finished and press Stop the data will be processed on the target machine and then streamed to the client machine. This allows you to use one machine to gather and analyze your data for running an application on a variety of platforms. You can profile the same application on single processor and multi-processor machines. You can investigate the differences between performance on G3s, G4s, and G5s.

Profiling Java Applications

Shark can now profile Java applications running on Mac OS X. In the Shark configuration pop-up list, you will see Java Alloc Trace, Java Method Trace, and Java Time Trace. You can use a demonstration application called Bouncy—download the Bouncy.DMG file, also on the ADC Member site. Note that unlike Celestia, Bouncy was created to highlight the Java profiling features of Shark and has a deliberate error with the fix included in comments. Run a Java Time Trace on Bouncy and you will see that most of the time is spent in the tree shown in Figure 13.

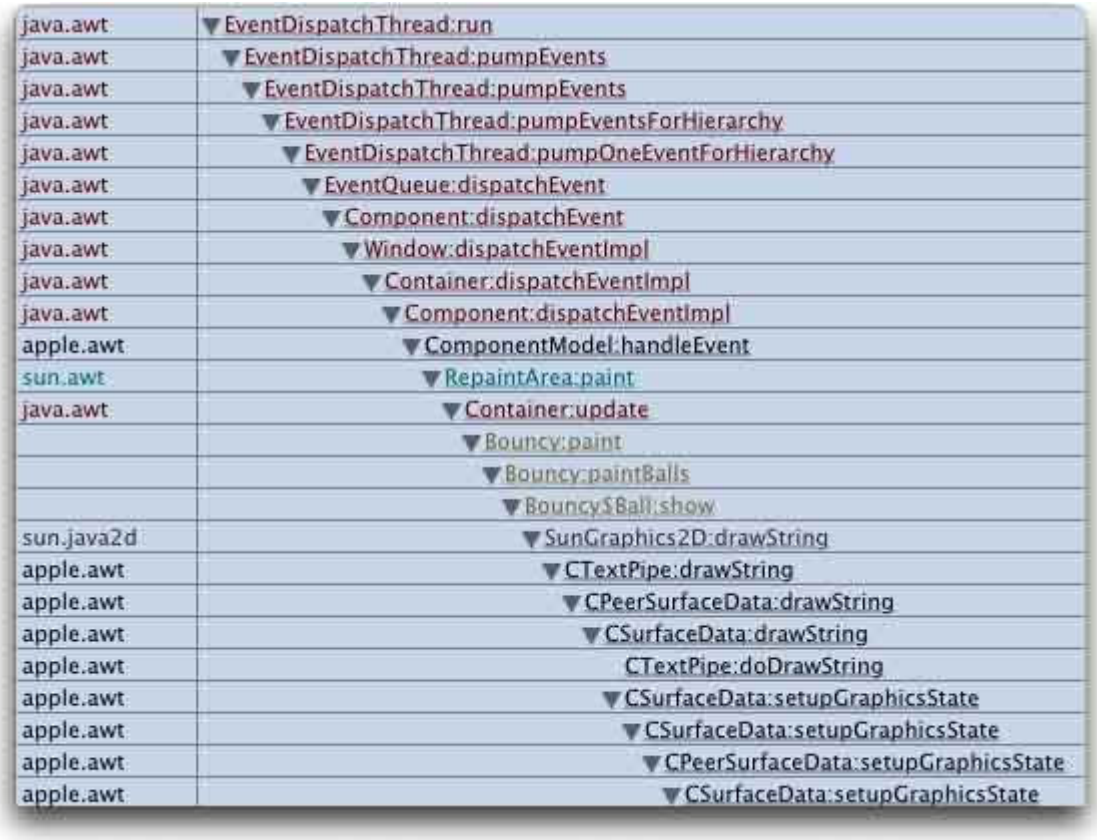

Figure 13: Time spent drawing in Bouncy.

The library names are listed on the left and the tree has been color coded by library. Notice that the three lines that we are particularly interested in have no library associated with them. As before, there is not much you can do to optimize the client libraries. These java.awt and sun.java2d libraries are not recognized as system libraries to checking that check box does not turn these off. This time you are going to charge anything in the apple.awt library to its caller. Highlight one of the lines on which the library is apple.awt and select Data Mining > Charge Library apple.awt to Callers. Do the same for the library sun.java2d.

Next look at the tree above the line containing `Bouncy:paint`. These are all method calls within the Java core classes that you can do nothing to tune. This is a sequence of events that results in the area of the screen being redrawn. Suppose that you are not really concerned with what happens internally before the update() method is called in the Container class. Highlight the line with `Container:update` and select Data Mining > Focus Symbol Container Update. This will cause the tree to be rooted at that line as is shown in Figure 14. In fact, this is the entire visible tree at this point.

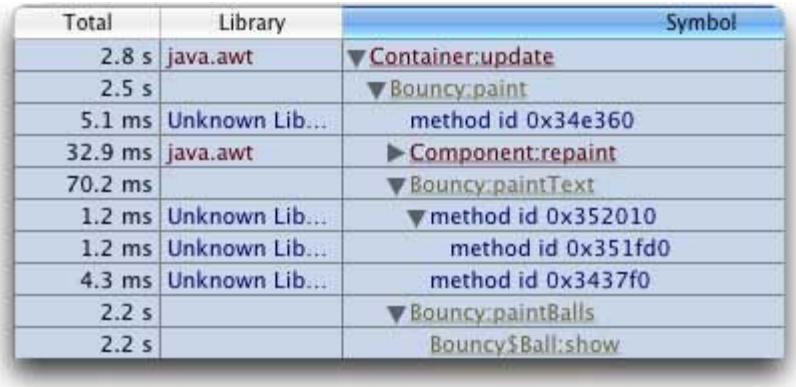

Figure 14: The result of Focus and Charge.

You can see that most of the time is being spent in Bouncy:paintBalls so double-click on that line. If you have not already done so, you will be prompted to navigate to the source file Bouncy.java. As with the Celestia

example, you are directed to the lines that are taking the most time. Navigate to line 174-188 in the source code and you will find that objects are being unnecessarily created every time the window is repainted. Both the offending code and a fix are included in these lines.

```
public void paintBalls(Graphics2D g, Ball balls[], int numberOfBalls) {
  for (int i=0; i < numberOfBalls; i++) {
    if ((i % 2) == 0) {
      // BOTTLENECK!!! replace with LUCIDA global
      //g.setFont(LUCIDA);
      g.setFont(new Font("Lucida", Font.PLAIN, 20));
    } else {
      // BOTTLENECK!!! replace with TIMES global
      //g.setFont(TIMES);
      g.setFont(new Font("Times", Font.PLAIN, 10));
    }
    ((Ball)balls[i]).show(g);
  }
}
```

Uncomment the line `g.setFont(LUCIDA);` and comment out the line `g.setFont(new Font("Lucida", Font.PLAIN, 20));` to replace the bottleneck code which creates a new Font object every time it is called with code that uses a single global. Similarly, switch in the global version for the TIMES font. You will immediately notice the speedup in performance of the application when you save, compile, and rerun Bouncy with these changes.

Tracing Memory Allocation

Another useful data point, when you are optimizing an application, is to look at the memory allocation. You can do this using the "Malloc Trace" configurations for C-based applications and with the "Java Alloc Trace" configuration for applications written in the Java programming language. Evaluate the initial version of Bouncy with the Java Alloc Trace configuration and on the Chart tab you will see that the Alloc Size chart looks like the one shown in Figure 15.

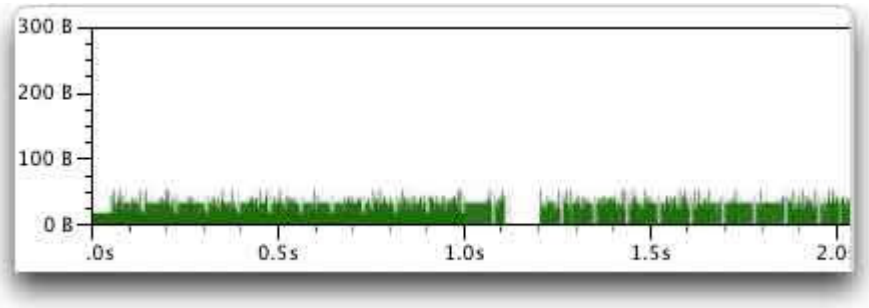

Figure 15: Creating too many objects.

You can see that, with an outlier between 1.1 and 1.2 seconds, a lot of memory is being allocated all the time. The chart shows every time memory is allocated or freed. Contrast that with the first two seconds taken in this snapshot after the changes in the creation of the Font objects have been made.

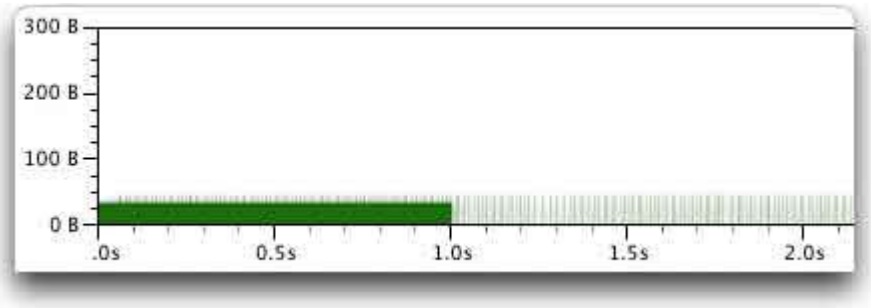

Figure 16: Reusing objects.

In Figure 16 you can see that the same amount of memory was allocated in the first second and then memory usage tailed off dramatically. You can see a more striking pair of images by comparing the aggregate memory allocation. Open the additional settings using View > Show Advanced Settings. In the Perf Counters table check the sigma checkbox in the Alloc Size row. Now you will see this profile for the first 8 seconds of the initial run of Bouncy before any performance enhancements are made.

Figure 17: Totally too many objects.

Contrast this with the first 8 seconds of the run of Bouncy after the performance enhancements are made. In Figure 18 you see a similar shape to that shown above in Figure 17. You should note the difference in the label on the vertical axis.

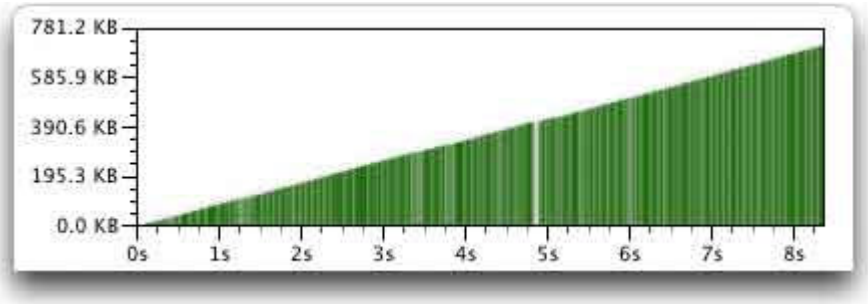

Figure 18: Totally not so many objects.

The aggregate for the first eight seconds in the unoptimized version is more than 3.5MB while the aggregate for the optimized version for the first eight seconds is less than three quarters of a single MB.

When to Use Shark

Deciding when to use Shark in your development cycle is a matter of judgment and application development style. You can tread a middle ground and optimize early to point to architectural decisions that must be made. You can also optimize later to identify the low level optimizations that are best made after the code base works and passes functional tests.

For More Information

- For an introduction to optimization with Shark, read Optimizing with Shark: Big Payoff, Small Effort which covers Shark 3.0, but from a beginning level.
- For performance issues and resources, see the Performance topic page.
- For tools issues and resources, see the Tools topic pages.

Posted: 2004-11-08
