How Profilers Can Aid Software Development

D A V I D R Ö N N Q V I S T

KTH Computer Science
and Communication

Bachelor of Science Thesis
Stockholm, Sweden 2010

How Profilers Can Aid Software Development

D A V I D R Ö N N Q V I S T

Bachelor's Thesis in Computer Science (15 ECTS credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2010
Supervisor at CSC was Mads Dam
Examiner was Mads Dam

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2010/ronnqvist_david_K10060.pdf

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden

URL: www.kth.se/csc

Abstract

It is difficult to write fast, bug free applications. A small portion of the code is run much of the time and it is hard to know where it is. Unexpected behavior caused by bugs can often show up in equally unexpected places.

There are a lot of tools available to help the developers find bugs and bottlenecks so that they can spend their time improving the right parts of their applications.

I investigated the ease of use for profilers and found the results well worth the learning time. Using simple methods you could quickly find many common errors or bottlenecks that would otherwise be hard to find.

Referat

Hur profilers kan hjälpa mjukvaruutveckling

Det är svårt att skriva snabba, buggfria program. En liten del av koden körs en stor del av tiden och det är svårt att veta var. Oväntat beteende från buggar uppträder ofta på lika oväntade ställen.

Till utvecklarnas hjälp finns en mängd verktyg för att hitta olika buggar och flaskhalsar så att de kan spendera sin tid på att förbättra rätt delar av sina program.

Jag undersökte hur lätta profilers var att använda och fann att resultaten var väl värda tiden att lära sig verktygen. Med enkla metoder kunde man snabbt hitta många vanliga fel och flaskhalsar som annars skulle vara svåra att hitta.

Contents

Chapter 1

Introduction

The "ninety-ninety rule"[1] attributed to Tom Cargil is an aphorism in software engineering that states:

> "The first 90% of the code accounts for the first 90% of the development time. The remaining 10% of the code accounts for the other 90% of the development time"[2].

It illustrates the difficulty of anticipating the hard parts of a software development project. Over the years it has been referred to in many variations such as: "90% of the time is spent in executing 10% of the code"[1], illustrating that a small part of the code is very performance critical.

These variations may not represent real scientific data but Barry Boehm measures that 20% of a programs code consumes 80% of its execution time[2]. Further Donald E. Knuth found that more than 50% of the execution time is accounted for by less than 4% of a programs code[3]. Optimising these parts of the code is crucial for the application performance, optimising other parts won't do much good.

Donald E. Knuth describes the strong negative effect of trying to improve performance in non-vital parts of the code saying that it leads to more work when considering debugging and maintaining the code, or as he puts it[4]:

> "premature optimisation is the root of all evil"

Therefore optimisation should only be done after the performance-critical parts have been identified.

Andrew J. Ko & Brad A. Myers has concluded[5, 6] that when guessing what caused an unexpected behavior or a bug the developers were wrong almost 90% of the time[7]. It can be said that writing fast and reliable code is very difficult. When trying to improve speed or fix bugs, much time will be wasted working on the wrong parts of the code if no tools are used to help identify the right parts.

[1] also known as the "ninety-ten rule"

[2] it is not a typo that the development time sums up to 180%. It is meant to show that writing code is harder than expected

1.1 Problem formulation

Are profilers easy enough to use for finding bugs and bottlenecks, so that they can help developers write better code?

1.2 Structure

This thesis describes the basics of analysing software for performance and correctness, describing a few different approaches and discussing their pros and cons. A more in depth study of profiling and instrumenting was made to detail how these tools can make it easier to find bugs and bottlenecks in applications.

The ease of using such tools to improve an applications quality and performance was investigated. The basic functionalities available in most profilers and "instruments" respectively, was used during development of a smaller application. The gain in performance and bugs found as well as the ease of finding them was documented to be discussed.

Part I

Background

Chapter 2

Definitions

2.1 Defining performance

To be able to improve the performance of an application it is critical to define performance. An engineers answer would be that performance is calculative speed. This is a good answer for applications working with simulations, 3D-rendering or other processor intensive purposes. These kinds of applications all have long continuous calculations and it is easy to define and measure performance by measuring the time of these calculations.

For algorithms there are ways to calculate and compare the asymptotic speed, called the computational complexity[16]. Complexity is a way of characterising how the amount of computations scales to the size of the data. Most common is to look at average case and worst case. It is a measurement for growth rate, so constant factors that can be crucial to the real-life speed won't be accounted for. Algorithms can also be compared by how much space is needed to perform the computations, this is called space complexity. It is important to note that complexity only describes the asymptotic speed or memory use of an algorithm and therefore is a very generic measurement.

In the end it is likely that an end user of an application will be the judge of its performance. In that case it is highly unlikely that the user will use a stopwatch or another objective way of defining performance. In those cases perceived performance may matter more than actual speed since the user is being subjective. What the user perceives as performance can be many things. Feedback and response time are often important to give a performant appearance.

In a situation where calculations are done over a large enough data set, the "faster" approach would be to do all calculations and then redraw the screen but a more user friendly approach would be to do "unnecessary" calculations to redraw the application during these calculations, updating values as they are calculated. Not only will the user get feedback that the application is running but it may even be that the values update so often that the user gets impressed with the speed of the calculations.

Splash screens, an image shown while the application is loaded, are often used to let the user know that the application is on it's way. This is a good way to give feedback and make the system feel responsive to the user. Large applications often use splash screens since they take a longer time to load into memory. Even with a splash screen the application will eventually be perceived as slow as the seconds go by.

Animating splash screens can be used to give the appearance that the application is ready. A simple but effective animation is to scale up an image of the application in a way that feels like a natural analogy to "opening" the application. When the interface of the application is loaded it takes the place of the image. In an application that launches in about 1 second, using a 0.5 second animation starting half way through the launch could make it seem like the application launched twice as fast.

Performance doesn't need to be just one thing, but can be a combination of all things mentioned above. How these things are weighted will vary from case to case but it could be said that improving on the current limiting factor will improve the performance no matter how it is defined.

This thesis is about tools that measure calculative speed and memory efficiency. A suitable definition for performant application would be applications that initialize quickly and run fast with low memory consumption. Splash screens are only about appearance.

2.2 Defining program correctness

A computer would consider an application that runs without errors and follows the code as running correctly. For many purposes this is very valid definition but it fails to account for semantic errors in the code. The developer could have meant for the application to do something but wrote valid code that did something else. It is important for both the developer and the user that the application does what it is intended to do. So a program that runs without errors and does what is intended to do is a good definition of a program running correctly.

The intensions of the developer will always be hidden from both the computer and any analysis tool. Therefore a program analysis tool cannot directly look at this kind of correctness.

Chapter 3

Modern analysis tools

It has been proven that no algorithm can determine if a program runs without errors for all programs with a small and finite space of states[8]. This can be done by reducing to the halting problem, stated and proven unsolvable by Alan Turing[9]. Although unsolvable, an approximate answer may find enough errors to make the program run long enough without errors to give it an error free appearance. This is a good goal for any tool analysing program correctness. Software can either be analysed by looking at the code (static analysis) or by running the application (dynamic analysis).

3.1 Static analysis

Static analysers parse the code and tries to find errors in it. This works really well for many common kinds of bugs as the analyser can track a large amount of logic branching in mere seconds. Therefore static analysis can offer almost effortless detection of some critical errors that can be difficult to find otherwise. Using a static analyser often can really help find errors as soon as they are introduced making the cost of fixing them as small as possible.

The downside is that static analysers will only find the special kinds of errors they are designed to look for. Dynamic analysis on the other hand will only follow the specific branch of code that the application is running.

3.2 Dynamic analysis

Dynamic analysers collect and represent application run-time data to be analysed mainly by the developer. The kinds of data being gathered depends on the capabilities of the tools and what data the developer is looking for. Getting large amounts of data is easy, gathering one kind of data every millisecond on a quad-core machine for a second would generate 4000 data points. Therefore an important part of a dynamic analyser is to visualise and help make sense of that data.

There are two goals in software analysis, program optimisation and program correctness, and most tools specialise in one of these. Certain aspects of optimisation and correctness, such as threading, IO or graphics often have their own set of tools since they have special properties to analyse. E.g. when looking at performance of a multithreaded application the tool needs to be able to trace and visualise not only the work done by each thread but how they work together and if one thread is locking another thread.

The two main approaches to get run-time data from an application are sampling and instrumenting. Sampling is the statistical procedure of gathering samples. When it is time to gather data, the sampling profiler halts the application using operation system interrupts and collects its data. Instrumenting (or instrumentation) is the procedure of modifying an application by inserting code that outputs analysis data. The ATOM platform[17] has been a big influence in application instrumentation[18].

3.3 Profiling

Profilers, sometime referred to as performance analysers, are dynamic analysis tools that gather data from an executing program. Although tracing and instrumenting is also considered profiling they will be discussed later in Section 3.4 on page 10.

3.3.1 Statistical profiling

A statistical profiler probes the application at regular intervals and gathers some data of its current state[1]. It may be argued that this sampled result is faulty because it does not measure between the sampling points. Indeed, worst case results with a large sample interval and a short run-time can give you faulty results with two kinds of problems. Routines may be completely missed between sampling points or may be interpreted as a longer routine if running when sampling occurs but not in between, as illustrated in Figure 5.2 on page 20.

In this worst case, the sample is truly a faulty result. However, in practice these faults average out over very large amounts of samples. Since the performance critical routines run much of the time they should have even better coverage and get a good measurement. The expected error of a routine is actually about the square root of the sampling period[19]. E.g. if a routine runs for a second and is being sampled every 0.01 seconds, the expected error is 0.1 s. If it is needed the sampling can sometimes be run more then once, accumulating the results across samples.

The strength of time sampling is that it can give a good overall understanding of an applications performance without skewing the results with high overhead, allowing the application to run at nearly full speed. With dedicated hardware, such as the PCSAMPLE register on some MIPS processors[11], the time cost can be further reduced resulting in an even lower overhead.

[1] It is common to use the applications instruction pointer to do this.

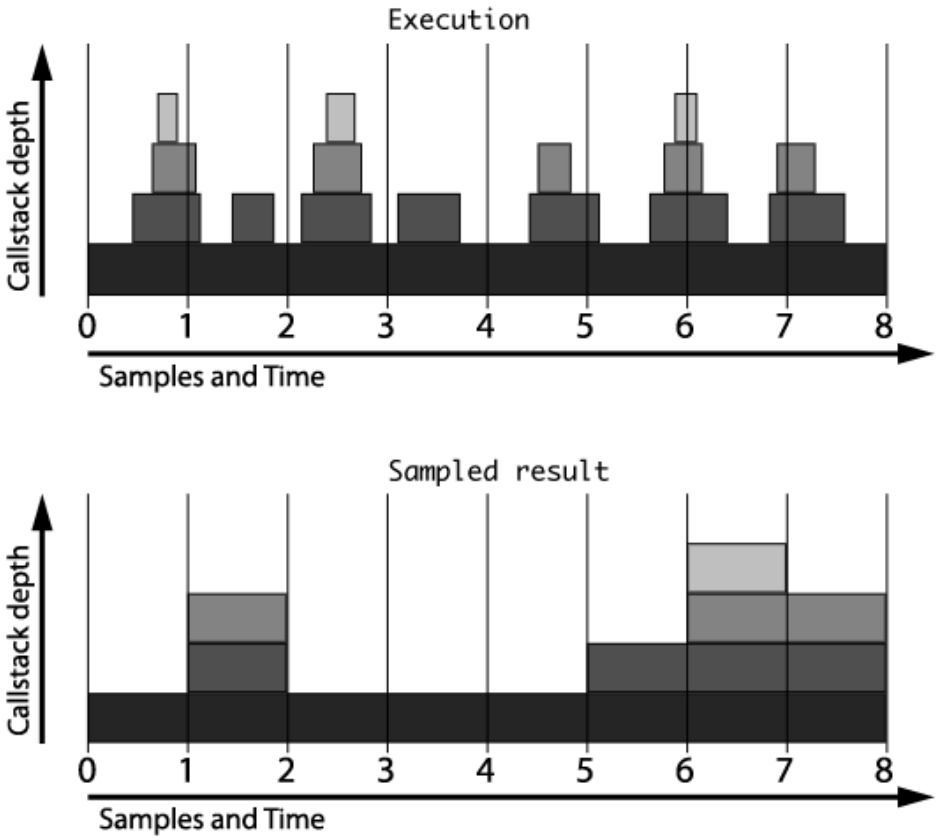

Figure 3.1. The real execution and its sampled interpretation of the call stack.

3.3.2 Call graph profiling

A call graph profiler is used to generate call graphs, directed graphs describing relationships between routines in an application. The call graph are used for better human understanding of the flow of data and work within the application. Call graphs can either be represented as an actual graph or as a call tree. An example call graph can be seen in Figure 3.2 on page 10 where the routines call their subroutines.

If the profiled application spends much of its time in third party libraries or other code that developer can't make changes in, it can be hard to improve the performance. A call graph can be used to understand why and when these libraries are being called so that changes can be made in the code that can be modified. E.g. if some library is used to fetch data from some source and the application uses this method often, it may be better to fetch data in larger chunks. A call graph could help spot these situations.

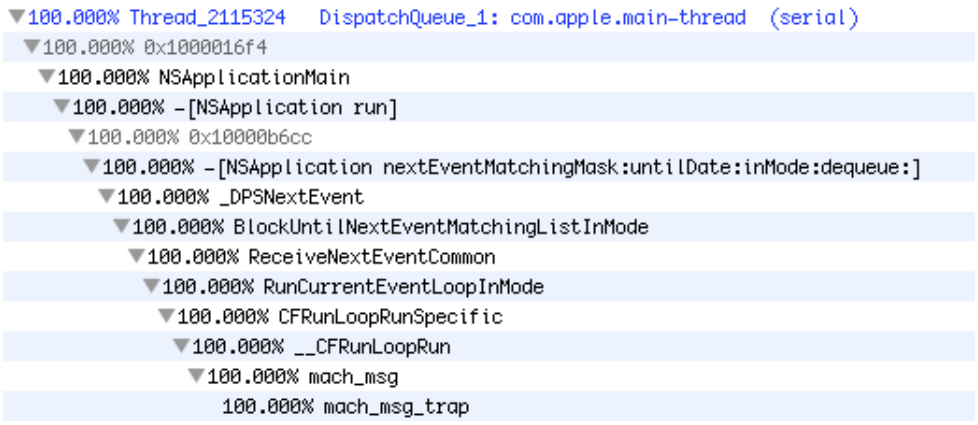

Figure 3.2. The sample call graph of a single thread.

3.3.3 Example tools

Some examples of common profilers are: AMDs CodeAnalyst and Intels VTune for Windows and Linux platforms as well as Shark for Mac OS X. They all support all the sampling methods mentioned above as well as sampling multithreaded applications and viewing the sampling result line by line in the source code.

3.4 Instrumenting and tracing

When instrumenting an application, analysis code is inserted into the application, either automatically at compile-time or manually with the rest of the code, the later is sometimes referred to as tracing. Instrumentation can be used to get exact data out of an application in many ways. E.g. to see the exact number of times a routine gets called a counter could be inserted at the entry of that routine. By inserting code in the exit of that routine, the execution time of each pass through the routine could also be calculated.

Instrumentation could also be much more thorough and flexible: look at input, output, call graphs, stacks and work with their own variables. This process can have a significant overhead if the analysis code is executed much of the time. This can happen if the instrumented code is executed very often or the analysis takes much time. Therefore instrumentation is more aimed towards a deeper level of investigation and analysis.

Say the application has been profiled using a statistical profiler and it has been determined that a lot of time is being spent reading and writing to disc. The application could then be instrumented to print out what files are being accessed, how often and what routines called the read/write. This will make it easier to determine the cause of the performance issue and solve it.

Instrumentation is also a common method for analysing program correctness.

In a survey[2] by Glenn R. Luecke et al.[20] the best systems for finding run-time errors in C and C++ programs, Insure++ and Purify, used instrumentation. Both of these systems slowed down the execution around 25 times compared to normal gcc compiled executables. This slowdown doesn't matter that much since the focus is on correctness. It is important to note that applications can be instrumented without a significant performance loss, especially when only parts of the application is instrumented.

3.4.1 Intrusiveness

Instrumenting an application can be intrusive, meaning that the instrumentation is making a significant amount changes to the application. Some level of intrusiveness is required to be able to get data. Being very intrusive will affect the application in several ways. The performance is lowered[3] since extra code is added and since the compiler cannot optimise the code. There may also be changes in the memory reference patterns[22]. All these effects are bad so the instrumentation should be as non-intrusive as possible. How intrusive the instrumentation needs to be depends on what data is being gathered. Instrumenting certain parts of applications, such as entries and exits of routines, can be less intrusive. Instrumenting these parts with code that doesn't modify the application can be called "non-intrusive" even though they are only low-intrusive.

3.4.2 Production systems

One important use of instrumentation and tracing is the ability to analyse production systems, that are already shipped and running. This can be done by inserting deactivated trace code into the application and activate it from an outside source. Another approach is to mark safe places for instrumentation and then insert trace code via JIT-compilation. This can be valuable when debugging a system, like a server, that can not be taken down just for testing[4].

This can also be useful if a hard-to-replicate bug appears. By turning on or adding trace code, information about why the bug is can be retrieved without restarting the application and having to try to replicate the conditions for that bug.

3.4.3 Example tools

Insure++ and Purify, as mentioned above are two examples of instrumentation tools, aimed at program correctness, available for Windows, Linux and Solaris platforms. Valgrind is a free alternative available on Linux and Mac OS X that also uses instrumentation.

[2]The purpose of the survey was to compare the capabilities and ease-of-use of commercial and non-commercial systems.

[3]The execution time may even exceed three orders of magnitude[22].

[4]DTrace, described in Section 5.2.1 on page 17, was made for this purpose[21].

Pin, available for Windows and Linux, and DTrace, available on Solaris and Mac OS X, are two examples of tools that allows dynamic insertion of code into an application via JIT-compilation.

Part II

Investigation

Chapter 4

Purpose

4.1 Conclusions from the literature

There are many tools available to help find many different kinds of flaws in applications. The introduction illustrated the difficulty of finding these flaws and errors by hand and pointed to the need for these tools. Necessity doesn't mean ease of use and we know from experience that small enough applications are manageable even by hand.

Managing, optimising and maintaining an application are all important parts of the software development process. Much work seems focused on what information can be provided in large applications by an experienced analyser. This is of course very important but so is the ability to scale a powerful tool down to moderate sized and even small sized applications.

4.2 Investigation purpose

The purpose of this investigation was to see how difficult these tools were to learn and use and how difficult it was to acquire good results.

4.3 Methodology

A statistical profiler, Shark, and an instrumenting approach, DTrace and "Instruments", was used in the investigation. To learn how to use these tools they were first used, in a smaller pre-investigation, to analyse a smaller application and look at intrusiveness in a third party application.

Further a bigger application was developed, during a few weeks, with rigorous use the same tools during and after development. The focus was laid on the functionality that is available in most of the tools of their respective kind to keep focus away from the specifics of the tools being used.

Errors or bottlenecks that where discovered where documented along with work that lead to the discovery and the gain from solving it.

Chapter 5

Tools

5.1 Shark

Shark is a low overhead statistical profiler by Apple Inc[10]. Its purpose is to help understand and optimise performance. Shark can profile just the application, the entire system or even sample on hardware and software performance events such as cache misses. In some common performance pitfalls Shark can give you advice of how to improve performance[1].

By default Shark samples each running thread every millisecond with a typical overhead on the order of 20 microseconds. This low overhead can be accomplished since the sample collection takes place in the kernel and is based on hardware interrupts[10].

Besides time sampling, Shark can also use performance events to trigger sampling, generate exact profiles for memory allocations and use static analysis on infrequently used code paths. These functionalities target more specific problems and won't be in the investigation.

If much time is spent in libraries or other code that the developer can't modify, that time cost can be assigned to the callers to see where the developer can improve its usage of those libraries. This can be done since the call stack is collected during samples.

5.2 DTrace and Instruments

5.2.1 DTrace

DTrace is a dynamic tracing framework, introduced by Sun Microsystems in 2005 released under the free Common Development and Distribution License (CDDL). It was originally developed for Solaris but has been ported to other Unix-like systems such as Mac OS X (since 10.5 "Leopard")[15].

[1] No such advise was given for my code.

DTrace is used to safely analyse live production systems and provide concise answers to more specific questions. The DTrace framework provides instrumentation points, called *probes*. Each probe is made available by a *provider*, instruments a *function* in a *module* and has a *name*. These four attributes servers a unique identifier for each probe, in the format:

```
provider:module:function:name.
```

A probe is triggered, *fires*, by specific behavior causing it to take some action, written in the D scripting language[12]. Actions are used to record data. DTrace actions *can* be used to change system state in a precisely described way but these actions are not allowed by default. DTrace supports lock-less thread-safe[2] aggregations and variables. The general form for the aggregates are:

```
@name[ keys ] = aggfunc( args );
```

with aggregating functions such as count, sum, avg, min, max[13].

A D Program consists of one or more *clauses*, describing the instrumentation. Each clause targets one or more probes using their identifiers with blanks as everything and question marks as wildcard, e.g.

```
syscall:::entry
```

targets the entries of all system calls. The probe triggering can then optionally be narrowed down by a predicate that works like and if statement, e.g.

```
/execname == "MyApplication"/
```

would prevent the probe from firing unless the application was ”MyApplication”. After the trigger conditions the action statements are written within curly braces[13].

The power of DTrace is that very specific data can be collected from already running applications with great ease. Running one-liner aggregation script over some key will print a histogram for the keys. Disabled probes has zero effect on the performance of the system. This is important since there are lots of probes in a system, my system[3] has over 86000 probes.

5.2.2 Instruments

Instruments is a debugging and performance tool for the Mac OS X platform. The Instruments application uses special tools, called ”instruments”, to collect different data over time. Unlike many other analysis tools, Instruments can run different instruments at the same time, allowing the developer to trace and viewing different data side by side. As many instruments as needed can be added to the

[2]This means that these aggregations can be used in multi-threaded applications without inserting locks

[3]The development was done on an iMac running Mac OS X 10.6.3

same trace document but it is recommended to use less than ten to not skew the performance[14].

The developer can run through the application once and record different things like CPU usage, memory allocations and user interaction and see that when a certain button is clicked the program runs slow because application allocated many objects. The developer can then make changes to that code and see the differences between several runs side by side. Since the user interactions where recorded, the same interactions can be replayed during the other recordings[14]. This is useful when replicating bugs but won't be used in the investigation.

Many of the built in instruments use DTrace to collect their data and custom D programs can be added to view the results side by side with the rest of the instruments[14].

5.3 Examples

For a simple example, say we want to know every time our application opens a file. The code for a D program doing this would be:

```
syscall::open:entry
/ execname == "MyApplication" / {
    printf("%s", copyinstr(arg0));
}
```

and the same D program written as a DTrace instrument for Instruments would look like:

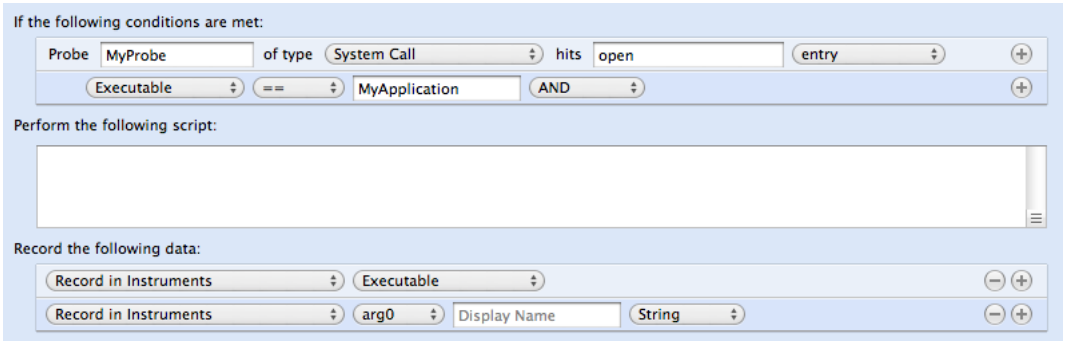

Figure 5.1. The first DTrace example, that prints out when an application opens a file, written as a DTrace instrument

For a more advanced example, say we want to know the average execution time of a function within our specific application, this differs from our previous example where the `open` function was in the system. This time our D program would need 2 probes and we would need to use thread scoped variables using `self->`. The code to do this could be:

```
pid1234::some_function:entry {
    self->entry_time = timestamp;
}

pid1234::some_function:return {
    @duration = avg(timestamp - self->entry_time);
}
```

In the example above our application was process ”1234” and the function of interest was `some_function`. We used an aggregate for the durations since they are thread-safe and print out when we stop our D program. To use the same D Program in Instruments it makes more sense to only calculate the duration as a local variable and calculating the average in post processing. A DTrace instrument for this could look like:

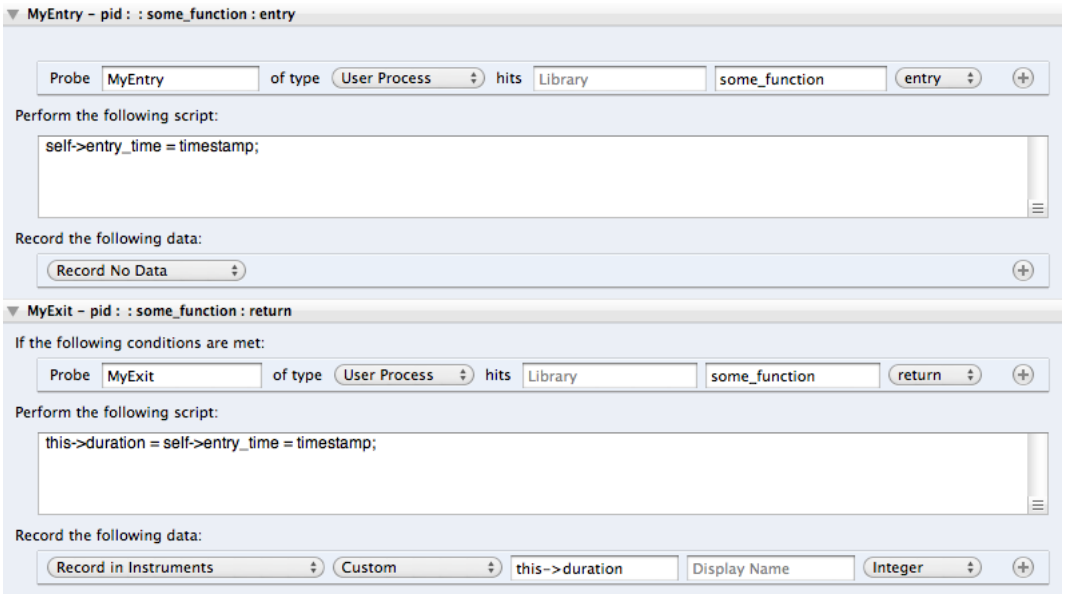

Figure 5.2. The second DTrace example, that calculates execution time of a function, written as a DTrace instrument

Part III

Results

Chapter 6

Pre-investigation

6.1 Slow graphics

I wrote a small, unoptimised test application that created a large number of shapes with different properties, shapes and shading and moved them around. I added shapes in different views from mixed graphics libraries, like OpenGL and Quartz 2D, until the application appeared slow. I wanted to sample the application to see what made it run slow and measure the improvements I could make.

The first run showed that 65% execution time was spent calculating normal distributions for a Gaussian blur. Assigning the time cost to the callers only showed that all execution time was in the drawing method, however I could figure out that the blur was used to create drop shadows on some of the shapes.

I looked at my code for drawing shadows and realised that the shadows was recalculated on the CPU for every frame. I could either use the graphics card and shaders, like GLSL, to calculate the shadows or reuse them. I choose to calculate them for the first frame and cache and reuse them at the other frames. This was enough to make the application run smooth again and significantly reduce CPU usage.

The total CPU usage went down a factor 10 by not recalculating the drop shadows between frames. Any further optimisation seemed unnecessary.

6.2 Looking at intrusiveness

To look at the intrusiveness of instrumentation I took a third party application[1] and instrumented it while it rendered a 10 second long movie clip. Without any instrumentation the process took 43.2 seconds on average to complete. This was compared against the same render during a time sample and a leak finding instrumentation, alone and during a time sample. The render times are shown in Table 6.1 on page 24. There was a moderate slowdown in rendering times when instru-

[1]The application was a trial version of a screen casting application called ScreenFlow

menting the application. Running several instruments side by side increased the render time to a few minutes.

The profiler results where indistinguishable from each other. This was expected since the instrumentation should not alter the execution of the application.

Table 6.1. Render time for different levels of instrumentation

	no instr.	time profile	leak instr.	time profiled leak instr.
1st run	42.2 s	52.3 s	83.2 s	75.4 s
2nd run	43.3 s	52.4 s	77.6 s	75.5 s
3rd run	43.2 s	48.0 s	77.9 s	85.4 s
4th run	43.1 s	48.8 s	79.0 s	88.3 s
5th run	44.0 s	51.4 s	74.3 s	82.6 s
average	43.2 s	50.6 s	78.4 s	81.4 s
times slower	×1	×1.17	×1.81	×1.88

Chapter 7

Main investigation

7.1 Description

A larger application was developed for the main investigation. The application generated a track in 2D and then generated random ”cars” and simulated their run through the track[1]. At each step the forces where calculated and balanced between the four parts of the car by Gauss-eliminating a small numerical equation system. Graphics was done using OpenGL and the code was written is C and Objective-C[2].

7.2 Investigation experiences

7.2.1 Memory management

Doing the regular analysis I was sometimes informed of memory related errors, leaks and over released object. The included "leaks" instrument could tell me what object was leaked (or over released) and where. Fixing these errors were almost effortless.

An example leak is shown below, in code, where an object was leaked because both the creation and adding it to the array retained the object. Autoreleasing the object after adding it to the array balanced the retain count.

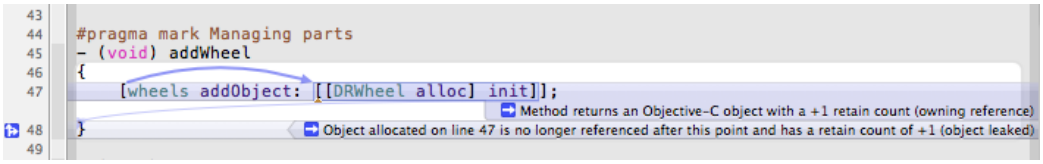

Figure 7.1. The code where an object was leaked and an explanation of why it leaked.

[1] A screenshot of the application is shown in Figure 7.3 on page 27
[2] See Appendix A

7.2.2 Slow collision detection

When running a regular time profile I noticed that 75% of execution time was spent doing collision detection, something that I hadn't anticipated. The application was not running slow but this was an alarmingly high value.

When I looked at what parts of the collision detection accounted for most of the run-time I noticed that lines of code that should run really fast (like simple addition) was taking up a lot of time, so they had to run very often. All these lines

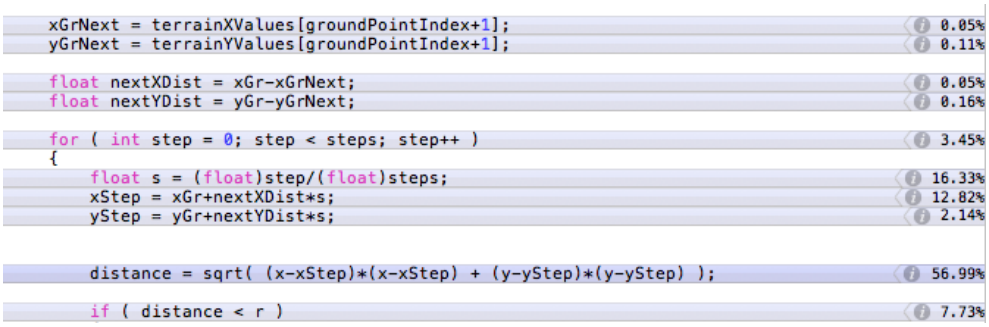

Figure 7.2. The time profile results line by line for the unoptimised collision detection.

had in common that they where in the same loop, so I had to optimise the number of times this loop was being run. This was because the collision detection was doing detailed collision detection, checking if a line to the closest points intersected wheels of the car, for all points.

By doing a binary search to find all pairs of points that could intersect the wheel and only do detailed collision detection for these points the time spend doing collision detection could be reduced to less than 2%.

Even though the application didn't appear as slow the CPU usage was lowered by a factor 6. I could possibly do more optimisations but during the development phase this is more then enough to solve a critical performance problem.

This was early in development and as the detection was improved to account for multiple collisions the time spend increased to about 4−5%.

7.2.3 Drawing circles

I encountered a problem where the application would sometimes slow down to an almost frozen state just after starting it. A time profile showed that almost all time was spend in my own code to draw circles. Instrumenting the input to that function proved that the radius was 0 when running slow. I had tried to prematurely optimise the code to use a lower amount of vertices in the circle for smaller radii. Somehow a 0 radius caused big problems.

This simple instrumentation was very helpful in pointing out the cause of the slow down. The real problem was that the simulation ran on different thread and the radius was not initialised immediately, so the first drawing could be done before

the radius was set. Simply checking if the radius was 0 and return without drawing was a good enough solution.

7.2.4 Text in OpenGL and screen flickering

At one point the application started flickering, so I profiled it to see what was slowing it down. I knew that the OpenGL code was completely unoptimised and written in a vary naive way, doing much off-screen rendering so my guess was that I needed to start making improvements there. The time profile result shocked me. 54% was spend drawing text on the screen. Looking at Figure 7.3, there is not much text on the screen and it would have taken me a long time figure this out by myself.

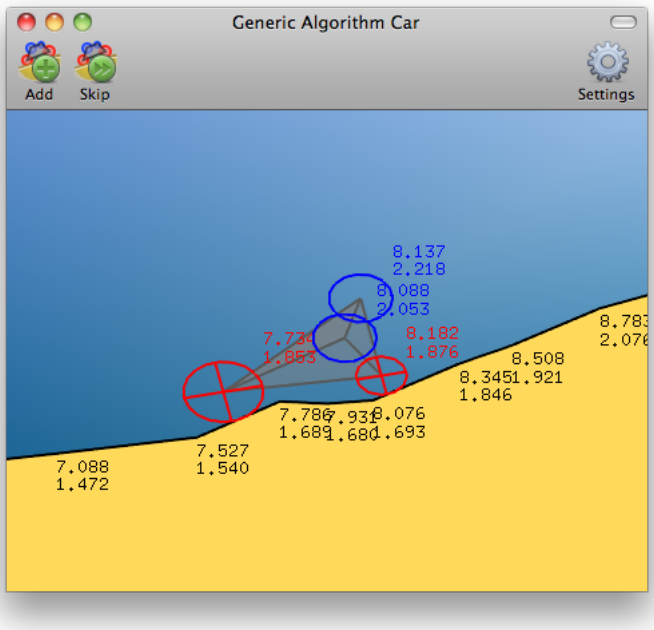

Figure 7.3. A screenshot of how the application looked when the text proved to be a bottleneck.

The text function was part of the OpenGL Utility Toolkit (GLUT) so the only improvement available was to call that function less.

The screen however was still flickering and it was doing so even without the text. The problem actually was slow OpenGL code that doesn't show up when time sampling or instrumenting the application because the code runs on the GPU. In fact GPUs differ a lot from CPUs when it comes to debugging and performance analysis. There are often special tools made for profiling code on GPUs but this doesn't lie within the scope of this thesis and won't be described here.

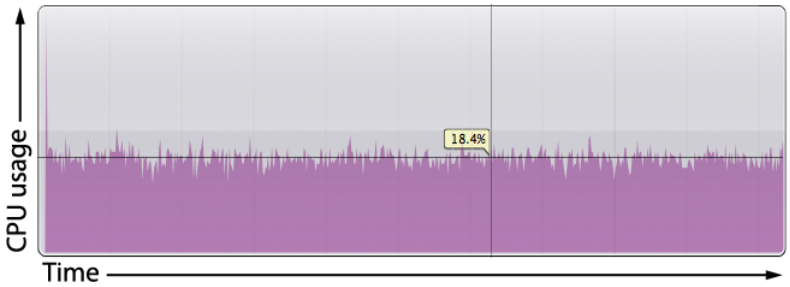

Figure 7.4. Graphs of the CPU usage before improving the text handling, highlighting the 18.4% line. The graph is linear showing $0\% - 35\%$.

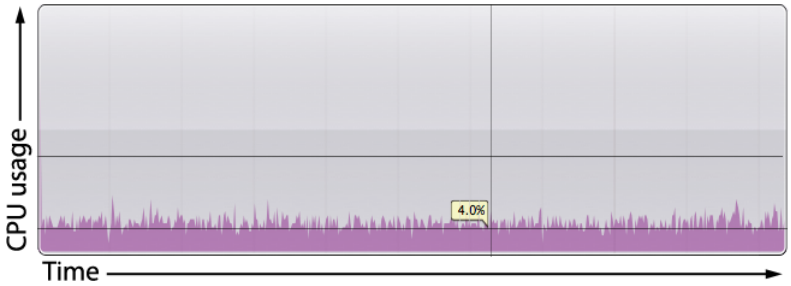

Figure 7.5. Graphs of the CPU usage after improvements, highlighting the 4.0% line and the 18.4% line (from Figure 7.4). The graph is linear showing $0\% - 35\%$.

7.2.5 Logic errors

During development there was some difficult errors where the application didn't do as it was supposed to. One of these were a problem with the collision detection where it missed collisions in the front of the wheel if the ground was ascending. Since the purpose was to use profiling and instrumenting to find the problem I tried to use these tools to find the cause of these bugs but I more than once had to fall back to drawing the basic steps of my collision detection on paper and think it through to find the cause of the problem. In this example the problem was that the collision detection returned the collision as soon as it found an intersecting point.

For this and other such occasions I often had to step trough the code using the debugger (GDB) and make changes to the code back and forth. In some cases, like this, I even had to take a step back and use pen and paper to fully understand why the application misbehaved.

Chapter 8

Post-development optimisation

At this point, the developed application ran fast enough for all it's code to only account for a couple of percent of the execution time. The rest was spent sleeping the thread. The graphics had a bad performance so it was improved enough to run smooth before the rest of the optimisations[1].

8.1 Goal

To create a purpose of optimising the application, the timed sleep was removed and the simulation step size was reduced to a small enough value to cause the simulation to run at a normal speed. This made it difficult to come up with a natural goal for the optimisation. I chose to aim at a 25% increase in speed, directly proportional to a 25% lowered CPU usage.

8.2 Results

The first step was to establish the base line for comparison by profiling the release version, with all the compiler optimisations turned on. The release version used 83.2% of one CPU on average[2]. This base line profile also pointed out that the collision detection accounted for approximately 65% of the total execution time. The second heaviest part of the application was the calculation and balancing of the forces that accounted for less than 6%.

From this data I choose to start make improvements in the collision detection and only modify the other parts if I couldn't reach my goal from the collision detection alone. To be able to lower the overall CPU usage by 25% I had to lower the time spent doing collision detection by approximately 40%, i.e. lower it from 65% to less than 40%.

Looking at the execution time line by line I could see that calculating the square root of each distance in the detailed collision detection was the heaviest lines by

[1] This lies outside the scope of this thesis, as mentioned in the end of Section 7.2.4 on page 27.
[2] The debug version used 89.6% of one CPU on average.

itself. Since it was only used to compare with the radius of the part it was much cheaper to square the radius once in the beginning and not do the 50-150 square roots. This was the first improvement, lowering the execution time to approximately 52% of the overall execution time.

Another profile revealed no obvious hot spots, but pointed to the detailed collision detection as a whole. This meant that finding pairs of points that *could* collide with the part was efficient but taking small steps along the line between those points was not. I started thinking about different algorithms to find the closest collision and realised that there where two different cases of collision detection to solve.

If the part was a wheel, then I needed to know the point along the line closest to the center of the circle. On the other hand if the part was not a wheel, then the car should brake so I only a needed to know if the line intersected the circle. Braking the detection up into these two pieces brought it to about 37.5% of the total execution time.

This had actually reached my goal but I still knew a simple improvement that could be made. That was to only walk along the line between the points while the distance to the center decreased, starting from the closer of the two points. Other than that some variables where moved to the top of the method to stop them from being recalculated unnecessary. Now the collision detection accounted for about 32% of the total execution time and the post development optimisation was complete.

Chapter 9

Discussion

9.1 Performance

My experience during this project was that the tools where easy to use and quick for common tasks. To know what parts of the code was running the most, I only had to start the profiler and attach it to the application and let it run for about 30 seconds. This meant that a minute every now and then was enough to ensure that the performance was kept under control during development.

The time profile was very easy to understand for code that was well re-factored into small enough pieces. The names of the *heavy*[1] methods where often enough to know what lines of code was relevant to look at. For the cases where the cause wasn't obvious the ability to view percentage of execution time within the code could often help show a pattern, especially for loops.

It is worth pointing out that time profiles can be a false positive if the application runs code on hardware that is not being profiled, like a GPU.

The post development optimisations required a little more thinking because there was no obvious mistakes slowing down the code. The statistical profiler still gave the same, good pointers of where to look and it was easy to make decisions about where to change the code. The line by line view was great for finding patterns within a method and even more important, show the low percentage parts of the method that was unnecessary to look at.

9.2 Correctness

The tools made it really easy to both find and fix smaller, common errors. Even though the effort was low, many of these errors could have been found even faster and with less effort using a static analyser. For the bigger and less common errors instrumenting the application was really useful when knowing what to look for, and even more if knowing where to look. Otherwise the instrumentation tried to compete with the debugger. Instrumentation and debuggers don't work together,

[1] Time profilers call methods that run much of the time for *heavy* or *hot*

at least not well. Debuggers are built to debug errors like these and they where better integrated with the IDE and was of more use in these situations. However instrumentation was a good alternative to adding log-statements in the code.

9.3 Cost of learning

Both the concept behind time profiling and how to use it was really straightforward. Interpreting the results was slightly harder but very easy to get accustomed to. I used a simple workflow of looking at the heaviest methods and assigning the cost of all methods that were outside of my code-base to their callers. This was a quick way to get a rough pointer of where to look for improvements and didn't require much learning. This approach worked well when there are obvious "hot spots".

The various approaches of instrumentation was harder to grasp. However it wasn't necessary to fully understand instrumentation to be able to start using it. It still took some time of reading user guides and following tutorials to know when instrumentation could and should be used. Compared to inserting debug code, instrumentation was a little bit hard to get used to but easier to clean up.

Using the built in "instruments" was straightforward for common things like checking for memory leaks. Considering the time of learning and using them in my investigation, I found it well worth using them when looking for these common errors. Finding many of these errors would have taken much more time by hand and most likely all would not have been found.

9.4 Conclusions

Some kind of tool should be used when developing an application, both for improving the performance and the correctness. Profiling was easy to start using and start learning and that there was a gain in both understanding of the code and performance of the application by doing so.

There are many different approaches to find common errors and some tool, no matter the approach, should be used to find these errors. When looking at application specific errors, especially semantic errors. Other tools have a hard time competing with a real, integrated debugger unless you know what you are looking for.

All in all I was surprised by the ease of getting some results and intrigued by the ability to get good results with the proper knowledge of the tools. I believe that these kinds of tools will become even easier to use in the future and that they will be able to give more precise information. This will hopefully increase their popularity and make it trivial to find even more common errors.

Appendix A

Objective-C

A.1 Language

Objective-C is a strict superset of C that adds object orientation in a SmallTalk like syntax, with square brackets, e.g. `[object method];`. Conventions recommend descriptive method names and the language supports arguments within the method name, e.g. `[myColor changeColorToRed:5.0 green:2.0 blue:6.0];`. Objective-C 2.0 also allows "dot" syntax for properties, e.g. `object.property;`.

The language itself is available for both Linux and Windows but is most popular for developing for Mac OS X.

A.2 Memory management

Objective-C and the Foundation-framework uses a retain and release based memory management system. All object that are allocated and initialised have a retain count of 1. When an object needs another object it retains it, by calling `[object reatin];`, thus increasing that objects retain count by 1. If several objects need the same object they all retain that object to keep it from going away, incrementing the retain count each time.

When one of them no longer needs to hold on to the object, it releases it by calling `[object release];`, thus decrementing the retain count by 1. When the retain count gets lowered to 0, no other object needs that object any more and it is deallocated.

Conventions determine whether a method should return a retained object or not. E.g. methods starting with `init` or `copy` should always return retained objects. It is not safe to return an object without retaining it, since it can get deallocated. For these cases objects returned are *autoreleased* by calling `return [object autorelease];`. Autoreleased objects will be have an increased retain count for a short time period, allowing the caller of the method to either use the object in an calculation and letting it get released or retain it to prevent it from being deallocated.

A.2.1 Instrumenting

By instrumenting the `retain` and `release` methods, an objects retain count can be traced. This information is very helpful when determining the cause of a leak. By saving what object retained and released the leaked object it becomes easy to find the one/ones that doesn't balance their retains and releases.

Bibliography

[1] Satish C. Gupta: Need for speed - Eliminating performance bottlenecks, IBM Software Group (2005)

[2] Barry W. Boehm: Improving Software Productivity in *IEEE Computer* 20, Vol. 9, pp. 43-57 (1987)

[3] Donald E. Knuth: An Empirical Study of FORTRAN Programs, in *Software - Practice and Experience 1*, pp. 105-133 (1971)

[4] Donald E. Knuth: Structured Programming with go to Statements, in *Computing Surveys*, Vol. 6, No. 4, pp. 261-301 (1974)

[5] Andrew J. Ko, R. DeLine & G. Venolia: Information needs in collocated software development teams, in *International Conference on Software Engineering*, Minneapolis, pp. 344-353. (2007)

[6] Andrew J. Ko & Brad A. Myers: Designing the Whyline: a debugging interface for asking questions about program failures, in *ACM Conference on Human Factors in Computing Systems*, Vienna, Austria, pp. 151-158. (2004)

[7] Andrew J. Ko & Brad A. Myers: Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior, Human-Computer Interaction Institute School of Computer Science, Carnegie Mellon University (2008)

[8] William Landi: Undecidability of Static Analysis, in *ACM Letters on Programming Languages and Systems*, Vol. 1, No. 4, pp 323-337. (1992)

[9] Alan Turing: On computable numbers, with an application to the Entscheidungsproblem, in *Proceedings of the London Mathematical Society*, Series 2, 42, pp 230–265 (1936)

[10] Apple Inc.: Shark User Guide (http://developer.apple.com/mac/library/documentation/DeveloperTools/Conceptual/SharkUserGuide/Introduction/Introduction.html) revision from 14 April 2008

[11] MIPS Technologies Inc.: EJTAG Specification, Revision 3.10, pp. 106-107 (2005)

[12] Sun Microsystems Inc.: DTrace User Guide (http://dlc.sun.com/pdf/819-5488/819-5488.pdf) revision from May 2006

[13] Bryan M. Cantrill, Michael W. Shapiro and Adam H. Leventhal: Dynamic Instrumentation of Production Systems, Solaris Kernel Development, Sun Microsystems (2005)

[14] Apple Inc.: Instruments User Guide (http://developer.apple.com/mac/library/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/Introduction/Introduction.html) revision from 20 January 2010

[15] Apple Inc.: Leopard Technology Overview (http://developer.apple.com/leopard/overview/) revision from 8 November 2007

[16] Jon Kleinberg & Éva Tardos: Algorithm Design, Pearson Education, Inc. (2006)

[17] Amitabh Srivastava & Alan Eustace: ATOM: A System for Building Customized Program Analysis Tools, in *SIGPLAN '94 Conference on Programming Language Design and Implementation* (1994)

[18] 20 Years of PLDI (1979 - 1999) (http://userweb.cs.utexas.edu/users/mckinley/20-years.html) from 29 March 2010.

[19] Statistical Inaccuracy of gprof Output (http://lgl.epfl.ch/teaching/case_tools/doc/gprof/gprof_12.html) from 29 March 2010

[20] Glenn R. Luecke, James Coyle, Jim Hoekstra, Marina Kraeva, Ying Li, Olga Taborskaia, and Yanmei Wang: A Survey of Systems for Detecting Serial Run-Time Errors, The Iowa State University's High Performance Computing Group (Revised 2006)

[21] Bryan Cantrill: Hidden in Plain Sight, in *ACM Queue*, February, pp. 26-36 (2006)

[22] Allen D. Malony, Daniel A. Reed and Harry A. G. Wijshoff: Performance Measurement Intrusion and Perturbation Analysis, in *IEEE Transactions on Pralell and Distributed Systems*, Vol. 3, No. 4, July, pp. 433-450 (1992)

[23] Stephen G. Kochan: Programming in Objective-C, Second printing, Sams Publishing (2004) ISBN: 0-672-32586-1