University of Edinburgh
Division of Informatics

A Real Time Profiling Tool with Three Dimensional Call Graph Visualisation

4th Year Project Report
Computer Science

Tim Angus

May 28, 2003

Abstract: A profiling tool is used to identify which functions in some arbitrary program are accumulating the most time and thus are candidates for optimisation. Existing profiling tools analyse pre-generated profiling data off-line, after the program under measurement has terminated. Often it would be useful to be able to examine the profile of a program *while it is still executing*. With a traditional profiling tool this data is presented in a textual form. If this method were applied to the proposed profiling tool however, the quantity of continually changing information would be overwhelming for most people to manage. Alternatively the data may be represented visually — often a more effective means of communicating information. This project aims to develop a tool capable of profiling a running process and rendering a call graph which incorporates the profile interactively.

Acknowledgements

I would like to thank Kousha Etessami, for agreeing to supervise my project and for offering ideas and advice therein.

Thanks to, Sandy Thomson, Meghan Perez, Ben Jansens, Alex Mordue, Amy-Anne Fairhurst, Charles Keepax, Andrew Seales, James Turnbull, Bruce Cran, Antonia Marland, Ryan McNally and Bob the Anonymous Monkey for participating in the usage study.

Contents

1. Introduction

A profiler is a tool used to generate a profile for some run of a program. This profile indicates to the user where the program under measurement spent its time. Additionally the profile provides data regarding the number of times each function is called and by what. More information may be calculated, such as call frequency.

Often a program is too complex to analyse and optimise at source level. A profiler helps here by identifying which functions consume the most CPU time and hence are the best targets for optimisation. More formally, this is known as *Amdahl's law*, postulated by Gene Amdahl in 1967.

$$s_{overall} = \frac{1}{(1 - F) + \frac{F}{s_{opt}}}$$

Figure 1.1: Amdahl's law

Here F is the fraction of a problem that can be optimised and s_{opt} is the speedup obtained by optimising that fraction. A profiler is used to identify the largest F in the program under measurement. A profiler can also be used to a limited extent as a debugging tool; sometimes a bug may manifest itself through either an excessive number of function calls or perhaps too few. This data is available from the profiler.

A traditional profiler operates as follows:

- Compile the program to be measured with some instrumentation.
- Execute the program to be measured to termination.
- Run the profiler on the profile data produced.

An important point here is that only the code actually executed has profile data generated for it. Therefore different runs of a program will generate different profiles. It is essential to bear this in mind when interpreting the results of a profiler.

A profiler is capable of producing two main distinct output forms, the *flat profile* and the *call graph*. A flat profile is simply a list of the functions in a program and their profiling details. This is most useful when the user wants to know the suboptimal sections of code in some program. A call graph is a graph in which

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 32.85      0.80     0.80  2511552     0.00     0.00  RotatePointAroundVector
 19.42      1.26     0.47      103     4.56     7.38  renderPolys
 11.98      1.55     0.29  1255776     0.00     0.00  persProject
  9.92      1.79     0.24      206     1.17     5.02  rotatePolyList
  1.65      2.33     0.04    85344     0.00     0.00  VectorNormalise
  0.62      2.36     0.01       21     0.71     2.62  calculatePolyNormals
  0.41      2.37     0.01    82944     0.00     0.00  B
  0.41      2.38     0.01       32     0.31     0.62  tesselatePatch
```

Figure 1.2: Example flat profile produced by gprof

graph nodes represent functions and graph edges represent function calls. Figure 1.2 shows an example flat profile output.

1.1 Existing profiling tools

The GNU project contains a wide range of development tools, one of which is *gprof*[1] — the GNU profiler. gprof is the lowest level profiler available on GNU based platforms. To compile a program (hereafter referred to as the client program) with profiling instrumentation, the option *-pg* must be added to the GCC command line (figure 1.3).

```
gcc -pg -o a.out main.c
```

Figure 1.3: Compiling a program for analysis with gprof

This adds instrumentation (code whose sole purpose is to gather profiling data) to the program being compiled. gprof and GCC are very tightly integrated such that GCC is capable of compiling code with instrumentation on a line by line basis. This allows gprof to produce a third output form, the *annotated source listing*. This is a source code listing with each line annotated by the number of times it was called. Typically other profiling tools cannot provide this facility because the granularity of their instrumentation is at the function call level, rather than per line.

When a program compiled with gprof instrumentation terminates, it writes a file *gmon.out* to the current directory. This file contains all the profiling data in a binary form. The gprof binary is then used to interpret this data and present it in a more human readable form. gprof is a very mature, flexible profiler and it supports a large number of platforms. Tight integration with GCC means that gprof is able to generate CPU timing data accurately. gcc -pg produces a binary that will sample the program counter in order to time stamp specific function events. The advantage of this is that the program being profiled is

unaffected by external load placed on the host computer. On the other hand, this approach ensures that the profile produced is completely unaffected by any asynchronous I/O that may occur. For example a file I/O call may block for an arbitrary length of time, but since the program counter is used for measurement, the profile produced will show this function as having used only a small amount of time. gprof is capable of profiling source code compilable with the C, Pascal or Fortran77 frontends. gprof's only major shortcoming is its inability to profile multithreaded and multiprocess binaries.

The *FunctionCheck*[2] profiler was created partially to alleviate this problem and others. GCC provides no direct extension mechanism for dealing with profiling instrumentation besides hacking the GCC source itself. It does however provide a facility to instrument functions at the function call level. This is enabled by supplying the `-finstrument-functions` on the GCC command line. Thereafter an instrumentation function is called on client function entry and client function exit. These instrumentation functions are written to gather profile data for some program.

FunctionCheck is a very capable profiler. Most importantly it handles multi-threaded programs by maintaining multiple call stacks for each separate execution thread. In addition to the function call profiles generated, FunctionCheck maintains a memory profile which tracks calls to *malloc* and *free*. The data garnered by this may be used to detect memory leaks (dynamically allocated memory which is never freed) or other anomalous memory allocation.

FunctionCheck consists of two main components; a client library and an analysis program. Usage of FunctionCheck proceeds as follows:

- Compile the program to be measured with `-finstrument-functions` and link it with the FunctionCheck client library *fnccheck*.
- Execute the program and utilise any features of interest. Terminate the program.
- Run the program *fncdump* on the profile generated by step 2. This is analogous to running *gprof* on a *gmon.out* file.

In addition to the usual *flat profile* and *call graph* output forms FunctionCheck provides a number of additions. The *cycles detection* output form is used for illustrating cycles in the call graph, for example recursive and mutually recursive functions. The *min/max time* output form is used for showing the minimum and maximum time spent in each function. The *function details* output form provides data regarding the filename and line number where functions are located. The *memory leaks and blocks* output forms present information about the memory profile generated. Finally the *multi-threads/multi-processes* output form details when and how a program forked or spawned child threads. The extra

functionality provided by FunctionCheck comes at a price. Firstly since the GCC instrumentation functions are utilised, FunctionCheck is not able to provide the per line profiles that gprof is capable of. This means that FunctionCheck is incapable of producing an annotated source listing. Secondly FunctionCheck uses the function `gettimeofday` to retrieve the time at which function events occur. This function returns *real time* so the accuracy of the profile is at the mercy of the operating system scheduler to a limited extent.

This problem is difficult to overcome without somewhat lower level interaction with the operating system. A related problem is solved by the *hrprof* [3] project. Instead of using `gettimeofday` or similar real time functions, hrprof uses the Pentium time stamp counter register to count individual clock cycles. Data gathered is still real time in nature as the profiler is no more aware of underlying process virtual time than other profilers, but it is inherently finer grained. hrprof's instrumentation is very similar to FunctionCheck's. In contrast to FunctionCheck however, hrprof is designed to be a drop in replacement for the gcc -pg stage when using gprof. hrprof writes a gmon.out profile file that is 100% binary compatible with gprof. In essence hrprof represents a slightly more fine grain data gathering process to gprof, rather than a complete profiler. The obvious disadvantage of the approach taken by hrprof is than its time stamping method is highly architecture and operating system specific - not every CPU provides the required facilities nor does every operating system provide the means for accessing them.

1.2 Motivation

All of the profiling tools examined share one thing in common - the analysis stage of profiling is performed entirely off-line on pre-generated profiling data. An alternative approach to this is to analyse the data produced in real time while the client program is still executing. The ability to observe the profile of a program in real time implies a much greater degree of interactivity for the user. In other words the user is able to react to certain profile characteristics much faster than is possible with the run, analyse, run, analyse profiling cycle of the traditional profiler. Clearly however, the overhead imposed by such a tool is greater than an equivalent off-line profiler since the analysis stage is being performed in parallel with the run of the actual program. Additionally this analysis overhead is not necessarily a constant across the entire run of a client program so the accuracy of the data produced is inherently less accurate than with an off-line profiler. Nevertheless the development of a real time profiling tool presents many interesting challenges and has the potential to be a genuinely useful development tool.

The interactive nature of a real time profiling tool means that any data produced

is variant over time and as such there is a significant challenge simply in providing a means to present this data to the user in an efficient, comprehensible manner. One way of doing this is to combine the *flat profile* and *call graph* into a single visual representation. The thinking behind this approach is that humans are congenitally better equipped at interpreting data when it is represented pictorially or graphically. This is epitomised by the often over used adage, *a picture is worth a thousand words*. Figure 1.4 emphasises this point. A graphically visualised call graph has potential uses out with profiling. Often a call graph may be useful in learning the structure of a complex code base. This data can easily be generated by gprof, but it remains in a relatively intractable textual form. The name *call graph* itself suggests that a graphical representation will be more intuitive that its textual equivalent.

Some previous work has been done in the area of call graph visualisation, but generally only involving static call graphs and not incorporating profile data. The project closest to this report is the *Call graph drawing interface*[4]. This small project is an interface to convert the call graph data generated by gprof into a form *VCG*[5] is able to understand. *VCG* is a graph drawing tool specifically designed for drawing many of the graphs involved in compilation and compiler development. The graph produced is 2D only and consequently is only really useful for small call graphs. Additionally the graph does not expose any information about the profile other than the call graph itself. Petter Reinholdtsen wrote a perl script[6] to perform much the same task, albeit with a *daVinci*[7] interface as well as *VCG*. This also indicates a minimal amount of profile data embedded in the graph itself, but ultimately it still suffers from the spatial limitations of a two dimensional coordinate system. *KProf*[8] is a front-end to *gprof* based on the *KDE* desktop project. It is a very mature GUI based tool but it offers little over gprof itself apart from a *GraphViz*[9] back-end to render call graphs like the other tools mentioned here. *KCachegrind*[10], also based on the *KDE* desktop project, is a front-end to *Cachegrind* which is in turn a part of the *Valgrind*[11] project. Valgrind is a CPU virtualisation program primarily intended for memory tracing and debugging. As a side effect, it is also suitable as a profiling tool. The most attractive feature of Valgrind is that in order to profile some program no re-compilation is necessary since all the instrumentation takes place inside the virtualisation. Currently the virtualisation doesn't have implementations for certain SIMD instructions like *MMX*, so binaries compiled for more recent architecture generations may have problems. Additionally there is the obvious disadvantage that Valgrind currently only virtualises the x86 platform, so it requires an almost complete re-implementation in order to be ported to other platforms. Nevertheless Valgrind/Cachegrind is an elegant solution for memory analysis. KCachegrind introduces a patch to Cachegrind to generate call *trees* (as opposed to call *graphs*) which are visualised using a coloured tile mechanism where tile size is proportional to CPU time expended.

```
granularity: each sample hit covers 4 byte(s) for 1.92% of 0.52 seconds

index % time    self  children    called     name
                                                <spontaneous>
[1]     57.7    0.00    0.30                 main [1]
                0.00    0.28      17/17          transformAndRenderPolys [2]
                0.01    0.01      32/32          tesselatePatch [9]
                0.00    0.00       1/1           buildColourTable [16]
                0.00    0.00       1/1           initPolys [18]
                0.00    0.00       1/1           parseFile [19]
                0.00    0.00       1/1           translationToCOM [21]
                0.00    0.00       1/1           drawInstructions [17]
                0.00    0.00       1/1           shutdownPolys [20]
-----------------------------------------------
                0.00    0.28      17/17          main [1]
[2]     53.8    0.00    0.28      17           transformAndRenderPolys [2]
                0.04    0.16      34/34          rotatePolyList [3]
                0.05    0.03      17/17          renderPolys [6]
-----------------------------------------------
                0.04    0.16      34/34          transformAndRenderPolys [2]
[3]     38.5    0.04    0.16      34           rotatePolyList [3]
                0.16    0.00  414528/414528      RotatePointAroundVector [4]
-----------------------------------------------
```

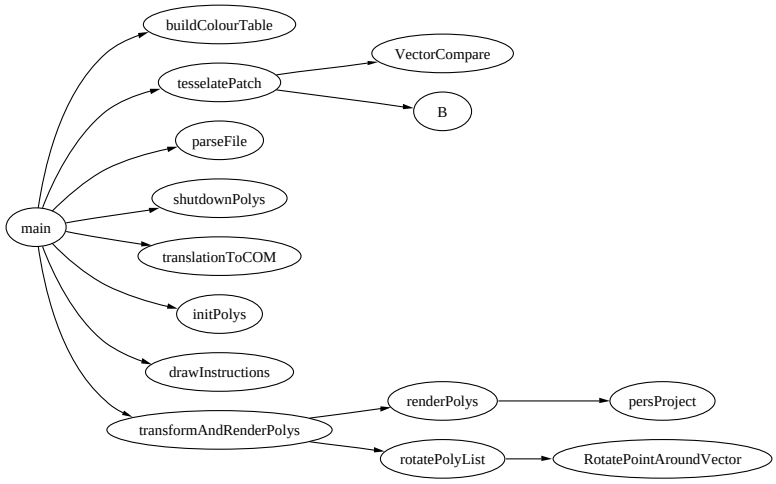

Figure 1.4: Example call graph extract produced by gprof and equivalent graphic representation

The primary motivation for this project stems from the fact that there simply isn't anything similar already implemented. Similarly, real time interactive three dimensional call graph rendering is not any area where there appears to be a great deal of prior work. Besides being an interesting undertaking, this project also represented an excellent opportunity to learn the technologies involved in the implementation.

1.3 Goals

The overall goal of the project is to implement a real time profiling tool with the ability to present the resultant data interactively using 3D graphics. There are a number of high level goals to be met as follows:

- The tool should be as portable as is realistically possible.
- The tool should operate on a network transparent basis. The reasoning behind this is explained in section 2.1.2.
- The tool should present the data in an intuitive manner that is easy for humans to interpret. Obviously this is somewhat subjective, so a small usage study will be performed to decide whether this has been achieved.
- Along a similar vein, the tool should be easy to use.
- The tool should be implemented sufficiently optimally to maintain interactivity on relatively modest hardware.
- The tool should be reasonably configurable.
- The tool must be able to profile programs written in the C language at a minimum. Such programs should not need any modification in order to be profiled.

2. Design

There are essentially two distinct, broadly separate parts to the project. There is the profiling tool itself — this must allow an arbitrary C program to be compiled to interoperate with a separate analysis program. Part of this analysis program is the 3D call graph visualisation that operates on the generated profile data. These problems are sufficiently modular such that they could be developed almost independently assuming a low coupled interface it agreed upon.

2.1 Real time profiling tool

The profiling tool needs to attach to the client program that is to be measured. There are a number of strategies that could be employed here, but at the lowest level it depends on the mechanism used to instrument the client program. *gprof* is unique from the perspective that its instrumentation is on a per line basis. Unfortunately this approach is relatively infeasible for the real time profiling tool. That is not to say it is impossible, but such an approach necessarily involves hacking GCC to communicate with an external analysis program rather than writing a *gmon.out* file on program termination. The extra unknown factor introduced through an unstable compiler increases the complexity of the implementation to a point where it is debatable whether the gains reaped by taking such a low level approach are valuable enough. It is certainly true that a per line profile is not easily or usefully visualisable in the form of a call graph.

Therefore a higher level of instrumentation must be used. GCC provides an option `-finstrument-functions`[12] which allows a program to be instrumented with two functions as seen in Figure 2.1.

```
void __cyg_profile_func_enter( void *this_fn, void *call_site );
void __cyg_profile_func_exit( void *this_fn, void *call_site );
```

Figure 2.1: GCC instrumentation functions

As their names suggest, these functions are called immediately following function entry and before function exit respectively. By implementing these functions it is possible to create a complete function level profiling tool by observing when calls to these functions take place. This is the approach taken by the *FunctionCheck* and *hrprof* profiling tools.

2.1.1 Architecture

Having established that function level instrumentation will be used through GCC it must be now be decided how the tool itself will integrate with a client program. If the client program is compiled with the `-finstrument-functions` option it must have implementations of the instrumentation functions corresponding to the given prototypes or it will fail to link at compile time. It would be possible to provide implementations of these functions in a C file that is compiled into the client program. A far more elegant solution is to implement these functions as a dynamic library which is linked into the client program as and when it is executed.

This library could be implemented using a monolithic paradigm whereby the entire profiling tool is implemented in the library itself. This is to some extent the approach taken by the *FunctionCheck* profiler. The vast majority of its functionality is implemented in the dynamic library. This seems a strange approach for an off-line profiler where any necessary calculation could be performed by the analysis tool post execution, having less of an effect on the client program's execution. Alternatively a lightweight implementation which only provides only the minimum functionality could be employed. The advantage here is that the impact of the instrumentation upon the client program is reduced and the accuracy of the profile is increased. This is a moot point from the perspective of a real time profiling tool however, since the analysis stage must be performed in parallel with the client program and necessarily incurs an overhead.

Yet the notion of a lightweight library with a low impact is still attractive. An alternative to the monolithic approach is to logically separate the tool into a dynamic library to be linked with the client program and a separate analysis program with which the library (optionally) communicates. The role of the library is simply to forward the data resultant from the instrumentation functions to the analysis program where it is interpreted and output in some fashion. This architecture is illustrated in figure 2.2. Clearly some IPC is necessary in order for the library and analysis program to communicate.

2.1.2 Network transparency

One of the goals stated in the introduction was that the tool developed should be network transparent. At first this may seem superfluous, however there are two subtle underlying reasons.

The first is simply a matter of available screen area. Given that the tool is to run in parallel with the program under measurement, both programs must be visible on the screen at the same time. Without resorting to multiple monitor

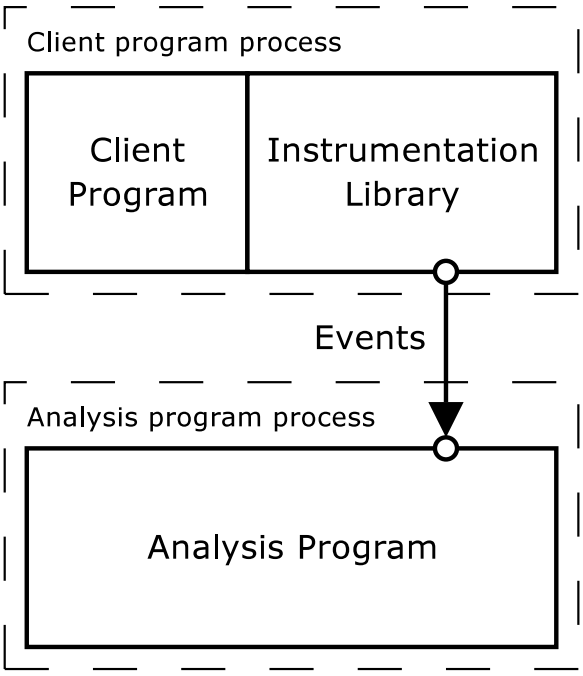

Figure 2.2: Real time profiling tool architecture

technologies or exceptionally high resolution displays it quickly becomes awkward to display the profiling tool and the client program simultaneously on the same screen. Additionally with both profiler and client program running on the same windowing system it is necessary to switch focus between each window frequently. If the two components are separated it becomes possible to use them at the same time, or at the very least somewhat more conveniently.

The second, perhaps more subtle reason stems from the fact that the profiling tool is measuring real time as opposed to process virtual time. In a scenario where both the profiling tool and client program are executing on the same multitasking operating system, the profiling tool visualisation represents a significant proportion of expended CPU time. In an ideal world, the operating system scheduler is perfect meaning that each process receives a constant quantity of CPU time. Especially in desktop operating systems this is rarely the case and inevitably on a highly loaded system a single process will receive varying quantities of CPU time with respect to time. The result of this in the context of the profiling tool is that any times measured using real time (i.e. with `gettimeofday`) will be distorted based on how long the process has been queued for. By moving the profiling tool to a logically separate computer system, the load on the system the client program is executing on is removed and the resultant profile will tend to be more accurate.

Obviously the requirement for network transparency heavily influences the choice

of IPC to use, to the point where the only sensible option is to use *BSD sockets*. Sockets are effectively a programmer level API for the Internet Protocol providing facilities for utilising TCP and UDP protocols. The connection between the client program and the profiling tool must be reliable since any losses or duplication will adversely affect the resultant profile. As a consequence UDP is not a suitable transport protocol. The data transferred between the two parties is discrete in nature, but the reduced overhead of a stream service and the connection orientated approach make TCP a good choice.

2.1.3 Data transfer

In order for the library to communicate with the analysis program, some information must be transferred. The greatest influence here is the knowledge available to the library through the instrumentation functions. There are essentially three separate variables available to each instrumentation function call; the time at which it was called, the symbol of the function and the return address of the function (implying the call function). At first glance this would appear to be entirely sufficient, but the return address (named `call_site` in the function) complicates matters.

The C keyword inline is a new addition to the C99[13] language standard. Functions specified with this keyword are expanded inline as they are called, avoiding the need for a stack frame. Conceptually this construct is intended as a replacement for preprocessor macros, preventing many of the associated pitfalls. Strictly speaking an inline function call should not be instrumented since once expanded by the compiler it no longer creates a stack frame when called. Inexplicably however, the instrumentation functions are still called for inline functions. Furthermore, the return address for such an instrumented function points to the caller of caller of the function, assuming it exists at all. There are two potential solutions to this problem. Firstly, to disable inlining altogether — this is possible with the `-fno-inline` GCC option. Disabling inlining is an unreasonable action to take since it involves changing the client program structure in order to operate. This contradicts one of the original goals set out in the introduction. It is also the case that on some platforms the `call_site` variable does not work at all. Secondly, it is possible to maintain a call stack for the client program. In this case the caller function is implied by the top of the call stack and maintaining caller identification is unnecessary. With this approach it is possible to simply ignore the `call_site` variable altogether and rely on the call stack to maintain caller and callee relationships.

The profile is only updatable when the instrumentation functions are called due to a change in the stack of the client program. The information from these calls is discrete and can be represented as an event. The data transferred between

Event type	Event payload	Event purpose
Function entry	Time stamp, Function symbol	Generated when a function is entered.
Function exit	Time stamp, Function symbol	Generated when a function exits.
Process exit	none	Generated to indicate to the analysis program when the client has terminated.

Figure 2.3: Event types

instrumentation library and analysis program consists of three things; the event type, the timing of the event and the symbol involved where the event type is one of function entry, function exit and process exit. These events are summarised in figure 2.3. The role of the instrumentation library now is simply to produce events as the client program executes while the analysis program consumes them with an event handler.

2.1.4 Data interpretation

Once a communication channel has been established between the instrumentation library and the analysis program, the data must be interpreted to form the profile. Broadly, this can be done two ways; synchronously and asynchronously. A synchronous interpretation of the data consists of a regular loop which consumes events as and when they become available. The natural approach is to handle the events asynchronously — processing the events when they arrive at the analysis program. Each approach has its merits.

Consider the scenario where a function in the client program has been entered and then blocks for an arbitrary length of time. Using the asynchronous model, following the initial function entry event the profile cannot be updated despite the fact that the function is still executing and accumulating time. The profile will of course be updated when the function exits. Using the synchronous model it is possible for the event handling loop to update the profile accordingly despite the lack of inbound events. The synchronous model will however be somewhat more CPU intensive since each iteration of the event loop is executed regardless of whether or not there is an event to process. The asynchronous model represents the minimum CPU usage since it only adjusts the profile when new data becomes available.

The synchronous approach is potentially more complex to program than the asynchronous approach since inter-event profile updates must be made. It is questionable whether these updates are worthwhile given that a program which

blocks frequently is unlikely to benefit from profiling. This coupled with the extra CPU usage mean that the asynchronous approach is most sensible.

In addition to the number of calls, there are two times that must be accumulated for each function, the *total time* and the *local time*. The total time spent in a function is calculated by measuring the time from when a function enters until it exits. The local time spent in a function represents the time actually spent executing, in other words the time the function spends on top of the call stack. This is better explained in figure 2.4.

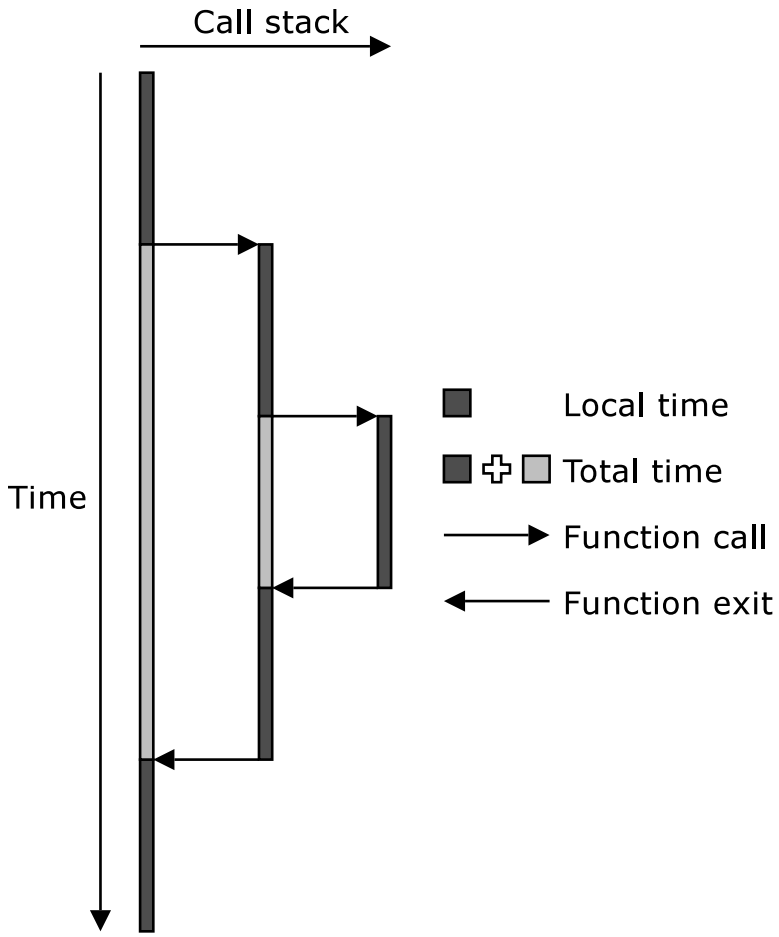

Figure 2.4: Local and total time

The call stack is maintained by the analysis program to keep track of these times. Each new call interrupts the local time of the calling function while leaving the total time to accumulate until the callee function exits.

The profile itself requires two data structures. Firstly a list of functions which contain the information total time, local time, total number of calls and call recency. This list must also indicate some identification for each function, either by

symbol or textual representation or both. The second data structure is required to maintain caller and callee relationships between the functions in the program. This structure is more open ended in terms of representation and is discussed in section 3.1.2.3.

2.2 Visualisation

With the underpinnings of the actual profiling mechanism proper in place, attention must be diverted to the presentation of the data. The ultimate goal is to present the data in the form of a call graph rendered in three dimensions, but alternative representative forms are also necessary. It would be foolhardy to embark on a complex means of representation without first ensuring the validity of the underlying data. It is prudent therefore to explore a means of output which is both simple and not open to interpretation.

An ordered flat profile as produced by gprof would illustrate the flat profile well. With a real time profiling tool however, the flat profile is constantly changing. A simple means of reflecting this is to show the profile at regular intervals in a similar manner to the Unix load monitoring utility *top*. Having established the integrity of the flat profile, the call graph must then be examined. Rather than attempt to examine the call graph in a textual form, an existing graph drawing tool can be used such as *GraphViz*[9]. Given a graph such a tool will utilise a graph layout algorithm to position the graph nodes and edges. Thereafter it will proceed to render the graph using the requested output format. A function which translates from the real time profiling tool's graph representation into a graph drawing tool's representation may be written relatively trivially. Such a function can be executed to produce a call graph at any point in the program's execution. Due to the nature of the graph drawing tool however such an approach will not yield a real time representation of the call graph. Ultimately it fulfils its purpose in providing a means to ensure that the call graph data is generated correctly.

The most important output method is the 3D call graph, rendered in parallel with the profile generation. This must expose information regarding the flat profile of the client program in addition to its actual call graph.

2.2.1 2D versus 3D

It is questionable whether or not a three dimensional representation is even necessary. In order to answer this question it is wise to re-examine the goals of the visualisation. The primary motivation is to provide a means by which a program's profile can be observed in real time with the minimum of ambiguity and

the maximum clarity.

Another important consideration is the ease of navigation of the resultant graph. This is especially important for large, complex graphs. Given only two dimensions the bounding area of the graph is given by a plane. Similarly for three dimensions the bounding area is a cuboid. If the same graph is represented in both two and three dimensions then the same amount of information is contained within each respective bounding area. Assuming a complex graph it is a realistic assumption that it is not possible to view the entire graph simultaneously while remaining legible and consequently useful. If views from each respective representation are taken such that each is just past a legibility threshold (clearly this is somewhat subjective) then the three dimensional projection will invariably contain more information than the equivalent two dimensional representation. The main consequence of this is that less navigation is required for the three dimensional representation.

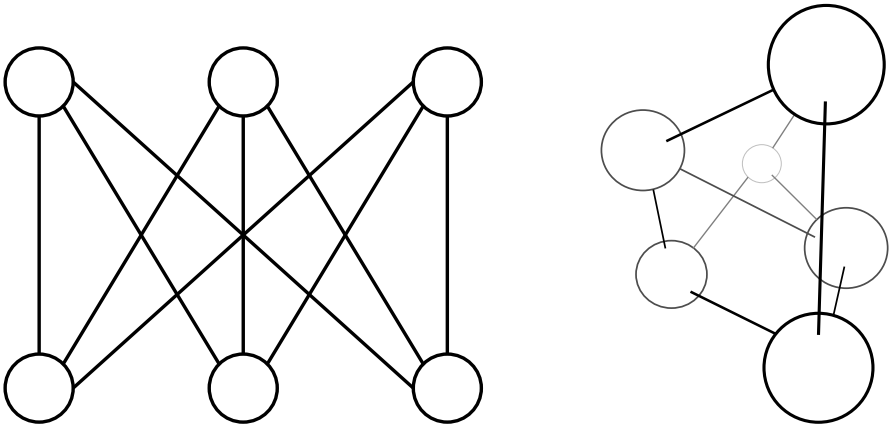

Figure 2.5: 2D and 3D layouts of the non-planar complete bipartite graph $K_{3,3}$

Besides spatial reasons, overall graph clarity is also very important. As with much of graph layout theory, the clarity of a graph is largely a subjective, aesthetic variable. Certain obviously undesirable properties can be reasoned about using mathematics. Edge crossings are one such property. A planar graph is defined as a graph that can be drawn on a plane (i.e. in two dimensions) such that no two edges cross. Unfortunately all but the most simple call graphs are non-planar meaning that their two dimensional representations are confused by edge crossings. Using a three dimensional representation however the graph nodes are not constrained to a plane thus the possibility of edge crossings is virtually eliminated. This is shown in figure 2.5. Depending on the layout algorithm used it is possible that subgraphs of a complete call graph are laid out on a plane and are vulnerable to edge crossings. In practice however this is relatively rare and the problem of non-planar graphs can be safely ignored.

2.2.2 Available graph attributes

Besides rendering the topology of the call graph, the flat profile of the client program must somehow been incorporated into the graph. Figure 2.6 shows the elements of the flat profile that can potentially be visualised in the call graph.

Variable	Explanation
Total time	The time spent between function entry and exit
Local time	The time spent when the function is on top of the call stack
Number of calls	The total number of calls made to the function
Edge calls	The number of times a function called another
Call frequency	The frequency with which the function is called
Call recency	How recently the function was called
Function name	The name of the function

Figure 2.6: Visualisable data

Ignoring the function name for now, all of the visualisable variables are numeric. Broadly speaking there are two domains that can be used in visualising the flat profile variables, the spatial domain and the intensity domain. The spatial domain refers to the size and shape of the glyphs that represent elements of the graph. The intensity domain specifies how these glyphs are coloured and combined with the rest of the scene. This can be split up in to various sub-components that may be used as indicators for the available data. Figure 2.7 shows the graph attributes that may be changed to represent profile features.

The colour attribute requires the most attention. When rendering a scene in three dimensions the surfaces are shaded to infer three dimensional shape on the two dimensional framebuffer. In order to use the intensity domain to represent a continuous variable therefore it must be shared to some extent with the shading algorithm used. The colour space used with the colour attribute is important, of which there are multiple options. The most commonly used colour space is RGB short for red, green and blue. The human eye is a trichromatic integrator, meaning that it measures light in three specific wavelength ranges. The RGB colour space is representative of these ranges. Potentially it offers three individual channels for three continuous variables respectively but since the eye interprets these channels as a single colour some thought is required to separate these channels when the user is viewing the visualisation. An alternative colour space to use is HSV, short for hue, saturation and value (brightness). The hue refers to the gradation of the colour, i.e. whether the colour is red, green, blue or other intermediate shade. The saturation of the colour refers to the intensity of the colour in the sense that a desaturated colour is diluted with a grey value more heavily than a saturated colour. The value channel gives the overall brightness of the colour.

Attribute	Explanation
Size/Scale	The glyphs can be scaled according to some continuous variable
Shape/Form	The glyphs themselves have some shape. Typically this attribute is best for representing discrete values.
Colour	The glyphs can be coloured according to a single continuous variable or potentially many. The colour space used is influential here.
Alpha	Related to colour, the alpha value refers to how pixels are combined with other coincident pixels. This is the transparency of the glyph and is mappable to a continuous variable.
Texture	A texture may be mapped to the surface of the glyphs. Similarly to shape this variable is probably best at representing discrete values. Since it uses the same domain as colour however, this is harder to combine with colour changes and still remain useful.

Figure 2.7: Available graph attributes

The HSV colour space is somewhat more suited to separation by humans since each channel is more distinct as far as the human perception system operates.

For the purposes of call graph visualisation, the number of variables which need to be simultaneously monitored is relatively small. Instead of introducing confusion into the colour space by using each channel separately a colour map can be used. A colour map is a mapping of scalar value to colour space. Typically colour maps are designed to be aesthetically pleasing rather than having a concrete mathematical basis. A common real world example of the use of colour maps is in representing elevation data on topographical maps. Instead of treating the colour of graph glyphs as multiple channels, a colour map can be used to represent a single continuous variable.

The alpha channel of a glyph may be assigned to another variable independently of the colour attribute. While it operates in the same domain as colour its effect on pixel colour is significantly different to a colour channel such that it can easily be interpreted by humans as separate to the colour applied. A sensible application of this attribute is to the call recency variable whereby functions not called recently are shown progressively more transparent.

Using a three dimensional perspective projection, a projected object's apparent size varies inversely with its distance. In two dimensions the view distance is effectively constant and as such rendered objects are the same size regardless of view point. When representing a variable in the three dimensional spatial

domain care must be taken. For example given two identically shaped objects displayed at different distances there might be an instance where each object is scaled such that when projected they appear to be the same size. Clearly when a variable is mapped to the object scale attribute this falsely reflects the value of the actual data. This problem occurs because humans infer the third dimension in the scene (depth) by the perspective projection. The context in which objects are represented in a scene is helpful to some extent. In the previous example if each of the two objects were rendered in the same spatial region as another object of known constant size the relative size of each would help to infer the size of each object in the scene. This is the case in the rendering of a call graph since each glyph is directly connected to at least one other glyph. The most effective solution is to provide some means whereby the depth dimension is not inferred. True stereoscopic vision systems are expensive and relatively rare however, so this is an unrealistic general solution. The use of glasses with red and blue filters to mask two superimposed images is a common means of producing stereoscopic vision cheaply. In order to work however, it substitutes resolution in the intensity domain to produce the depth perception effect.

2.2.3 Graphics library

Computer graphics is a relatively young, constantly evolving field of computer science. As such the range of tools and libraries available is large and diverse. While there are generalised, accepted graphics libraries available their use isn't always necessarily the best course of action.

The field of visualisation has spawned tools tailored specifically for the visual representation of data. One such tool is *VTK*[14] — the Visualisation Tool Kit. VTK is an object orientated framework consisting of a vast hierarchy of classes dealing with various aspects of data visualisation. The coverage of VTK is quite exceptional, catering for a large proportion of visualisation problems. Indeed the infrastructure provided by VTK is more than sufficient to implement the call graph renderer — going as far as providing implementations of graph layout algorithms. This flexibility comes at a price however. VTK is a very large and unusual dependency for a program to have, 99% of which will be redundant. Additionally despite having all the pieces necessary to create a call graph rendering tool, the granularity of the control available to the user is somewhat more coarse than with a generalised graphics library. This has implications for overall optimisation of the implementation which generally requires fine control of the source code involved. VTK itself is not a graphics library but a set of tools used to transform datasets into graphics primitives. It does however depend on a graphics library — *OpenGL*[15].

The 'open' in OpenGL is short for open standards and GL for graphics library.

Descended from SGI's IRIS GL, OpenGL was first ratified in 1992. OpenGL is not an implementation but a 2D and 3D graphics API designed from the ground up to be portable, scalable and extendable by an entity known as the Architecture Review Board (ARB). The ARB is comprised of leaders of the graphics industry, currently including representatives from 3DLabs, Apple, ATI, Dell, Evans and Sutherland, Hewlett-Packard, IBM, Intel, Matrox, NVIDIA, SGI and Sun Microsystems. Implementations are available for virtually every conceivable platform including *Mesa*[16], open source under the *GPL*. Furthermore, hardware accelerated OpenGL implementations are now common place such that relatively cheap graphics adapters are capable of providing significant speed increases for any graphics application using the library. OpenGL is a very capable graphics library including a well defined extension mechanism for the incorporation of new innovations produced by the ever evolving computer graphics field.

For the purposes of rendering a call graph, OpenGL provides all the required facilities. From the perspective of maintaining portability OpenGL is the only sensible choice when compared with competing technologies such as Microsoft's *Direct3D*[17], a graphics library with similar industry support to OpenGL but supporting only the *Windows* platform. The OpenGL API itself is also pleasant to use and very well defined.

2.2.4 Graph layout algorithm

A graph G is a set of vertices v connected by edges e, thus $G = (v, e)$.

Figure 2.8: Definition of a graph

A significant proportion of the graph rendering problem is deciding on the physical locality of the glyphs that make up the graph. This is the responsibility of the graph layout algorithm. Many such algorithms have been developed, partly because the success of the algorithm is defined by aesthetic criteria and as such fairly subjective. In general the problem of graph layout is specified as a placement of the graph vertices v (see figure 2.8) such that some aesthetic quality is achieved or approached. For the purposes of rendering the call graph in real time however, the overriding requirement is that the algorithm used is efficient and relatively scalable. Aesthetics are still important, but not at the expense of interactivity.

In 1984 Peter Eades[18] wrote a paper on the method of force directed placement for generalised graph layout. His heuristic modelled the graph as a physical system, where graph nodes were represented by rings and edges by springs. Eades' springs did not conform to Hooke's linear spring law however, instead using a logarithmic term. Additionally each ring in the system repulsed every other, much

like electrically charged particles with the same charge repel each other. Given any initial condition, the forces active on the graph components are iteratively computed and graph nodes moved until a state of equilibrium is reached. In such a state (of which there may be many), the position of the graph nodes is reasonably isotropic and the layout is generally accepted to be a pleasing. Further work has since been put into force directed placement. Kamada and Kawai[19] modified Eades' force calculations such that the distance between non-adjacent vertices was made proportional to the shortest path (graph-wise) between them. The thinking behind this approach is to attempt to ensure physical locality for graphs nodes that were logically local to one another. Fruchterman and Reingold[20] further developed this idea such that graph nodes are not laid out too close to one another.

Another approach to graph layout is simulated annealing, first explored in 1989 by Davidson and Harel[21]. Annealing is the process of heating and cooling some material in order to alter its structure to a more regular form. A simulation of this process is applied to a graph such that after an iteration a node is moved to a potentially more suitable position. This suitability is measured based on a probability. At high virtual temperatures movements are large. As the simulation is cooled each successive movement is reduced in magnitude until some predefined threshold is surpassed. Simulated annealing's main strength is its ability to layout planar graphs without any edge crossings.

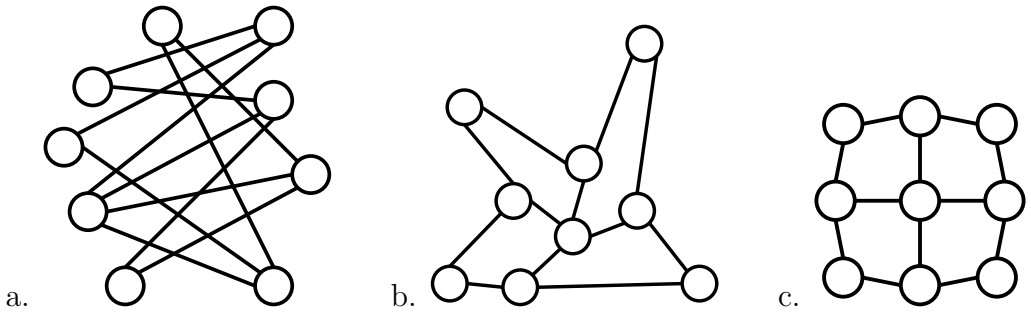

Figure 2.9: Example results of simulated annealing (b) and force directed placement (c) layout algorithms applied to an initial random configuration (a)

In comparison to force directed approaches however, symmetry and geometric structure are not exposed to the same degree. The benefits of edge crossing avoidance are very small when these algorithms are extended to three dimensions (see section 2.2.1). More crucially the performance of the force directed approach is superior to simulated annealing. Furthermore, without any edge length normalisation the results of force directed placement are typically aesthetically nicer. This is shown in figure 2.9.

There are numerous other approaches to graph layout, albeit not applicable to

the general case. Ostensibly the character of call graphs is often tree like, but it would be naïve to assume this is always the case. Therefore a force directed layout algorithm is the best approach to take. The earliest research into graph layout was confined to two dimensional graphs — extension to three dimensions is trivial however. There are several more recent papers concerned purely with three dimensional graph layout. These include *Fast Interactive 3-D Graph Visualization*[22] and *Some Three-Dimensional Graph Drawing Algorithms*[23].

3. Implementation

Having made the required design decisions, attention must now be turned towards the actual implementation of the tool.

3.1 Profiler

There are two elements to the profiler itself. Firstly a dynamic library named `librtprof.so` that is linked with the client program being profiled. This library then communicates with an analysis program, `rtprof`. `rtprof` is short for real time profiler.

3.1.1 Instrumentation library

As first described in section 2.1, the GCC option `-finstrument-functions` is used to instrument every function in the client program. The intended architecture of the profiling tool is a dynamic library linked with the client program which communicates with an external analysis program (figure 2.2). In order to satisfy the requirement for network transparency some IPC is required, in this case BSD sockets.

For the instrumentation library to communicate with the analysis program however, it must first establish a connection. Since the analysis program does not necessarily run on the same physical computer system as the client program, the instrumentation library must be instructed where to connect. This indication mechanism is provided by an *environment variable* named `RTPROF_SKT`. When a client program is executed the first call to `__cyg_profile_func_enter(...)` attempts to parse this environment variable (figure 3.1) and connect to the indicated location.

<Socket variable>	→	<TCP socket> | <Unix socket>
<TCP socket>	→	rtprof://<hostname>
<Unix socket>	→	unix://<filename>

Figure 3.1: Grammar for the environment variable `RTPROF_SKT`

The profiling tool supports two socket types, both of which utilise a reliable stream transport. The usual means of connection is via a socket using the TCP protocol. This method is used when the analysis program and instrumentation

library exist on separate computer systems. An alternative means of connection is provided by a so-called unix socket. When a process (in this case the analysis program) opens a unix server socket a special file is created by the operating system. Connecting clients then use this file as a means of identification. Communication between parties occurs within the operating system. The advantage compared to using a TCP socket is that there is no networking overhead incurred and as such communication is somewhat more efficient. Obviously of course this is only useful when the client and analysis programs are executing on the same computer system.

A desirable property for the instrumentation library was to ensure a low impact on the client program. In terms of implementation this means that the library should contain the absolute minimum of necessary functionality:

- Connection with the analysis program and error recovery therein.
- Report function entry events to the analysis program.
- Report function exit events to the analysis program.
- Report process exit events to the analysis program.
- Handle the abnormal termination of the analysis program or network failure to that effect.

There are two failures that can occur while the instrumentation library attempts to connect to the analysis program. Firstly the parsing of the environment variable `RTPROF_SKT` may fail. This can occur either because the user has not used to correct syntax, or because the user has not set the environment variable at all. Secondly the socket connection itself may fail. Usually this will occur when the analysis program is not running, but it could conceivably happen under other circumstances such as intervening network filters preventing connections on the port used. Neither of these errors are fatal — should they occur the instrumentation library will print a message to `stderr` to that effect and the client program's execution will continue as normal albeit with a performance penalty incurred by the (in this case redundant) instrumentation functions.

```
MSG_NOSIGNAL
       Requests  not to send SIGPIPE on errors on stream oriented sock-
       ets when the other end breaks the connection. The EPIPE error is
       still returned.
```

Figure 3.2: Extract from the `send(2)` manual page

In the case where the analysis program fails (hopefully this isn't a frequent occurrence) or is deliberately terminated or externally killed, the instrumentation

library must ensure that the client program does not fail or suffer otherwise adverse effects. By the very definition of failure, the analysis program does not indicate to the instrumentation library that it has failed, so the instrumentation library must detect this. Usually when `send(...)` is called on a closed stream, it generates a signal `SIGPIPE` to the calling process. The programmer must write a signal handler to take recovering action. In the case of the real time profiling tool however, the instrumentation library must have the minimum impact on the client program. It might be the case for example that the client program already has a signal handler for `SIGPIPE` and generating this signal could potentially cause confusing errors. Instead therefore, the instrumentation library needs to handle broken streams without the use of signals. This is accomplished by the `MSG_NOSIGNAL`(figure 3.2) socket flag. Instead of calling a signal handler the instrumentation functions simply check the return value of `send(...)` and cease further socket operations if a problem is detected. The socket code is clearly the slowest part of the instrumentation, so if socket operations are stopped the client program will experience a sudden increase in speed.

```
typedef struct functionEvent_s
{
  unsigned char type;
  void          *this_fn;
  timeStamp_t   ts;
} functionEvent_t;
```

Figure 3.3: The `struct` passed by the instrumentation library

Figure 2.3 listed the events that must be transmitted from the instrumentation library. A header shared between the instrumentation library and analysis program defines a `struct`, `functionEvent_t`(figure 3.3). As the name suggests, this struct embodies the events passed to the analysis program which in turn forms the profile. The field `type` is one of `EV_ENTER`, `EV_EXIT` or `EV_PROCEXIT` — the `enum`, `event_t`. Each of these corresponds to an event type. `this_fn` is one of the parameters passed to each instrumentation function whose purpose is to identify the function involved. It is an offset within the binary file which may be resolved to a human readable textual function name by a separate library. `ts` is an `unsigned long long` time stamp indicating when the event occurred. On the *Linux/x86*[24] platform, this has size 8 bytes and stores the time as returned by `gettimeofday` in the form of the number of μ-seconds since the Epoch (00:00:00 UTC, January 1, 1970). An alternative to this approach would be to keep the time in the format returned by `gettimeofday`, however computing the difference between two clockwise times is more complex than with two combined times. Appendix A shows the instrumentation functions as implemented.

3.1.2 Analysis program

3.1.2.1 Symbol resolution

A parameter of each event passed by the instrumentation library is the symbol of the function that is involved. This takes the form of an offset within the client program's binary file. In human terms this is close to useless from the perspective of source level profiling. Therefore some means of converting between the symbol offset and a human readable function name was required. The GNU Binutils[25] package contains a variety of utilities for the manipulation of binary files. One of these utilities is `nm`, whose purpose is to list human readable symbols in a binary file. On a programmatic level, `nm` and the rest of Binutils use a library called *BFD* — the Binary File Descriptor library. This library is used by the GNU assembler *gas* and GCC in order to manipulate binary files as necessary. For the purposes of the real time profiling tool however, BFD is only used to replicate the functionality of `nm`.

When the analysis program is started one of the first things it does is to parse command line arguments using `getopt_long(...)`. After processing command line switches, the remaining command line arguments are assumed to be one or more binary files. Each of these is processed in turn and BFD is used to list and convert the symbols within each. Each function name is inserted into a bucket-chain hash table hashed by offset symbol. Thereafter the function names corresponding to each offset symbol may be efficiently retrieved as required.

3.1.2.2 Parallelism strategies

It is the responsibility of the analysis program to perform two tasks; generate the profile from the event data transmitted by the instrumentation library and secondly to present the profile graphically. In order for these to be executed simultaneously, some form of parallelism is required. Initially the use of threads was explored through the *POSIX Threads(pthreads)* library which is a part of glibc[26].

The first implementation using pthreads was ostensibly a success. Both the profile generation and visualisation executed harmoniously and smoothly. At this point however mutual exclusion issues had not been considered and the visualisation occasionally caused segmentation faults as it attempted to read data from the profile while it was still being written. The event handler was thenceforth designated to be a critical section in which the visualisation was disallowed access to the profile. In this case however, the analysis program spent more than 95% of its run time in this critical section, the result of which being that the visualisation was severely starved for CPU time and largely unusable

due to a lack of interactivity. Two solutions presented themselves. Firstly by maintaining the threaded architecture but splitting the visualisation so that it would render a frame regardless of the state of the critical section. In this case the visualisation would only render a frame for the most recently available data, a copy of which it maintained itself. If the critical section was clear, this copy would be updated with the most recent data. However, this approach was regarded as another complication to an already complicated system.

The choice to use a threaded architecture in the first place was heavily influenced by the decision to use *blocking sockets* in the event handler. The term blocking sockets refers to socket operations that halt further program execution until they are complete. In terms of the real time profiler the use of blocking sockets means that when an arbitrary client program blocks in an arbitrary function, the event handler of the analysis program blocks too. When using blocking sockets in this way therefore, some high level parallelism was needed. If the event handler was rewritten such that it used non-blocking sockets however, execution of the analysis program could continue regardless of the state of the client program. Calls to the visualisation renderer could now be interleaved with the event handler so that outwardly each part of the analysis program apparently executed in a separate execution thread. In reality these elements are internally scheduled within the same process.

This is the approach that was taken. Initially the use of non-blocking sockets was avoided due to the extra complication of handling partial events. In hindsight however, the non-blocking sockets implementation is somewhat more simple than the previous equivalent implementation using threads. As an added bonus, the analysis program no longer depends on the pthreads library.

3.1.2.3 Generating the profile

```
typedef struct stackFrame_s
{
  void                *symbol;

  timeStamp_t         calleeEntryTime;
  timeStamp_t         calleeExitTime;

  struct stackFrame_s *next;
} stackFrame_t;
```

Figure 3.4: The `struct` that is placed on the stack

Central to the generation of the profile is a stack on which function calls are placed. Each 'stack frame' on this *call stack* contains two time stamps; the time

at which a child function is entered and the time at which it exits (figure 3.4). When a function is entered its `calleeExitTime` time stamp is set to the event time stamp. Thereafter the event handler computes the difference between event time stamps and the callee entry and exit time stamps to advance the local and total time (figure 2.4) for the function(s) involved.

This asynchronous approach implies that the profile is only updated when events actually occur. This is generally not a problem since programs that would benefit from being profiled will typically produce an uninterrupted stream of events. There is a more subtle deficiency in the stack based approach however. The very nature of a stack implies that only the top of the stack is both visible and manipulable. For the calculation of local time this is not an issue since a function will only accumulate local time when it is on the top of the call stack. For total time however, every function on the call stack accumulates total time simultaneously. Using a stack, total time for some function is only updated once a direct descendant is entered or exited. Therefore functions near the bottom of the stack have diminished, conflicting total times in relation to those on the top of the stack. In reality this doesn't have any particular detrimental implications since function total time is of little use when interpreting a program's profile. Currently it is not actually visualised. This problem is by no means insolvable. A possible solution is to pop the entire stack at every function event, update each function's total time accordingly and repush each popped function back on to the stack. Alternatively a more sensible solution is to pervert the stack abstract data type such that it is 'transparent' and may be addressed like a linked list.

The stack alone is not enough to generate the profile. The data garnered by computing time stamp differences must be stored in some data structure. Since the ultimate goal of the real time profiling tool is to visualise a call graph, a call graph abstract data type is prudent. The datatype is embodied in the `struct` shown in figure 3.5. This stores various global data relating to the call graph such as `totalCalls` — the total number of function calls made. The graph nodes and edges themselves are stored in separate sub-data structures.

The `graphNode_t struct` stores information pertaining to each function in the client program. Instances of this `struct` are stored in a bucket-chain hash table in a similar manner to the symbol table covered in section 3.1.2.1. Retrieval of the graph node data is facilitated by a function `graphNode_t **listNodes( sortField_t sf, int *n, graph_t *g )` which uses `malloc(...)` to allocate a linked list of `graphNode_t struct`s. The list returned by this function is optionally sorted by `qsort(...)` on a specified field. The graph ADT originally used an adjacency matrix to represent edge connections between nodes. For small programs with few functions this strategy worked well. For larger programs however, analysis proved that typically less than 2% of the elements in the matrix were utilised. Furthermore, listing this data structure (as required by the layout

```
typedef struct graph_s
{
  int          numNodes;
  graphNode_t  *nodeBuckets[ MAX_BUCKETS ];

  int          numEdges;
  graphEdge_t  *edgeBuckets[ MAX_BUCKETS ];

  timeStamp_t  totalLocalTime;
  timeStamp_t  totalTotalTime;

  timeStamp_t  maxLocalTime;
  timeStamp_t  maxTotalTime;
  timeStamp_t  maxInactiveTime;

  long         maxEdgeCalls;
  long         maxNodeCalls;

  long         totalCalls;
} graph_t;
```

Figure 3.5: The `struct` used to represent graphs

algorithm) confirmed the inefficiency of this representation, with each cell of a potentially very large matrix needing to be queried individually ($O(n^2)$ space and listing efficiency). In place of this flawed edge representation therefore, a simple edge list bucket-chain hash table was implemented — very much similar to that used by the node hash table. This is embodied by the `graphEdge_t struct`.

One element of the profile is generated differently to the others. While total time, local time and number of calls are all accumulated asynchronously by the event handler, call recency is evaluated synchronously based on the time as reported by the analysis program. The reason for this is so that in the case where the client program blocks functions are not falsely represented as being active. The code for the event handler may be found in appendix B.

3.2 Output methods

The profile of a program is useless without some means of observing it. This section details the implementation of the methods with which the profile is presented.

3.2.1 GraphViz output

In order to verify the validity of the generated call graph, a module was written to output the profile data in a form that could be converted to a graphic representation. Section 1.2 briefly mentioned a tool named *GraphViz*[9]. GraphViz is a set of tools to layout directed graphs using a variety of target file formats. At the core of GraphViz is a graph description language that specifies graph nodes and connecting edges. Extra options are available to perturb graph nodes in some defined manner — for example it may be specified to scale edge lengths by some factor. The *DOT language* is described in appendix C.

For the purposes of graph verification, only a small subset of the DOT language is used. The function shown in figure 3.6 writes a digraph to a specified file, or `stdout`. If the command line option `--dotfile` is given this function is called when the client program exits. Each edge and the nodes on which it is incident are listed in the output file. Thereafter this valid DOT file is converted to a graphic form. GraphViz provides two tools for this, `dot` and `neato`. `dot` utilises a rank based layout algorithm and consequently is best suited to laying out hierarchical data. Similarly to the analysis program, `neato` uses a force directed placement layout algorithm and as such is more suitable for visualising call graphs. Figure 3.7 depicts a call graph laid out by `dot`.

Since the use of GraphViz is simply to verify the integrity of the call graph - the profile data itself is not incorporated into the DOT file. However this is possible. The DOT language provides facilities for colouring graph nodes and altering their shape. A possible future avenue of interest would be to extend the GraphViz frontend such that it exposed more detail regarding the profile proper.

3.2.2 OpenGL output

OpenGL[15] is a standardised graphics library used for rendering two and three dimensional graphics to a framebuffer. It supports a variety of graphics facilities such as Z-buffer based depth sorting, back face culling, alpha blending, lighting and shading. Combined with its companion library GLU, short for GL Utilities, OpenGL can also render basic 3D geometric shapes. OpenGL alone is not responsible for rendering directly to the screen however. In order to do so, some 'glue' is required between OpenGL and the underlying windowing system of the host operating system. On XFree86 this is provided in the form of an X extension named GLX, while Microsoft Windows uses a mechanism called WGL. Each support the notion of an OpenGL context — a region of memory which is mapped to some window or otherwise visible screen region. At this level the management of OpenGL contexts is decidedly unportable.

```
/*
===============
dotOutput

Write a graphviz dot file
===============
*/
void dotOutput( char *filename, graph_t *g )
{
  graphEdge_t **p;
  int         n, i, j;
  FILE        *f;

  if( !strcmp( filename, "-" ) )
    f = stdout;
  else
    f = fopen( filename, "w" );

  p = listEdges( &n, g );

  fprintf( f, "digraph callgraph\n{\n" );

  for( i = 0; i < n; i++ )
  {
    fprintf( f, "\t\"%s\" -> ", p[ i ]->from->textSymbol );
    fprintf( f, "\"%s\";\n", p[ i ]->to->textSymbol );
  }

  fprintf( f, "}\n" );

  fclose( f );
  free( p );
}
```

Figure 3.6: The GraphViz DOT file output function

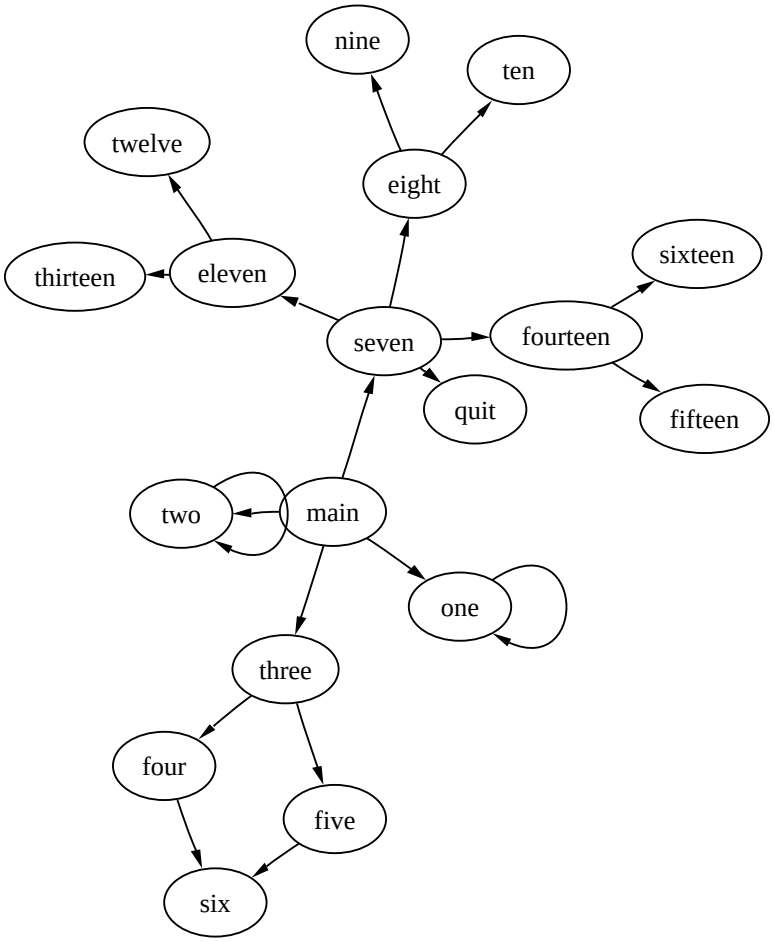

Figure 3.7: A GraphViz DOT file generated by `rtprof` and laid out using `neato`

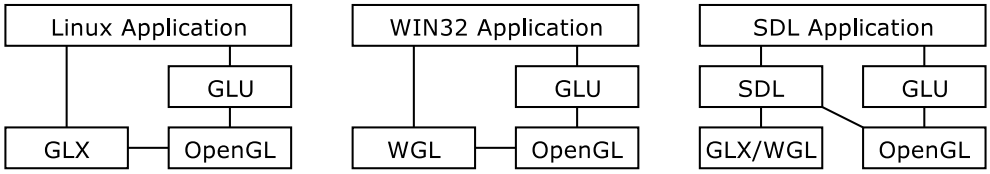

Figure 3.8: How SDL fits in to the traditional OpenGL architecture

Fortunately there is a library named *SDL*[27], short for Simple Directmedia Layer. SDL is essentially a comprehensive abstraction layer for many of the systems involved in computer games development — the driving force behind its development. The underlying premise of SDL is that if some application is developed with portability in mind on top of the library, then migration to foreign platforms will involve the minimum hassle. Currently there are certified ports for Linux, Windows, BeOS and Mac OS but there are also unofficial ports for a range of other platforms including Solaris, IRIX, FreeBSD and indirectly for the Playstation 2 games console. SDL provides a platform transparent means of managing OpenGL contexts (figure 3.8) as well as lower level video operations such as colour depth selection and video mode switching. Additionally SDL delivers a range of peripheral subsystems including but not limited to, input, sound, threading and portability utilities. From the perspective of the real time profiling tool the most useful of these is the input subsystem. This provides both asynchronous event based input handling mechanisms and synchronous input polling mechanisms for a variety of hardware devices. Figure 3.9 shows the code required to create an OpenGL context on any platform which SDL supports.

```
if( SDL_Init( SDL_INIT_VIDEO ) < 0 ) //initialise SDL
  return;

if( SDL_SetVideoMode( 640, 480, 0, SDL_OPENGL ) == NULL )
{
  SDL_Quit( );
  return;
}
```

Figure 3.9: Creating an OpenGL context using SDL

3.2.2.1 Geometry

A graph consists of *edges* and *vertices*. In order to represent the graph pictorially, glyphs must be drawn to the screen which represent each of these elements. In the case of the real time profiling tool, these glyphs must also incorporate elements of the generated profile.

The higher level functionality library GLU provides useful facilities here. Instead of rendering the glyphs manually triangle by triangle, GLU has functions to draw a number of common simple geometric primitives including cylinders, discs and spheres. All of these are useful in constructing the geometry to represent graph elements. In terms of geometry it was decided that graph nodes should be represented by spheres while graph edges are represented by extruded arrows.

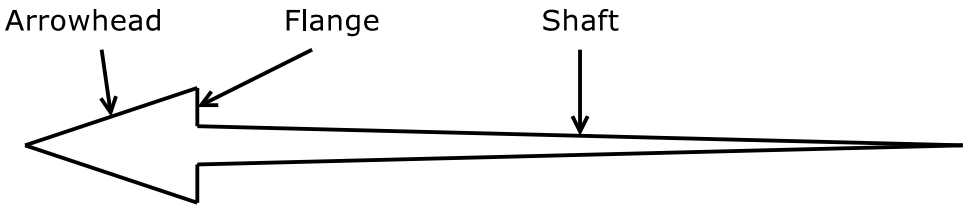

Figure 3.10: The anatomy of a normal graph edge

Concentrating solely on geometry, graph nodes are rather simple to add to the scene from a programmatic point of view since all that is required is a single function call to `gluSphere(...)`. Edges are more complex. In the general case, an edge as rendered by the analysis program has three components; the shaft, flange and arrowhead (figure 3.10). In this case the function call `gluCylinder(...)` is used to add the shaft and arrowhead to the scene whereas the call `gluDisk(...)` is used for the flange. If these were the only graph primitives required then programming the required geometric figures would be exceptionally straightforward. Unfortunately there is a special edge case — that of the recursive edge.

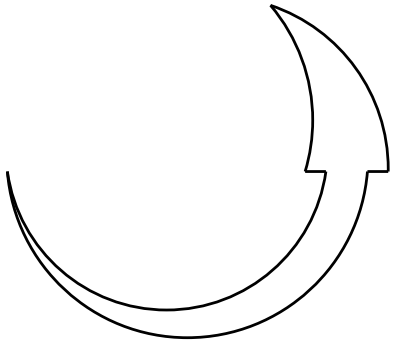

Figure 3.11: A recursive edge

This edge must have both end points connected to the same node and thus necessarily is curved. Since the edge must come back on itself the obvious shape is circular or elliptical (figure 3.11). This figure has the same anatomy as the regular edge, but it is perturbed around the circumference of a circle. There is no GLU primitive available to draw tori (a donut shape), much less partial tori with variant diameters as would be useful here. A library named *GLE*[28] is

available which is intended for the generation of tubing and extrusions as required here, however the benefits reaped by adding another library dependency to the analysis program are outweighed by its prohibitively low prevalence on most computer systems. Furthermore manual generation of the graph primitive on a per polygon basis is not excessively complex.

$$x(t) = \sin(2\pi t)$$

$$y(t) = \cos(2\pi t)$$

Figure 3.12: Parametric equation of a circle (radian measure)

For this the parametric equation of a circle (figure 3.12) is used in two instances. In the first the points along the length of the edge must be drawn around a circle whose centre is one edge radius away from the node centre and secondly these points are rotated about the length of the edge in order to give the primitive volume. These uses are shown in figure 3.13. On a programmatic level, the `GL_TRIANGLE_STRIP` OpenGL facility is used to generate the constituent parts of the recursive edge. The edge is segmented along its length such that each segment is approximated by a circular ribbon of triangles. This continues along the edge, peturbing the strips to form the shaft, flange and arrowhead as necessary. Obviously the more segments used increases the accuracy of the figure rendered, but care must be taken such that the overall polygon count of the primitive is not detrimentally large.

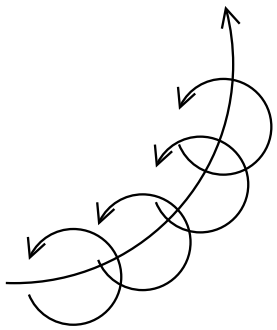

Figure 3.13: The two ways in which the parametric circle equation is used in drawing recursive edges

One of the visualisation channels identified in figure 2.7 is the scale of the glyphs rendered. In the case of graph nodes, this is a simple case of applying a scaling factor in every dimension. For edges however, the overall length of the edge must not change in relation to its scale, so edge scaling is only performed in the diameter of the edge.

3.2.2.2 Colour map and alpha values

The colour of the glyphs is easy to set using OpenGL, requiring only a function call to `glColor(...)`. In order to choose a colour however, a colour map is required to convert from scalar value to some colour space. Generation of a colour map is possible by mapping curves to parameters in a colour space (i.e. RGB or HSV). In practice however, the success of a colour map is a largely aesthetic variable governed by subjectivity rather than mathematics. In the case of the analysis program colour maps are generated in the RGB colour space by listing an arbitrary number of discrete colours then linearly interpolating between them. The discrete colours themselves are distributed evenly throughout the scalar space. Figure 3.14 shows a colour map generated using this method.

Figure 3.14: An example colour map blending from $(0.0, 0.0, 0.0)$ to $(0.38, 0.0, 0.0)$ to $(0.0, 0.5, 0.61)$ to $(0.0, 1.0, 0.0)$

Appendix D shows the implementation of this linear interpolation based colour map generator.

The transparency of each glyph is another mappable variable. Alpha blending as it is known, involves combining objects with the existing framebuffer through the use of some defined blending function. This is shown in figure 3.15 where *Source* is the pixel value of the object to be combined and *Destination* is the corresponding pixel value in the framebuffer. The factors $< srcBlend >$ and $< dstBlend >$ are defined when specifying the blending function. In the case of the visualisation this is `glBlendFunc( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA )` giving rise to the blending function $Object.ObjectAlpha + Framebuffer.(1 - ObjectAlpha)$. This is what is generally recognised as transparency where each pixel's alpha channel gives a weighting of the pixel value used in combination.

$$Source. < srcBlend > + Destination. < dstBlend >$$

Figure 3.15: OpenGL blending function

OpenGL is a Z-buffer based graphics library. A Z-buffer is a buffer the same size as the framebuffer which stores a depth value for every pixel relative to the camera. When a pixel is written to the screen it is first compared for depth

with the Z-buffer, and skipped if it is obscured by a corresponding existing pixel. Consequently, the order in which objects are added to the scene is unimportant. When alpha blending is employed however, for a consistent scene objects must be drawn in back to front order since one object cannot be alpha blended with another if it has not first been drawn to the framebuffer. The net result of this is that currently scenes rendered by the analysis program are not consistent with respect to alpha blending. In order to solve this problem a significant change must be made to the drawing architecture such that it is possible for nodes and edges to be drawn interleaved, instead of the nodes first approach currently used. The glyphs must then be depth sorted and added to the scene in back to front order. The author's favourite algorithm for depth sorting is *radix sort*[29] since it is highly efficient at sorting fields which are already partially sorted. However any efficient sorting algorithm can be applied.

3.2.2.3 Text

No matter how pleasing the rendering of the graph glyphs, the rendered call graph is virtually useless unless some identifcation is associated with each graph node. The visualisation must annotate each graph node with the name of the function it represents. Neither OpenGL or GLU provides any facilities for rendering text to a GL context. The GLX extension provides a function named `glXUseXFont(...)` which provides a means for converting between an X bitmap font and an equivalent list of OpenGL commands, known as a call list. This approach is rather unportable however and oddly does not work on Informatics department machines. The *GLUT*[30] (GL Utility Toolkit) library is a lightweight window system independent toolkit based on OpenGL. Conceptually it is slightly higher level than GLU in terms of functionality. A subset of its functionality is to render both bitmap and stroke based fonts from a selection of GLUT fonts. The visualisation uses the GLUT font `GLUT_BITMAP_HELVETICA_12`.

3.2.2.4 Layout algorithm

Peter Eades'[18] generalised graph layout algorithm is outlined in figure 3.16. This is the parent to all current force directed placement layout algorithms and consequently is probably to most simple and easy to implement. With some modification it is also relatively efficient.

This is the algorithm used by the visualisation to layout the call graph, with some small differences. In place of the $C1$ constant, the algorithm implemented uses a constant < 1 which is placed infront of the *repulse* function in addition to the *attract* function. I found this tended to reduce 'bunching' whereby complex subgraphs resolve such that node density was somewhat variant across the entire

The repellant force between any two vertices is given by:

$$repulse(d) = \frac{C3}{d^2}$$

The attractive force between any two nodes connected by an edge is given by:

$$attract(d) = C1. \ln(\frac{d}{C2})$$

Where d is the distance between the vertices.

1. place vertices of graph G in random locations
2. repeat M times
   (a) calculate the force on each vertex $\vec{F}$ using the above functions
   (b) move the vertex $C4.\vec{F}$

Where values of $C1 = 2.0$, $C2 = 1.0$, $C3 = 1.0$ and $C4 = 0.1$ are thought appropriate for most graphs. A value of $M = 100$ should result in a graph layout tending towards a minimal state for modestly large graphs.

Figure 3.16: Peter Eades' FDP graph layout algorithm

graph. Instead of iterating to a minimal state, the algorithm implemented performs only a single iteration. Originally this algorithm was intended to perform graph layout statically (figure 3.17), but in this instance the layout is performed dynamically as the analysis program runs. Consequently, while computing power has moved on since 1984, performing too many iterations of the layout algorithm quickly reduces the framerate of the overall visualisation. For the time being the minimum number of iterations is performed per frame.

"An implementation of the algorithm on a VAX11/780 (in unoptimsed pascal) is fast (in fact, I/O bound) if the number of vertices in G is less than 30."

Figure 3.17: Peter Eades' 1984 effciency assessment of the algorithm

A further modification of the algorithm is in the initial placement of the nodes. Originally the algorithm specifies that the nodes should be placed entirely randomly (within some numerical bounds obviously). This is acceptable for a static layout run, but in the case of the call graph visualisation new nodes may be added

to the graph as it is laid out. In this case a randomly placed node has a low probability of being initially placed within the locality of its eventual layout position. Furthermore, such a placement is likely to be disruptive to the remainder of the graph layout, reducing the overall effectiveness of the visualisation. Since graph nodes are added to the profile in some order it is intuitive to place new nodes within some local area around the graph node with which it is connected by an edge. Placing new child nodes such that they are coincident with their parent nodes is not a viable solution since the d term in the $repulse$ function is zero and evaluation would result in a divide by zero error. This may be solved by creating a special case whereby the result is a large positive number approximating infinity, but the force is still unresolvable since the vector between the nodes is undefined. A suitable vector may be derived by computing the direction from the centre of mass of the graph to the initial layout position of the new node, but the computational cost of calculating an ever changing centre of mass is prohibitive in an already expensive algorithm.

Instead of this approach therefore, the random initial placement is retained albeit with modifications. When a node is added to the call graph its placement is determined first by the position of the parent node then by a random distribution within some small bound. New nodes are therefore added within a reasonably local area with respect to their parent nodes, and will cause less disturbance than had they been placed in an entirely random fashion. Furthermore the random number generator used is seeded by the function's symbol name so that separate (albeit similar) runs of the same program are laid out in a pseudo-deterministic manner.

Recursive edges are a special case in the layout algorithm. Each recursive edge is associated with a dummy graph node that is never rendered. This dummy node serves purely to layout the recursive edge and is not involved in the profiling in any way.

3.3 Optimisation

An important goal stated in section 1 was that the visualisation tool should be implemented efficiently enough that it remained sufficiently interactive to remain usable. When developing any heavily CPU bound application, optimal programming must be considered at all times. More specifically this means avoiding suboptimal programming practices such as loop invariant variables and passing by value when you can pass by reference. On a higher level there are more global optimisation strategies that can be applied to specific problems.

```
subDiv = (int)( ( ( LOD_DIST - distance ) / LOD_DIST ) * (float)NODE_SUBDIV );

if( subDiv < 3 )
  subDiv = 3;

subDiv = (int)( (float)subDiv * ( 1.0f + ( size / MAX_NODE_SIZE ) ) );
```

Figure 3.18: LOD function for graph nodes

3.3.1 Level of detail

In any polygon based modelling system the overall rendering efficiency is governed heavily by the number of polygons in the scene. Clearly therefore, it is advisable to keep this quantity to a minimum. In terms of the glyphs themselves this means using the minimum number of polygons possible while maintaining the desired level of detail.

This quantity may be reduced further however, by noticing that if two identical glyphs are displayed at different scalings or view distances, the one that occupies fewer pixels on the framebuffer could be represented by fewer polygons and still maintain the same apparent level of detail. This is because framebuffer resolution is finite — each pixel is just a sample of the actual model behind the visualisation. If a model is projected on to a 2 dimensional framebuffer where it occupies only a single pixel, a single polygon representation of the model will result in the same projection as that of a model that contains many thousands of polygons. Varying the number of polygons used to approximate some geometric structure is called level of detail or LOD for short.

In the visualisation this is acheived by subdividing the mesh of the glyphs more if the distance to the object from the camera is small. The LOD function employed for graph nodes is shown in figure 3.18. The first line specifies that node sub divisions is a function of distance so that at zero distance the primtive is fully sub divided, but at further distances it falls to zero. The last line then specifies that graph nodes that are scaled larger are subdivided more as they will occupy more pixels in framebuffer space.

Graph edges are decimated in much the same manner except for a special case whether the flange and arrowhead components of the edge are not drawn at all once a specific distance threshold is surpassed. Instead an extended shaft is rendered such that the difference is nearly imperceptible at the threshold level used. This helps to reduce the number of polygons by a factor of around ten when an edge is at its furthest from the camera.

3.3.2 Layout algorithm efficiency

The Eades graph layout algorithm is efficient apart from one crucial flaw, the ln term in the *attract* function. This is computationally expensive to evaluate, especially when it is performed for every graph edge, multiple times a second. The ln term does not need to be evaluated with a great deal of accuracy however. As long as the resultant function is approximately logarithmic in terms of growth it will suffice for the purposes of the *attract* function.

Let f be inifinitely differentiable in $\{x \in \mathbb{R} : -R < x < R\} = (-R, R)$. If

$$|f^{(n)}(x)| < C.B^n \textbf{ or } |f^{(n)}(x)| < Cn!R^{-n}$$

for all n in N and $x \in (-R, R)$, where C and B are constants and R is a radius of convergence, then

$$f(x) = \sum_{n=0}^{\infty} a_n x^n \textbf{ for } |x| < R, \textbf{ where } a_n = f^{(n)}(0)/n!$$

Figure 3.19: Taylor-Maclaurin Theorem

Figure 3.19 describes the Taylor-Maclaurin theorem. This provides a means of expressing a function (within some confines) as a power series. The key concept here with respect to computer science is that by summing the power series to a finite number instead of infinity, a function is produced that approximates the original yet is somewhat less computationally intensive.

$$\ln x = 2\left[\left(\frac{x-1}{x+1}\right) + \frac{1}{3}\left(\frac{x-1}{x+1}\right)^3 + \frac{1}{5}\left(\frac{x-1}{x+1}\right)^5 + \cdots + \frac{1}{2n+1}\left(\frac{x-1}{x+1}\right)^{2n+1}\right] \textbf{ for } x \geq 0$$

Figure 3.20: Taylor series for $\ln x$

In place of the ln term in the *attract* function, a Taylor series is used, as seen in figure 3.20. Originally a function was written to calculate $\ln x$ using a Taylor series to an arbitrary order. After some experimentation it was found that the first order power series for $\ln x$ was sufficiently accurate to replace the term. This is $\frac{2x-1}{x+1}$ — significantly cheaper to compute than the $\ln x$ implementation in most C libraries. This is clearly an approximation (figure 3.21) since it tends to 2 relatively quickly.

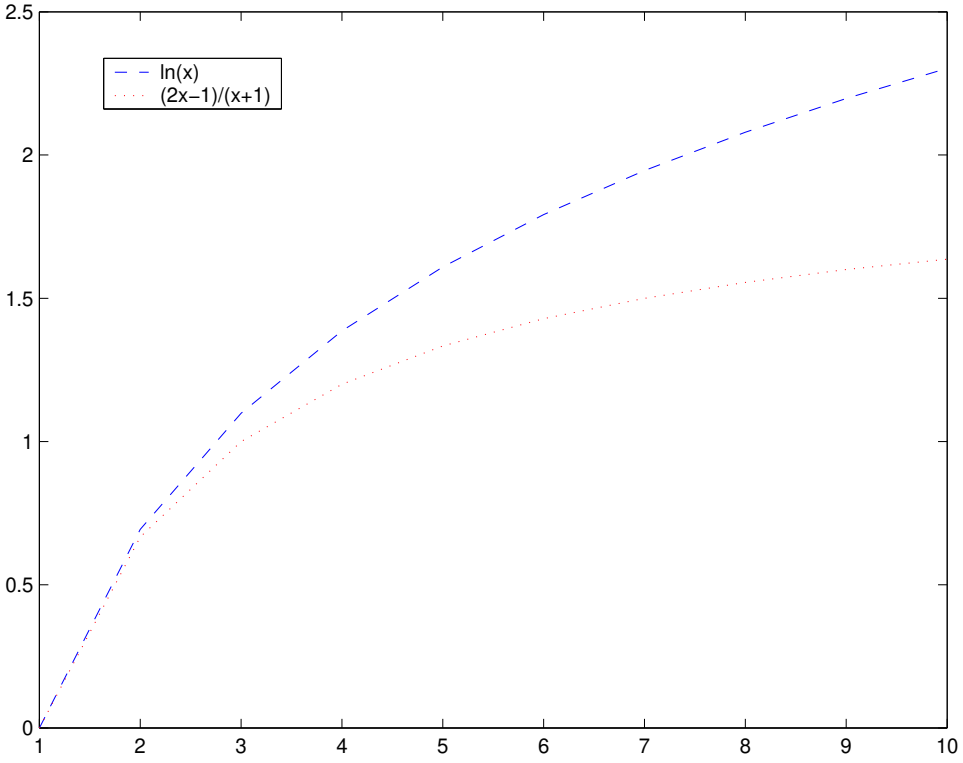

Figure 3.21: The growth of $\ln x$ versus $\frac{2x-1}{x+1}$

4. Evaluation

Completing the implementation of any piece of software does not have any bearing on whether or not it has succeeded at its intended purpose. This is especially true for a visualisation.

The eventual visualisation operates as follows:

- The size of each graph node represents the amount of local time spent in the function it represents. This is done proportionately so that two nodes of the same size have accumulated the same quantity of run time over the program's execution.

- The colour of each graph node represents the number of times it has been called. This is also proportionate. A colour map is used from blue through to purple through to red. For example a node which is blue has been called few times in relation to the total number of function calls made whereas a node which is red has been called many times.

- The colour of each graph edge represents the number of times one function called another. The colour uses the same colour map as with the graph nodes.

- The transparency of graph nodes and edges indicates the activity of a function. A largely transparent graph node represents a function which has not been active for some time as opposed to a totally opaque graph node which has been active very recently.

- Both graph nodes and edges are rendered with a dark shell around them if they are currently active - in other words if they are on the call stack.

- Each graph node is annotated to the right with the name of the function it represents. This text is distance culled to prevent both over-crowding and the associated computational cost of rendering the text.

Edge scaling was not used in the final visualisation primarily because there is only a single continuous variable (number of calls) associated with every edge. Furthermore, given the choice between colour and scaling, colour is far more effective at conveying the information. Figures 4.1 and 4.2 show some screenshots of `rtprof` visualising non-trivial programs.

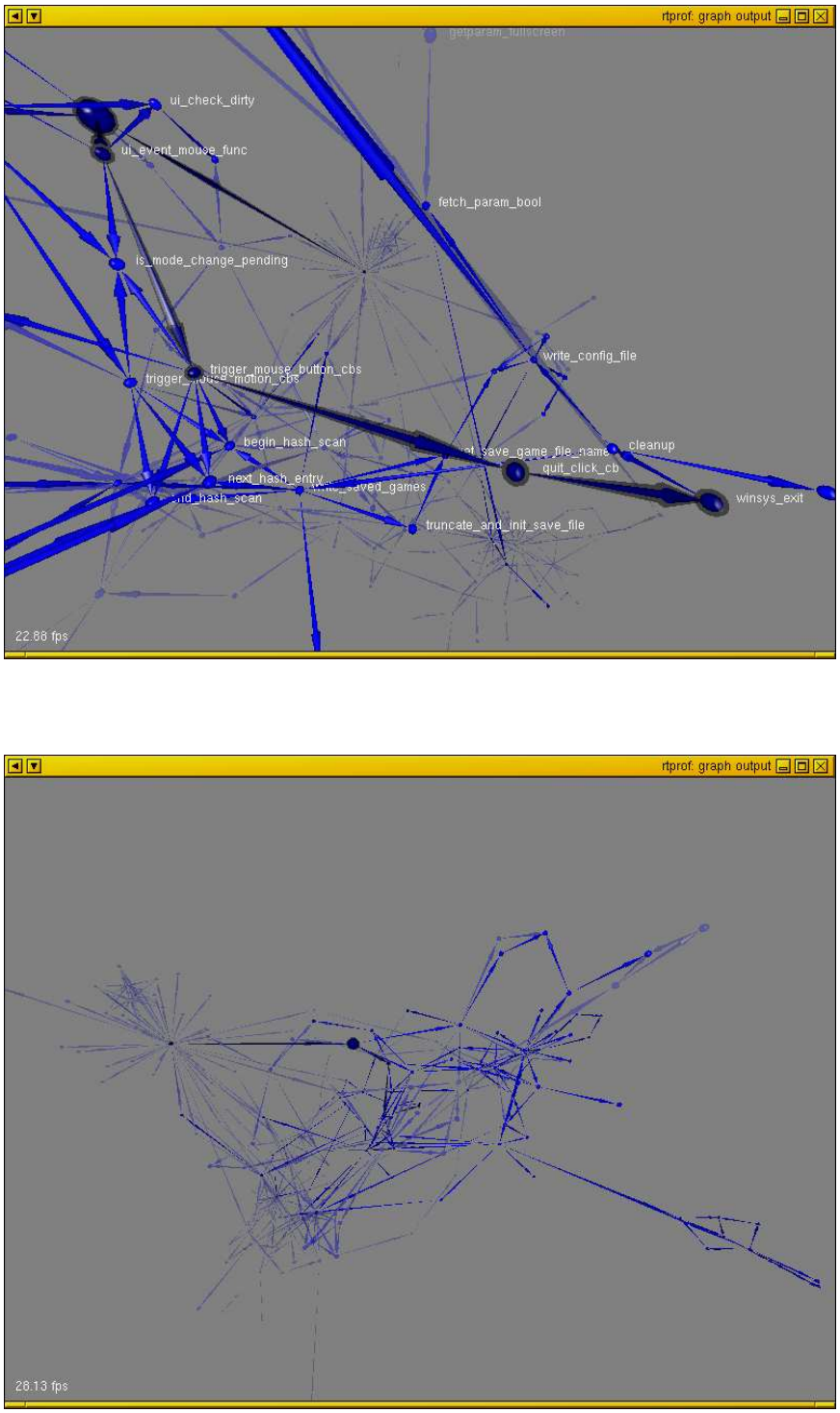

Figure 4.1: A visualisation of the computer game 'Tuxracer'

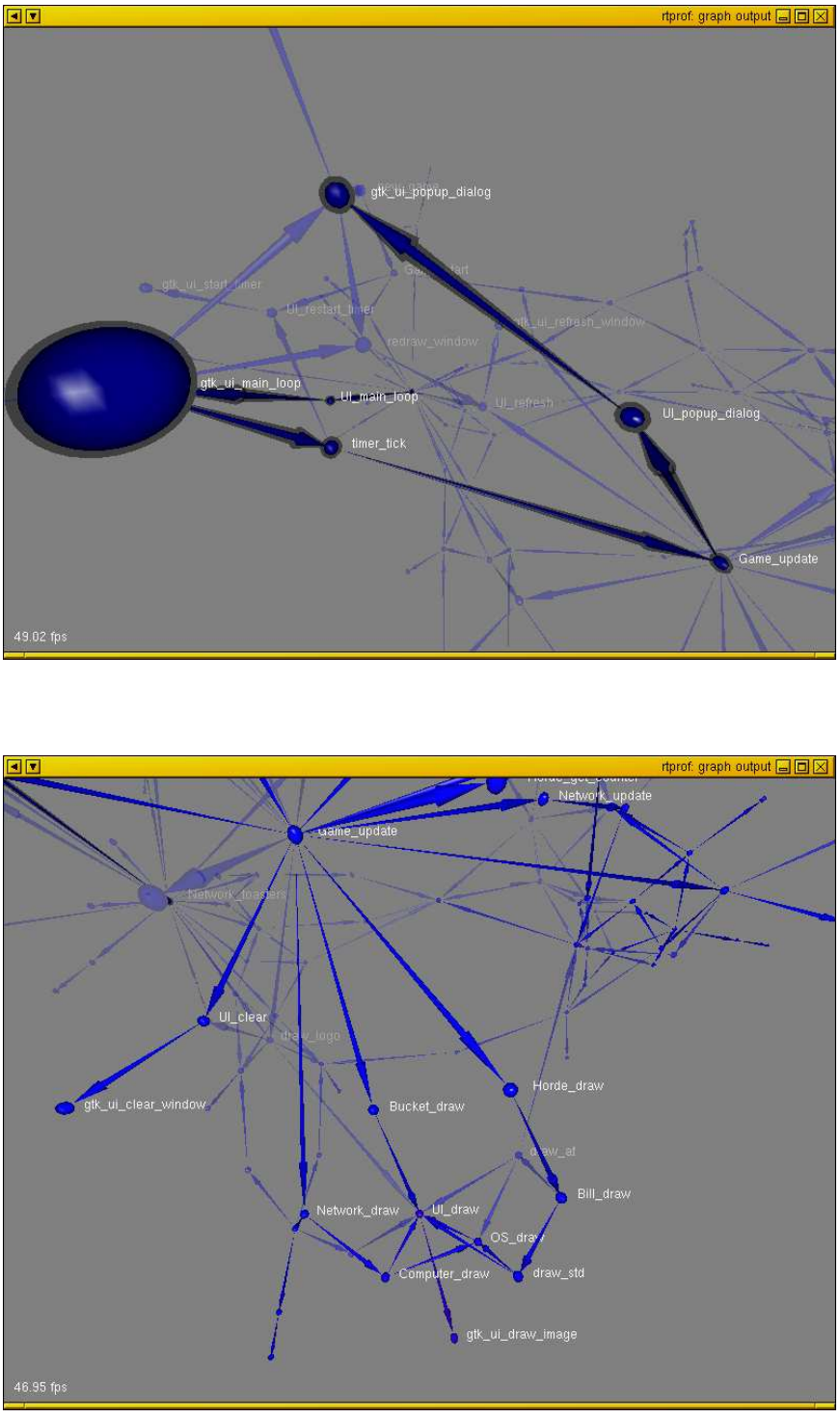

Figure 4.2: A visualisation of the computer game ‘xbill’

4.1 Usage study

With any visualisation there is a subjective element governing whether or not the visualisation effectively represents the underlying data as intended. A large part of the assessment of the success of the project involves evaluating whether or not this goal has been attained. In order to test for this a small usage study was performed utilising a number of undergraduates and external test subjects.

This usage study consisted of a questionnaire in which respondents answered a number of specific queries regarding the visualisation of a simple pathological test program designed to illustrate the function and capability of the real time profiling tool. The static call graph for this program was shown previously in figure 3.7. This graph shows recursive, convergent and tree-like function call formations. What it doesn't show however is more specific elements of the profile such as the number of calls or time elapsed in a function. The questionnaire concentrated mainly on these elements of the profile. In addition the latter section of the questionnaire asked more general questions regarding the opinion of the subject with respect to the profiling tool. The questionnaire took the form of an online web form with a results processing script written in *PHP*[31] which stored results in a *MySQL*[32] table. This meant that the questionnaire could be completed without the use of an Informatics computer system thus enlarging the available sample space. Figures 4.3, 4.4, 4.5, 4.6 and 4.7 represent what each respondent should be looking at when answering the associated questions:

Question 1. *Which function is called the most times?* The correct answer to this question was six.
100% of the respondents chose the correct answer.

Question 2. *Which function(s) call the function six?* The correct answer to this question was four and five.
100% of the respondents chose the correct answer.

Question 3. *Of these function(s) which makes the most calls to six?* The correct answer to this question was five.
100% of the respondents chose the correct answer.

Question 4. *A recursive function is one which calls itself. Identify one recursive function using the visualisation.* The correct answer to this question was either one or two.
100% of the respondents chose the correct answer.

The answers to questions 2 and 4 suggest that the program topology is conveyed well by the visualisation.

Question 5. *Which function is the most deeply recursive? i.e. which function calls itself*

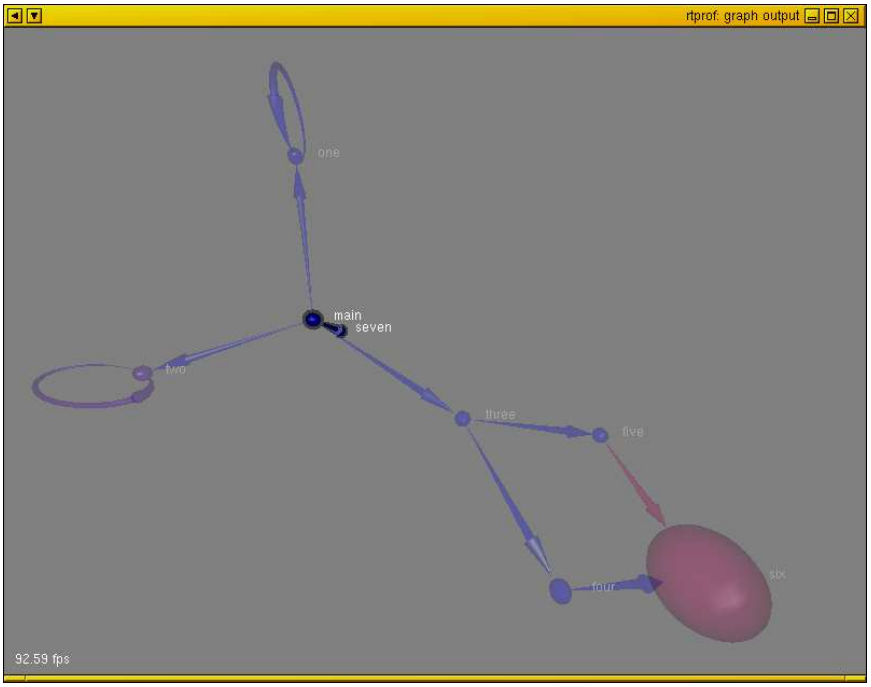

Figure 4.3: The visualisation relevant to questions 1-6

the most times? The correct answer to this question was two.
100% of the respondents chose the correct answer.

The answers to questions 1, 3 and 5 suggest that the use of colour has been very effective in conveying information; in this case number of calls.

Question 6. *In which function has the program spent most time?* The correct answer to this question was six.
60% of the respondents chose the correct answer.
24% of the respondents thought the visualisation was ambiguous.
8% of the respondents answered main.
8% of the respondents answered two.

This was probably the most interestingly answered question. The questionnaire was presented in a form where no prior knowledge of profiling or call graphs was required. In an effort to simplify the explanations given, the distinction between total and local time (figure 2.4) was not covered. It is conceivable that those with a computer science background will realise the function with the greatest total time is main, yet the visualisation shows six as accumulating the most (local) time. Having deduced this, the answer given is either ambiguity or main. Clearly this explanation is purely

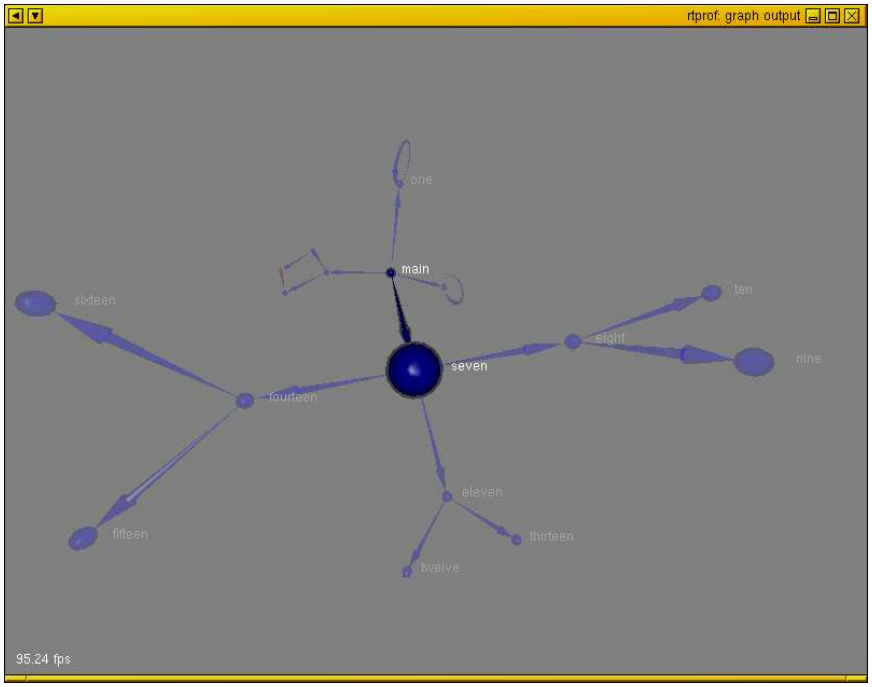

Figure 4.4: The visualisation relevant to questions 7-8

conjecture, but it seems the likely case. There is no concrete explanation as to why one respondent answered two.

At this point the respondent was asked to press six keys while the test program was focused. This had the effect of creating six leaf functions in a tree structure beneath the function seven.

Question 7. *How many new functions have been called?* The correct answer to this question was 9.
68% of the respondents chose the correct answer.
16% of the respondents answered 6.
8% of the respondents answered 8.
8% of the respondents thought the visualisation was ambiguous.

In order to answer this question the respondent must have known how many functions had been called before pressing the keys. Some of the respondents incorrectly equated the number of key presses to the number of functions called while others apparently miscounted. One respondent commented that they had not counted how many functions had been called before pressing the keys so answering this question was not possible. In hindsight there is little information to be gained from the results of this question and it should not have been posed.

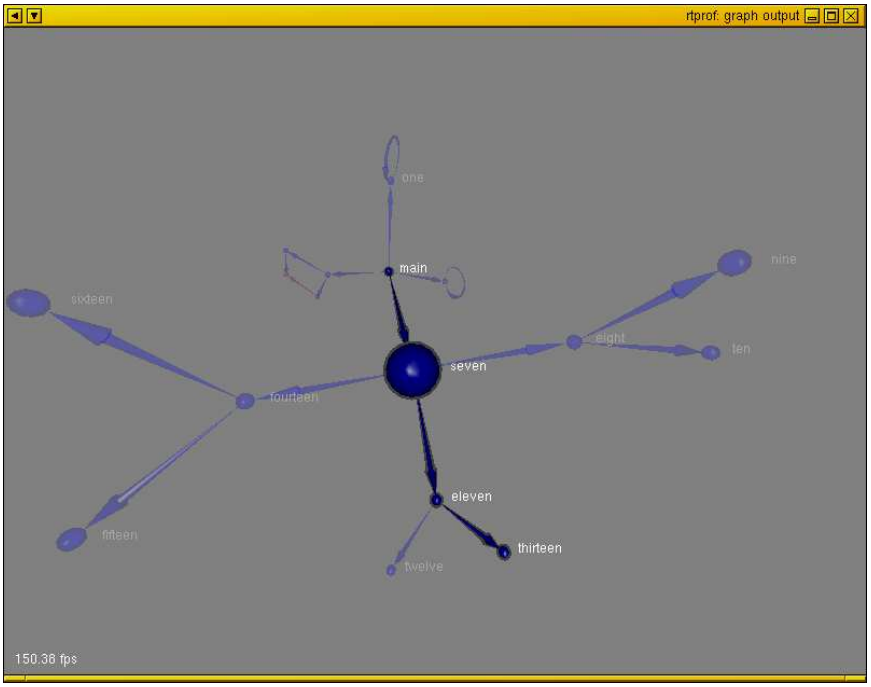

Figure 4.5: The visualisation relevant to question 9

Question 8. *Which function is currently executing? In other words which function is on top of the call stack?* The correct answer to this question was seven.
84% of the respondents chose the correct answer.
16% of the respondents didn't understand the question.

Question 9. *Change to the test program and press the key i. What is the state of the call stack the moment after you press this key?* The correct answer to this question was main→seven→eleven→thirteen.
84% of the respondents chose the correct answer.
8% of the respondents answered main→seven.
8% of the respondents didn't understand the question.

The single respondent who answered main→seven probably did not observe the visualisation momentarily after pressing the key. There was only a two second window in which to observe the effects. In hindsight this delay should have been slightly longer.

Question 10. *Which function has now been called the most times?* The correct answer to this question was six.
54% of the respondents chose the correct answer.
30% of the respondents answered seven.
16% of the respondents thought the visualisation was ambiguous.

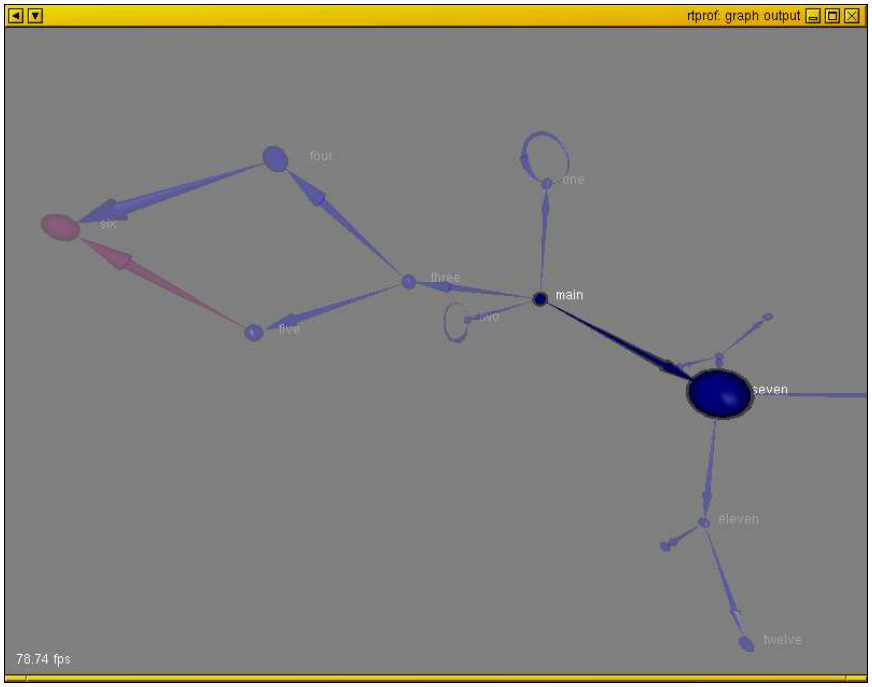

Figure 4.6: The visualisation relevant to questions 10-11

This was a trick question to some extent since it is the same as question one — and has the same answer. Surprisingly 30% of the respondents thought the most called function was now seven, perhaps by confusing the activity marker with the number of calls colour indication.

Question 11. *In which function has the program now spent most time?* The correct answer to this question was seven.
76% of the respondents chose the correct answer.
8% of the respondents answered main.
8% of the respondents answered six.
8% of the respondents thought the visualisation was ambiguous.

This response was slightly surprising in that a greater proportion of the respondents answered correctly than the previous question 6. The respondent that answered main presumably assumed the representation of total time as with question 6.

Question 12. *What is the name of the function that last executed before the test program closed?* The correct answer to this question was 'quit'.
100% of the respondents chose the correct answer.

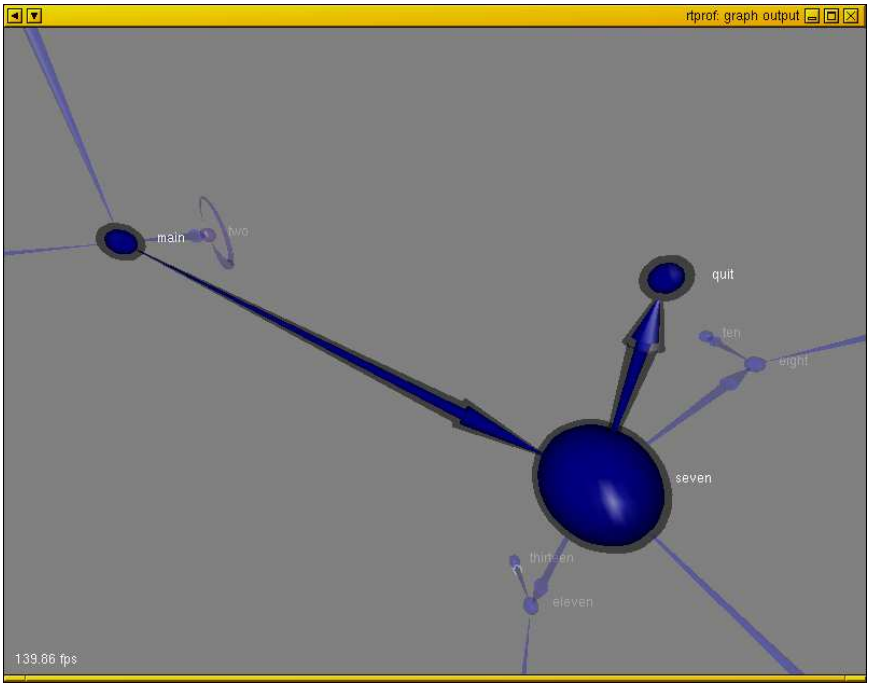

Figure 4.7: The visualisation relevant to question 12

The last two questions on the questionnaire asked were more general and answered on a sliding scale. Firstly *Do you consider the real time profiling tool to be useful?* The respondents answered with an average of 78% usefulness. This result was satisfyingly high with most of the respondents answering 80% or higher. Secondly the respondents were asked *Regardless of your opinion of the usefulness of the profiling tool, do you consider it to be 'cool'?*. The response to this question was unanimously 100%. It appears if nothing else the profiling tool has succeeded in impressing people!

Broadly speaking the usage study has provided some good evidence of the profiling tool's success. All of the questions posed were answered correctly by greater than 50% of the respondents, while the average correct response was somewhat higher. Section 2.2.2 discussed how two dimensional projections of three dimensional space will tend to distort the apparent dimensions of the objects themselves. The usage study has highlighted this through the confusion in questions regarding time. It is not clear whether this is solely due to the indistinction of local and total time however, so this area deserves some more investigation. The raw data for the usage study may be found in appendix E.

4.2 Goals achieved

Section 1.3 outlined the high level objectives of the real time profiling tool. These aims have been accomplished to a varying degree, but in general the project has been a resounding success.

- *The tool should be as portable as is realistically possible.* Issues of portability have been conscientiously considered throughout the development of the profiling tool. It is predominantly portable, with one significant caveat — currently both the instrumentation library and analysis program rely heavily on the GNU tool chain, specifically GCC and binutils. Realistically this can never be replaced by entirely portable tools, since compilers and low level binary format manipulation libraries are necessarily somewhat architecture dependent. Providing the target platform has a compiler with some means to instrument functions and resolve symbols however, it is possible to modify the profiling tool such that platform specific dependencies are exchanged at compile time by the C preprocessor.

- *The tool should operate on a network transparent basis.* The tool is entirely network transparent with subtle provisos regarding the platform of each machine involved in the profiling. Solutions for these annoyances are discussed in section 4.4.3.

- *The tool should present the data in an intuitive manner that is easy for humans to interpret.* This is definitely the most difficult goal to reason about. Generally speaking the usage study indicates that this goal has been achieved, but to what degree is subject to debate.

- *The tool should be easy to use.* The user interface for the visualisation is not as intuitive as it could be. In the current iteration, the user must memorise six keys and their respective functions. Navigating the graph can also be difficult. More specifically finding the area of a program in which the interest lies is not autonomous — the user must manually move the viewpoint to find specific functions. As far as this preliminary version stands it is sufficiently usable to be useful, but there is a great deal of improvement that could be made in this area. Section 4.4.4 discusses improvements that could be made to the user interface.

- *The tool should be implemented sufficiently optimally to maintain interactivity on relatively modest hardware.* To some extent this is difficult to evaluate since the definition of 'modest hardware' depends on the current marketplace and architecture advances. Nevertheless the author's 866Mhz laptop with reasonable graphics hardware manages to maintain interactivity (6 frames per second) for client programs with up to around 500 functions. Clearly this will be somewhat reduced on systems relying on software ren-

dering. Ultimately some data management is required. It is unlikely that a user wishes to examine as mamy as 500 functions at once, so it would be prudent to only render an interesting subgraph of the call graph in order to boost interactivity.

- *The tool should be reasonably configurable.* The only means of configuration available currently are options provided on the command line. These allow the user to specify whether or not to use the call graph renderer at all, where and if to write the GraphViz DOT file and which type of socket to open. Section 4.4.7 discusses configuration issues.
- *The tool must be able to profile programs written in the C language at a minimum. Such programs should not need any modification in order to be profiled.* The tool has been shown to work with many programs written in C. However it is difficult to *prove* that the profiler will work with *every* C program. Nevertheless the instrumentation library was designed from the outset to have a low impact on the client program and thus is unlikely to cause problems. In contrast an accepted profiler, FunctionCheck implements must of its functionality directly from the instrumentation functions, rather than in a separate process. The C program does not need direct modification to be profiled, although it must be compiled specially. This is necessary with any profiling tool without resorting to virtualisation.

4.3 Impact on the client program

"The procedure of measurement has an essential influence on the conditions on which the very definition of the physical quantities in question rests..."

Figure 4.8: Niels Bohr's 1935 postulate

In 1935 Niels Bohr famously proclaimed that it is impossible to measure something without affecting it in some way (figure 4.8). His comments referred to quantum mechanics, but they nevertheless still apply to many other things, including the real time profiling tool. In this case the profiling tool has a detrimental effect on the performance of the client program. There are three variables that affect this. Firstly and most importantly, the rate at which the analysis program consumes events. Secondly, the rate at which events are transferred between the instrumentation library and the analysis program. Finally, the rate at which the client program generates events.

Ideally the rate at which the analysis program consumes events is both maximal and constant. A constant rate is important so that the measurement penalty

incurred by consuming each event does not affect the real times measured by the instrumentation library inconsistently. The event transfer rate depends on the socket type and the communication channel used. With current CPU speeds, at least a 100Mbit/s network interface is necessary for event transfer rate to have a minimal effect on the profile produced. The rate at which the events are produced depends entirely on the client program. Assuming the socket queue is a constant size, the client program is able to produce events as often as it likes, minus a penalty for each instrumentation function.

Due to the number of variables involved, the effect that profiling a program has on its speed is not easy to define in general. For a program with a very high rate of function calls, it may run as slow as 10% of the unprofiled speed. Similarly a program with a very low rate of function calls it likely to run at a speed very close to that of the same program without profiling.

4.4 Potential improvements

The real time profiling tool is already useful and usable to an extent, but the developmental possibilities that could be explored have by no means been exhausted.

4.4.1 Extension to other languages

The profiling tool has been implemented in C and is intended for profiling programs written in C, but there is essentially no reason why any procedural language for which there is a GCC frontend cannot be profiled. This claim remains unproven however. Object orientated languages can be profiled using the profiling tool but the resultant profile is unlikely to be very useful. This is especially apparent with the C++ frontend of GCC. The nature of object orientated programming implies a great deal of compiler transformation. GCC introduces a large quantity of name space mangling into C++ programs such that while a profile for a C++ program may be correct, getting any useful information from it in relation to its source code is close to impossible. This is not a problem specific to the real time profiling tool, gdb and other GNU tools are known to have serious problems with C++ based programs.

Without the realms of GCC, extension to other languages is certainly possible assuming the compiler used provides some means to instrument a program at a source code level. Microsoft compilers certainly have the ability to instrument functions, though this has not been explored in any great depth.

4.4.2 Profiling improvements

4.4.2.1 Instantaneous profiling

The profile as generated is constructed cumulatively from when the client program starts until the point at which it exits. From the perspective of a real time profiling tool, this is of dubious use. For example, a program used to visualise terrain might do a large quantity of preprocessing on some input mapping data, then visualise the result of this processing. Depending on how long the visualisation is run for, with a traditional profiler such as gprof, the preprocessing stage of the program will dominate the profile. This is also the case with the real time profiling tool in its current state. If the purpose of profiling this program is to improve the efficiency of the visualisation stage, this data is less accessible from the generated profile since it is overshadowed by the preprocessing stage. This information is still present in the profile, but the resolution at which it is observable has been greatly reduced.

Instead of generating a profile for a programs entire run, a profile may be generated for some recent time window of the program. In this case the previous activities of a program (in the example the preprocessing stage) will have no bearing on the current state of the profile. However in order to do this, a cumulative approach to building the profile in no longer possible since data generated at the beginning of the profile must be removed as the time window advances. Alternatively a history of events could be kept and the profile continually regenerated as the time window advances. Obviously this is potentially relatively computationally intensive and may impact the overall interactivity of the profiling tool, so this time window would only be effective for relatively short periods of time. Nevertheless this is an interesting avenue to explore and is genuinely likely to increase the usefulness of the profiling tool.

4.4.2.2 Predictive profiling

The real time profiler is intended for examining the operation of processes which do not block frequently. This intention is reflected in the asynchronous paradigm used to collect profile data. The profile is only updated when a function is called or exits. In the case of a program that blocks, the profile stored in memory is inaccurate until such an event occurs since the function times are not incremented.

Although it is not the intention of the tool to be used to profile blocking processes, it is clearly a positive feature. When a process blocks, the analysis program no longer has reliable data regarding the status of time spent within a function since there are no events to be processed. However, the analysis program maintains its own real time clock and as such is able to infer that a function currently

marked as being on top of the call stack is still accumulating time if there is no corresponding function exit event queued for it.

In this state the analysis program can 'predict' the accumulation of the function's local time using its local clock up until the point a new event arrives. When this occurs the predicted time accumulation is rolled back and the profile is updated with the concrete data garnered from the event.

4.4.3 Network improvements

It is potentially dangerous to send raw binary data on a socket stream. This is because the endianess of the host architectures may differ and the receiving host may interpret the data in the wrong order. In the case of the current implementation of the network layer however, this consideration is not observed. Instead the event `struct` itself is sent on the socket. The ramifications of this are that both hosts involved in the profiling must be the same platform.

In addition to the endianess issues, the byte alignment and packaging of a `struct` are likely to differ between diverse platforms. To guarantee successful data transfer between two different platforms, an extra lower level must be implemented. This layer 'marshalls' binary data into a platform independant network centric form and 'unmarshalls' the data at the receiving end. In terms of implementation this is accomplished with the functions `htonl`, `htons`, `ntohl` and `ntohs`. These are used to convert between the host's byte ordering and a constant network ordering (most significant byte first). This extension is important to ensure the profiling tool's portability.

4.4.3.1 Efficiency improvements

Figure 3.3 shows the `struct` representing an event. With `struct` bit packing enabled, this has size 13 bytes. When profiling a program that might generate many thousands of events every second the amount of data transferred becomes a critical factor in the overall efficiency of the profiler. Of this 13 bytes, 8 are a time stamp which takes the form of the number of μ-seconds since the Epoch (00:00:00 UTC, January 1, 1970). This form of date storage is forecast to roll over in 2038. It is highly improbable that anybody using the profiling tool will need 35 years worth of date storage so there is clearly some room for improvement here. Two strategies present themselves. Instead of transferring time stamps in Epoch form, time stamps could be transfered that reflect the time since the client program was started. In this case the storage requirements could be greatly reduced to some realistic bound. Alternatively the time stamp could be replaced by another value that represented the time since the last event.

4.4.3.2 Client side symbol resolution

In order to get any meaningful information from the profiling tool the symbols from the binary file of the client program must be resolved into human readable function names. In the current implementation this is performed by the analysis program based from the binary filenames it is passed on the command line. In the case where the profiler is being used remotely, the machine running the analysis program does not necessarily have access to the binary file which is being profiled. Often this file needs to be copied to the visualisation machine for the symbols to be resolved. At best this is an inconvenient annoyance, but it is potentially resolvable. By moving the resolution stage to the instrumentation library and devising some means of communicating each function name to the analysis program, it no longer needs to be aware of the binary file at all. However this change flies in the face of the requirement to maintain a simple, lightweight instrumentation library. In this case the dependency on libbfd is moved from the analysis program to the instrumentation library.

4.4.4 UI improvements

4.4.4.1 Colour map scale

In the case of a simple colour map that blends from one colour to another, it is relatively clear what a specific colour represents in terms of scale. However for a more complex colour map it would be a useful addition to the user interface to have a colour map scale rendered to the screen. In this instance if there is any confusion regarding the meaning of some specific colouring, then it can be compared with the colour map scale.

4.4.4.2 Control instructions

A comment made in the usage study proclaimed that remembering which keys to use to navigate the call graph is difficult. A quick reference could be rendered to the screen either permanently or by popup, although in this case which key is used to activate this is a problem.

4.4.4.3 Function details

While the visualisation accurately reflects the profile as it is stored in memory, on its own it provides no numerical data. A possible user interface improvement is to display concrete textual data for the function or edge directly infront of

the viewer. In order to accomplish this some tracing mechanism is required. Tracing means to fire rays into the physical simulation and test for intersection with objects. Each object therefore requires volume. In the case of the graph nodes this is simply a case of testing for the intersection of a line (the trace) with a sphere (the graph node). Edges are harder since the cylindrical bounding volume is not necessarily axis aligned, but clearly still possible albeit slightly more computationally expensive. This test for intersection need not occur every frame or involve every graph glyph, so it can be made very efficient.

4.4.5 Scheduler improvements

Section 3.1.2.2 discussed the reasons behind using *non-blocking* sockets. With this approach the visualisation and the profile generation are scheduled internally within the analysis program. In the current implementation the scheduler is somewhat naïve. For every ten thousand events (assuming ten thousand are queued) the program will render a single visualisation frame. This is effectively a heavily weighted round robin scheduler. In practice this doesn't work very well — while the client program is still running interaction with the visualisation is rather choppy and unresponsive. A better scheduler might take real time into consideration and provide a loose guarantee for frames per second rendered.

4.4.6 Adaptive layout

The human vision system is a trichromatic integrator. The bandwidth of this system is typically accepted to be in the region of 70-80Hz. This means that beyond this data rate i.e. 70-80 frames per second, a human being is unable to appreciate very much difference in frame rate. In actual fact in order for movement to appear continuous only a frame rate of around 25 per second is necessary. Depending on the graphics hardware used, the visualisation often renders the call graph in excess of 80 frames per second, especially with smaller programs.

Instead of using CPU time to needlessly increase the frame rate, it could be diverted to the graph layout algorithm. In this case if the frame rate passed some upper threshold an extra iteration of the layout could be performed. If the frame rate is still above this threshold at some later point the layout will perform yet more iterations. This process continues until the frame rate is below the upper threshold. At this point the layout and the visualisation have been 'balanced' such that the visualisation is rendering as often as necessary and the layout algorithm is iterating as often as possible, the result of which being the graph layout will be performed quicker.

The same principle can also be applied in reverse. If the visualisation has passed a lower bound of frame rate whereby the interactivity has dropped to unacceptable levels, an iteration of the graph layout can be skipped thereby increasing the overall framerate but slowing down the progress of the graph layout.

4.4.7 Configuration

4.4.7.1 Remappable variables

As it currently stands the mapping of profile variables to visualisation channels is static and can only be altered at compile time. By having a configuration means whereby any variable can be mapped to any compatible visualisation channel the flexibility of the profiling tool is increased. For example it might be that while observing the profile of some program you do not care about the number of calls made to a function and instead wish to apply local time proportion to the superior colour visualisation channel rather than node scale.

4.4.7.2 Remappable keys

The default keys used to control navigation within the visualisation are sensible, but not necessarily to every persons taste. The ability to remap each control to a different key potentially increases the usability of the program for many people. This extension should be implemented in combination with section 4.4.4.2.

4.4.7.3 User defined colour maps

The default colour map ranges from blue $(0.0, 0.0, 1.0)$ to red $(1.0, 0.0, 0.0)$. Using this colour map interim values between the minimum and maximum scaling are relatively indistinct. By using a colour map which interpolates between a larger number of colours the resolution of the colour map is effectively increased. At a glance however, this colour map will be harder to interpret than a simpler equivalent. If the colour maps are user definable the benefits of both worlds are reaped since the user is able to choose a colour map to use based on the nature of the profile being generated.

5. Conclusion

The underlying goal of the project was to create a profiling tool which is capable of measuring a program while it is still running. In addition to this the tool was to present the generated profile graphically in a form easy to interpret by humans.

The first of these goals has been achieved. In this case the real time profiling tool is genuinely unique — there is no comparable tool available for the Linux platform that is able to profile a running process in this way. In comparison to existing profiling tools, the profile generated by the real time profiling tool offers little extra other than the ability to watch it as it is beginning constructed. However the ground work is there to produce potentially more useful instantaneous profiling information as described in section 4.4.2.1.

The call graph and profile renderer has also been a success. The resultant visualisation is both efficient and accurate. In terms of data representation the graph topology is incontrovertibly correct. The representation of the actual profile data proper is harder to evaluate, but the usage study performed suggests positive results.

Development of the real time profiling tool has reached a stage where it is already useful. However, it is by no means at an end. There are multiple avenues of interest still to be explored and many possibilities for extension to the project. It is conceivable that specific elements of the visualisation are potentially useful in fields other than profiling, perhaps visualisation of network usage (c.f. *Etherape*[33]).

In addition to the completion of the real time profiling tool, this project has satisfied a peripheral goal. Through its development, the author has learnt and become proficient in the use of OpenGL and other related libraries. Furthermore the development of the real time profiling tool has been a pleasant and satisfying experience.

Appendix A. Instrumentation functions

```
/*
===============
__cyg_profile_func_enter

Instrumentation function called on entry
===============
*/
void __cyg_profile_func_enter( void *this_fn, void *call_site )
{
  if( connection < 0 && !attemptedConnection )
  {
    attemptedConnection = true;

    if( ( connection = connectToRtprof( ) ) < 0 )
      fprintf( stderr, "WARNING: librtprof cannot connect to rtprof\n" );
    else
      atexit( disconnectFromRtprof );
  }

  if( connection >= 0 )
  {
    gettimeofday( &tv, NULL );

    fe.type = EV_ENTER;
    fe.this_fn = this_fn;
    fe.ts = tv.tv_sec * 1000000 + tv.tv_usec;

    if( sendFE( connection, fe ) < 0 )
    {
      disconnectFromFailedRtprof( );
      fprintf( stderr, "WARNING: could not send to rtprof; disconnected\n" );
    }
  }
}
```

```
/*
===============
__cyg_profile_func_exit

Instrumentation function called on exit
===============
*/
void __cyg_profile_func_exit( void *this_fn, void *call_site )
{
  if( connection >= 0 )
  {
    gettimeofday( &tv, NULL );

    fe.type = EV_EXIT;
    fe.this_fn = this_fn;
    fe.ts = tv.tv_sec * 1000000 + tv.tv_usec;

    if( sendFE( connection, fe ) < 0 )
    {
      disconnectFromFailedRtprof( );
      fprintf( stderr, "WARNING: could not send to rtprof; disconnected\n" );
    }
  }
}
```

Appendix B. Event handler

```
/*
===============
serviceConnection

Function to service a librtprof connection
maxEvents is the maximum number of events to read this call
===============
*/
boolean serviceConnection( int connection, callStack_t *s,
                           graph_t *g, int maxEvents )
{
  static functionEvent_t fe;
  static int             size = 0;

  boolean         clientConnected = true;
  graphNode_t     *parent, *child;
  graphEdge_t     *edge;
  void            *parentSymbol;
  stackFrame_t    sf, *sfp;
  timeStamp_t     ts, delta;
  int             count = 0;
  int             eventCount = 0;

  graphNode_t     **nodes;
  graphEdge_t     **edges;
  int             numNodes, numEdges;
  int             i;

  //while there are events to be processed read from the socket
  while( ( size = readFromConnection( connection, size, &fe ) ) == SFE &&
         eventCount++ < maxEvents )
  {
    child = parent = NULL;
    parentSymbol = NULL;

    assert( size == SFE );
    size = 0;

    //deal with the event
    switch( fe.type )
    {
      case EV_ENTER:

        if( !emptyStack( s ) )
        {
          sfp = peekStack( s );

          if( ( parent = searchNodes( sfp->symbol, NULL, g ) ) != NULL )
          {
            delta = ( fe.ts - sfp->calleeExitTime );

            parent->totalTime += delta;
            if( parent->totalTime > g->maxTotalTime )
              g->maxTotalTime = parent->totalTime;

            g->totalTotalTime += delta;

            parent->localTime += delta;
            if( parent->localTime > g->maxLocalTime )
```

```
        g->maxLocalTime = parent->localTime;

      g->totalLocalTime += delta;
    }

    sfp->calleeEntryTime = fe.ts;

    parentSymbol = sfp->symbol;
  }

  sf.symbol = fe.this_fn;
  sf.calleeExitTime = fe.ts;
  pushStack( sf, s );

  if( ( child = searchNodes( sf.symbol, parentSymbol, g ) ) != NULL )
  {
    g->totalCalls++;

    child->calls++;
    child->active = true;
    child->lastActive = getusecs( );

    if( child->calls > g->maxNodeCalls )
      g->maxNodeCalls = child->calls;

    if( parent != NULL )
    {
      edge = searchEdges( parent, child, g );
      edge->calls++;
      edge->active = true;
      edge->lastActive = child->lastActive;

      if( edge->calls > g->maxEdgeCalls )
        g->maxEdgeCalls = edge->calls;
    }
  }
  break;

case EV_EXIT:

  if( !emptyStack( s ) )
  {
    sf = popStack( s );

    if( ( child = searchNodes( sf.symbol, NULL, g ) ) != NULL )
    {
      delta = ( fe.ts - sf.calleeExitTime );

      child->totalTime += delta;
      if( child->totalTime > g->maxTotalTime )
        g->maxTotalTime = child->totalTime;

      g->totalTotalTime += delta;

      child->localTime += delta;
      if( child->localTime > g->maxLocalTime )
        g->maxLocalTime = child->localTime;

      g->totalLocalTime += delta;

      child->active = false;
      child->lastActive = getusecs( );
    }

    if( !emptyStack( s ) )
```

```
        {
          sfp = peekStack( s );

          if( ( parent = searchNodes( sfp->symbol, NULL, g ) ) != NULL )
          {
            delta = ( fe.ts - sfp->calleeEntryTime );

            parent->totalTime += delta;
            if( parent->totalTime > g->maxTotalTime )
              g->maxTotalTime = parent->totalTime;

            g->totalTotalTime += delta;

            edge = searchEdges( parent, child, g );
            edge->active = false;
            edge->lastActive = getusecs( );
          }

          sfp->calleeExitTime = fe.ts;
        }
      }

      break;

    case EV_PROCEXIT:
      close( connection );
      close( serverSocket );
      clientConnected = false;
      break;

    default:
      break;
  }
}

ts = getusecs( );

nodes = listNodes( SF_NONE, &numNodes, g );
edges = listEdges( &numEdges, g );

g->maxInactiveTime = 0;

for( i = 0; i < numNodes; i++ )
{
  nodes[ i ]->inactiveTime = ts - nodes[ i ]->lastActive;

  if( nodes[ i ]->inactiveTime > g->maxInactiveTime )
    g->maxInactiveTime = nodes[ i ]->inactiveTime;

  nodes[ i ]->localTimeFraction = (float)nodes[ i ]->localTime /
                                  (float)g->totalLocalTime;

  nodes[ i ]->totalTimeFraction = (float)nodes[ i ]->totalTime /
                                  (float)g->totalTotalTime;

  nodes[ i ]->callsFraction = (float)nodes[ i ]->calls /
                              (float)g->totalCalls;
}

for( i = 0; i < numEdges; i++ )
{
  edges[ i ]->inactiveTime = ts - edges[ i ]->lastActive;

  if( edges[ i ]->inactiveTime > g->maxInactiveTime )
    g->maxInactiveTime = edges[ i ]->inactiveTime;
```

```
    edges[ i ]->callsFraction = (float)edges[ i ]->calls /
                                (float)g->totalCalls;
  }

  free( edges );
  free( nodes );

  return clientConnected;
}
```

Appendix C. GraphViz DOT language

```
        graph   :   [ strict ] (graph | digraph) [ ID ] '{' stmt_list '}'
    stmt_list   :   [ stmt [ ';' ] [ stmt_list ] ]
          stmt  :   node_stmt
                |   edge_stmt
                |   attr_stmt
                |   ID '=' ID
                |   subgraph
    attr_stmt   :   (graph | node | edge) attr_list
    attr_list   :   '[' [ a_list ] ']' [ attr_list ]
        a_list  :   ID [ '=' ID ] [ ',' ] [ a_list ]
    edge_stmt   :   (node_id | subgraph) edgeRHS [ attr_list ]
      edgeRHS   :   edgeop (node_id | subgraph) [ edgeRHS ]
    node_stmt   :   node_id [ attr_list ]
      node_id   :   ID [ port ]
          port  :   port_location [ port_angle ]
                |   port_angle [ port_location ]
port_location   :   ':' ID
                |   ':' '(' ID ',' ID ')'
    port_angle  :   '@' ID
      subgraph  :   [ subgraph [ ID ] ] '{' stmt_list '}'
                |   subgraph ID
```

Appendix D. Colour map generation function

```
/*
===============
buildLerpCTable

Build a colour table by interpolating between a
number of discrete colours
===============
*/
void buildLerpCTable( vec4_t *colours, int numColours )
{
  int     i;
  int     numRanges = numColours - 1;
  int     range;
  float   rangeSize = (float)MAX_CTABLE_ENTRIES / (float)numRanges;
  float   scale;
  vec4_t  from, to;

  if( numColours < 2 )
    return;

  for( i = 0; i < MAX_CTABLE_ENTRIES; i++ )
  {
    range = (int)floor( (float)i / rangeSize );
    Vector4Copy( colours[ range ], from );
    Vector4Copy( colours[ range + 1 ], to );

    scale = (float)( i % (int)rangeSize ) / rangeSize;

    if( scale < 0.0f )
      scale = 0.0f;
    else if( scale > 1.0f )
      scale = 1.0f;

    colourTable[ i ][ 0 ] = ( ( 1.0f - scale ) * from[ 0 ] + scale * to[ 0 ] );
    colourTable[ i ][ 1 ] = ( ( 1.0f - scale ) * from[ 1 ] + scale * to[ 1 ] );
    colourTable[ i ][ 2 ] = ( ( 1.0f - scale ) * from[ 2 ] + scale * to[ 2 ] );
    colourTable[ i ][ 3 ] = ( ( 1.0f - scale ) * from[ 3 ] + scale * to[ 3 ] );
  }
}
```

Appendix E. Usage study raw data

	1	2	3	4	5	6	7	8	9	10	11	12a	12b	Useful	Cool	Comments
5	7	7	6	2	3	1	7	5	3	a	a	quit	1	3	5	viewport rotation would be nice, as you could maximise the usagage of the viewport. On my PC here, the visualisation was kinda vertical, in a cinema style horizontal boz window.
7	7	7	6	2	3	3	7	5	5	8	1	quit	1	4	5	That's quite a long questionnaire man...
8	7	7	6	2	3	a	6	5	3	a	8	quit	1	4	5	
9	7	7	6	2	3	a	7			8	8	quit	1	5	5	Need a (possibly) pop up widow with direction controls for thoes with bad memories! As an extenstion to program could have key showing actual values of that colours/sizes transparencies represent.
11	7	7	6	3	3	7	7	5	3	7	8	quit	1	5	5	Little difficult to navigate. Shift/Control don't seem to be relative to your current direction. A nice feature would be to center on a function by name.
12	7	7	6	2	3	7	7	5	3	8	7	quit	1	5	5	
13	7	7	6	3	3	a	7	5	3	7	8	quit	1	3	5	
14	7	7	6	3	3	7	a	u	3	7	8	quit	1	4	5	um...well i cant work the controls. I loose the image off the edge of the window and cant 'find' it again and Question 7 there is no choice for I am too uesless to remember what was there before so dont know what is new sorry :)
15	7	7	6	2	3	7	7	5	3	7	8	quit	1	4	5	I felt the rotation was a little tricky to get it to do what you want. But mostly quite a beast.
16	7	7	6	3	3	7	7	5	3	8	8	quit	1	4	5	
17	7	7	6	2	3	7	4	5	3	7	8	quit	1			
18	7	7	6	2	3	7	4	5	3	7	8	quit	1	5	5	Super cool.
20	7	7	6	3	3	7	7	5	3	7	8	Quit	1	5	5	Good Luck!
														3.92	5.0	

The following results were filtered for obvious reasons:

--6	a	a	4	4	4	a	8	8	4	8	a			a	3	4
--10	u	u	u	u	u	u	u	u	u	u	u			u	1	1
--19	8	7	6	2	2	7	5	5	3	7	8	seven	1		4	5

Bibliography

[1] Jay Fenlason and Richard Stallman. gprof manual, 1998. http://www.gnu.org/manual/gprof-2.9.1/gprof.html.

[2] Y.Perret. Functioncheck profiler, 2001. http://www710.univ-lyon1.fr/~yperret/fnccheck/profiler.html.

[3] Pace Willisson. Functioncheck profiler, 2001. http://sourceforge.net/projects/hrprof.

[4] Vadim Engelson. Call graph drawing interface, 1996. http://www.ida.liu.se/~vaden/cgdi/.

[5] Georg Sander. Visualization of compiler graphs, 1995. http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html.

[6] Petter Reinholdtsen. gprof-callgraph.pl, 2000. http://www.student.uit.no/~pere/linux/gprof-callgraph/gprof-callgraph.p%l.

[7] Michael Frhlich and Mattias Werner. davinci graph drawing interface, 2001. http://www.informatik.uni-bremen.de/daVinci/.

[8] Florent Pillet and Colin Desmond. Kprof - kde gprof frontend, 2001. http://kprof.sourceforge.net/.

[9] Gregg Vesonder. Graphviz graph layout system, 2003. http://www.research.att.com/sw/tools/graphviz/.

[10] Josef Weidendorfer. Kcachegrind - a frontend to valgrind, 2003. http://kcachegrind.sourceforge.net/.

[11] Julian Seward and Nick Nethercote. Valgrind, 2003. http://developer.kde.org/~sewardj/.

[12] -finstrument-functions documentation, 2003. http://gcc.gnu.org/onlinedocs/gcc-3.2.2/gcc/Code-Gen-Options.html#Code%\%20Gen\%20Options.

[13] C99 standard, 1999. http://std.dkuug.dk/JTC1/SC22/WG14/.

[14] Kitware. Vtk visualisation framework, 2003. http://public.kitware.com/VTK/.

[15] Opengl graphics library, 1992. http://www.opengl.org.

[16] Brian Paul. Mesa open source gl implementation, 1999. http://www.mesa3d.org.

[17] Microsoft. Direct3d graphics library, 2003. http://www.microsoft.com/windows/directx/default.aspx.

[18] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[19] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989.

[20] T. Fruchterman and M. Reingold. Graph drawing by force-directed placement. *Software — Practice and Experience*, 21(11):1129–1164, 1991.

[21] R. Davidson and D. Harel. Drawing graphs nicely using simulated annealing. Technical Report CS89-13, Department of Applied Mathematics and Computer Science, Weizmann Institute of Science, 1989.

[22] I. Bruss and A. Frick. Fast interactive 3-d graph visualization. In *Proceedings of Graph Drawing '95*. Springer, 1995.

[23] Diethelm Ironi Ostry. Some three-dimensional graph drawing algorithms. Master's thesis, University of Newcastle, Australia, 1996.

[24] Linus Torvalds. Linux kernel, 2003. `http://www.kernel.org`.

[25] Gnu binutils, 2003. `http://www.gnu.org/software/binutils/`.

[26] Roland McGrath. Gnu lib c, 2003. `http://www.gnu.org/software/libc/libc.html`.

[27] Sam Lantinga. Simple directmedia layer, 2003. `http://www.libsdl.org`.

[28] Linas Vepstas. Gle - gl extrusion library, 2003. `http://linas.org/gle/`.

[29] Peter M. McIlroy, Keith Bostic, and M. Douglas McIlroy. Engineering radix sort. *Computing Systems*, 6(1):5–27, 1993.

[30] opengl people. Glut manual, 2003. `http://www.opengl.org/developers/documentation/glut/index.html`.

[31] Andi Gutmans and Zeev Suraski. Php scripting language, 2003. `http://www.php.net`.

[32] Mysql database system, 2003. `http://www.mysql.com`.

[33] Juan Toledo. Etherape network monitoring tool, 2003. `http://etherape.sourceforge.net`.