Instrumenting the Linux Kernel

David Pearce

Final year project report, June 2000

Abstract

An instrumentation tool has been developed for the Linux kernel that uses the relatively new technique of runtime code splicing. This permits instrumentation of a running system, without any source code modification, by redirecting the flow of control to pass through carefully written code patches.

The result is a tool that can insert arbitrary code before and after almost every basic block of an executing Linux kernel. A new twist on the original technique has been developed, called local bounce allocation, that overcomes some of the problems encountered on variable length architectures. Furthermore, the tool has been demonstrated to produce results that are at least as accurate as an existing instrumentation tool whilst providing a major improvement upon it.

Foreword…

In the development of a high performance engine, the mechanical engineer faces the problem of guaranteeing that the components will not fail under the high stresses involved. He must ensure, for example, that a component will not crack at high temperature. To do this, he must be able to predict how each atom of the component will react at temperature, and what effect this will have on its neighbours. He must also predict how the components will react with each other and how air molecules passing through the system affect pressure and dynamics. In short, a monumental number of variables are involved and must be tamed for the engine to work correctly. In spite of all this, the mechanical engineer can produce an engine that will operate in extremely harsh conditions in a predictable manner. How does he do this? How can the behaviour of the system be known when such a large number of variables are present?

The answer, of course, is that he doesn't know how the system will behave until it has been built. Instead, the engineer uses design procedures to help remove some of the uncertainty. But, for such a complicated beast as the high performance engine, this will not be enough. The final trick in the engineers hand is to observe the engine in motion and determine if it is indeed behaving as expected. He will then tweak the engine in an effort to obtain the expected behaviour and once this is achieved, he will feel confident that he understands what takes place inside the engine whilst it is running.

I was inspired by the ability of modern engineering to achieve such feats as described above because I felt that the typical programmer rarely gains the same level of understanding for the software he develops. This was illustrated to me by a few emails sent between developers of the Linux kernel. They were discussing how to make a sensitive measurement regarding the optimal design of the scheduler. I was surprised to find that the effects of extreme optimisations, such as fitting structures into cache lines, were being discussed using, almost exclusively, the speculation of a few individuals. Although these people were experienced programmers it seemed that the number of variables was too large to be accounted for. Some measurement was required to back up the speculation and all that was on offer were the results of a few primitive tests. The aim of my project was to address this by finding a way to accurately observe the behaviour of the Linux kernel.

Many thanks go to my supervisor, Paul Kelly, who has on many occasions pointed me in the right direction. Also, thanks should go to Ariel Burton who has also provided some invaluable help.

Table of Contents

1 **Introduction** 9
1.1 Achievements 9
1.2 What was involved 9
1.3 Overview of the remainder of this document 9
2 **Background to Program Instrumentation** 10
2.1 Instrumentation Techniques 10
2.1.1 Source Code Augmentation 10
2.1.2 Compiler Level Augmentation 10
2.1.3 Binary Rewriting Methods 11
2.1.4 Runtime Code Splicing 12
2.1.5 Non-intrusive instrumentation 14
2.2 Types of Instrumentation and their representation 15
2.2.1 Timers and Counters 15
2.2.2 Path Profiling and Critical Paths 15
2.2.3 Usage Patterns 16
2.3 Additional Background 16
2.3.1 Methodology in the Linux Kernel 16
3 **Overview of the GILK instrumentation tool** 17
3.1 Design Criteria 17
3.1.1 Objective 17
3.1.2 Requirements 17
3.1.3 Design Choice 17
3.2 Terminology 18
3.3 The structure of GILK 18
3.4 Valid Splice Points 19
3.4.1 Local Bounce Allocation 19
3.4.2 A More Aggressive Bounce allocation Strategy 20
3.5 The Relocated Sequence 20
3.6 The Code Patch 20
3.6.1 State Saving 21
3.6.2 The Return Point 21
3.7 Splice Dependence 23
3.8 Dynamic Code Generation 23
3.9 The Instrument Function 23
3.10 The ILK Device Driver 24
3.11 Sample Collection 24
4 **Analysis of GILK** 25
4.1 Correlating with IKD 25
4.1.1 A brief introduction to IKD 25
4.1.2 Design of the experiment 25
4.1.3 The Event Sequence 25
4.1.4 Examining the Results 26
4.2 Measuring the overhead of Instrumentation with GILK 30
4.2.1 Design of the Experiment 30

4.2.2 Examining the Results 30
5 **Reflection and Future Work** 32
5.1 The strong points 32
5.1.1 The specialised disassembler 32
5.1.2 Dynamic code generation 32
5.2 Improvements to GILK 32
5.2.1 Sample generation and collection 32
5.2.2 Instrument co-ordination 33
5.2.3 Presentation of Results 33
5.2.4 The Disassembler 33
5.3 Future Work 33
6 **Conclusion** 35
References 36
7 Appendix A 37
7.1 Ktrace 37
7.2 Adapting ktrace for accurate measurement 37

1 Introduction

The aim of this project was to find a way of measuring aspects of the Linux kernel that might uncover otherwise hidden information regarding its behaviour. To this end, an instrumentation tool called GILK, has been developed for the Linux kernel that is capable of extremely sensitive measurements. The tool makes use of a relatively new technique applicable to instrumentation, called runtime code splicing. This allows GILK to insert measuring code before and after almost any basic block in the kernel.

1.1 Achievements

The main achievements of this project are:

- A new twist on the existing method of runtime code splicing, termed **local bounce allocation**, has been developed.

- An implementation of the runtime code splicing technique has been developed for a variable length instruction set (the Intel x86 architecture) and a detailed account is presented in this document.

- An advanced instrumentation system for the Linux kernel has been produced that is capable of extremely sensitive measurement.

1.2 What was involved

The project involved the development of the instrumentation tool plus an analysis of it. The development took the majority of the effort and required extensive knowledge of the Linux kernel as well as the Intel x86 architecture.

The analysis involved using the tool in an effort to establish the validity of its results. Success was achieved in showing that, for the same sequence of events, GILK and another tool, called IKD, took identical measurements.
The second half of the analysis was concerned with measuring the impact on system performance caused by instrumentation with GILK. A figure has been obtained that provides an indication of the overhead imposed by using GILK.

1.3 Overview of the remainder of this document

The remainder of this document will begin with an introduction to program instrumentation. This will cover the main techniques currently used in the field and will also provide an indication of other work that relates to this project. Some relevant material on the Linux kernel will also be provided, as this is necessary for an understanding of the methods employed in GILK.

After this, an overview of the implementation will be provided. This will focus primarily on the runtime code splicing technique used in GILK as a detailed account of such an implementation has not been encountered before.

The analysis performed on GILK will be presented in section four. This will attempt to outline the experiments performed and provide an explanation of their meaning. Finally, a reflection on the project and a list of future work will be presented.

2 Background to Program Instrumentation

This section will take the reader on a journey through the field of program instrumentation. The primary objective is to cover the existing techniques for program instrumentation and their associated problems. The secondary objective is to introduce some of the concepts that will be used throughout the remainder of this document

2.1 Instrumentation Techniques

Program Instrumentation can be performed by adding instrumentation code to a target program. This can be performed at four distinct stages of a program's lifecycle. These are the source code, compilation, binary and executing stages. There also exists another method, termed non-intrusive instrumentation, which allows instrumentation to take place without modification of the target program.

2.1.1 Source Code Augmentation

Programmers typically add instrumentation code by hand to their software whilst it is being developed. This instrumentation code, often called debugging code, is used to provide the programmer with a better understanding of how his program is working. It is very common to see simple print statements (such as "printf" in 'C') being used to output the value of variables at specific program points, provide execution tracing and to highlight otherwise unnoticeable errors. This primitive instrumentation method is quick and simple. But, it has many drawbacks including a dependency on the program in question and the difficulty of removing such code.

A slightly more advanced way of achieving the above is to provide a conditional compilation option. For example, in the 'C' language it is usual to find a macro called ENABLE_DEBUGGING (or something similar) which when defined causes simple debugging code to be included during compilation of the program. This is advantageous to a more manual use of debugging code, because a production version of the program can be generated simply by undefining the macro.

There are some more subtle problems, which relate to program instrumentation at the source code level. The most important of these being the change in behaviour of the program. This happens because parameters of the program such as size, relative location of functions, heap fragmentation and stack usage typically differ between the instrumented and non-instrumented versions of the program.

Some examples of serious program instrumentation implemented at the source code level include IKD [15] and KernProf [14]. Both of these provide some form of instrumentation for the Linux Kernel. They require a patch to be applied to the kernel source tree, which necessitates a recompilation of the kernel before instrumentation can take place. This introduces an interesting problem. The patches are dependent on specific versions of the Linux kernel and so must be updated with each kernel release. This generates a lot of work for the patch maintainer, as new versions of the Linux kernel are released almost every month. The net result is that the patches usually lag behind the latest kernel releases.

2.1.2 Compiler Level Augmentation

A solution to the dependency problem described above is to insert instrumentation using the compiler. In this method, the source code is not touched and instrumentation is added by compiling the program. An example of this is the '–pg' switch of GCC. This switch causes GCC to automatically insert a call to the function "mcount" at the start of each function. The "mcount" function simply increments the current call count and writes it to an output file. This is illustrated in figure 2.1.

The call count can then be viewed using gprof[13]. The advantage to this approach is that instrumentation is decoupled from program source. However, GCC (for example) can only provide a very limited amount of instrumentation, including function profiling, basic block profiling and critical path profiling. If more complex instrumentation is required, a patch to GCC must be written.

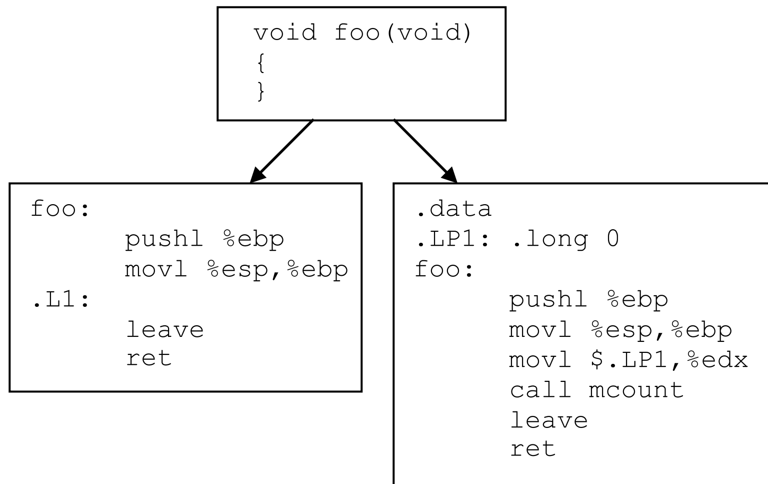

Figure 2.1 - Showing the function foo (top) being compiled without the -pg switch (left) and with the -pg switch (right)

The main drawback of this approach is the requirement to recompile the target. For a large piece of software, this can consume a significant amount of time. Also, it is often difficult to restrict the instrumentation to just the relevant program points. This results in the burden imposed by instrumentation being greater than really necessary.

However, there is an important advantage to this approach. During compilation, a great deal of information is generated by the compiler, about the structure of the resultant executable. This includes information such as control flow graphs, live register analysis, symbol information and knowledge of which parts represent machine instructions. The last part may seem mysterious, as modern executables are divided into code and data segments. However, compilers commonly put read-only data in the code segment for reasons of performance and it is often difficult to distinguish data from instructions in these cases.
This information, critical for instrumentation at the machine code level, is discarded once compilation is completed. Instrumentation tools, which work at the machine code level, go to great lengths to recover this information. Therefore, a tool which works at the compilation level is in the unique position of having access to this information for free. This means that instrumentation can often be added with a minimal impact on program performance. These topics are discussed in [1] and [2].

2.1.3 Binary Rewriting Methods

A number of instrumentation tools have been developed which add instrumentation code to the target's executable. Examples of this solution include QP,QPT [1], EEL [6] and BIT [7]. These tools remove the need to recompile the target in order to perform instrumentation. This addresses two issues. Firstly, the turnaround time for instrumentation is significantly reduced. Secondly, it becomes possible to instrument software when the source code is unavailable. This is a major improvement over previous methods for the following reasons:

1. A software program is normally linked with executable libraries. The source code for these is not always available. However, to better diagnose the performance of a program it is useful to understand the characteristics of any libraries it uses.

2. The effects of compiler optimisations are visible in the executable file. This means that they can be measured to determine the impact on the target program.

The advantages to this approach are offset by the additional complexity of implementing it. These topics will be briefly covered here. A more detailed examination can be found in an excellent paper by James Larus and Thomas Ball [1].

The technique used by the QP/QPT binary rewriting tools begins by generating a control flow graph of the program. Then, instrumentation is added to the incoming edges of basic blocks. This is achieved by shifting program code along to make space for the instrumentation code. This results in the relative positioning of the original instructions being changed. Therefore, all branch and call instructions must be adjusted to accommodate for this. These ideas are illustrated in figure 2.2.

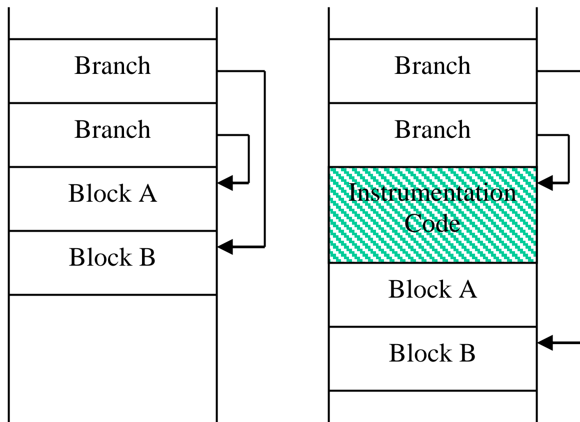

Figure 2.2 -Showing part of an executable file before (left) and after (right) instrumentation has been added to basic block A. The branch leading to block B has been adjusted to allow for the change in location of its target.

Most architectures provide the ability to perform indirect branches and these hinder the generation of control flow graphs. The reason for this is obvious: How can the destination of an indirect branch be known beforehand, when its value depends upon a runtime variable (register or memory)?

Data flow analysis may uncover the destination of a few indirect branches, but not all. However, the work done by Larus and Ball indicates that compiled code makes use of indirect jumps in a controlled fashion, which can be analyzed.

In fact, they argue that indirect branches are used almost exclusively for the switch statements found in most high level languages. Moreover, compiled switch statements use (static) jump tables. If these tables can be located, their values can be adjusted to reflect the new structure of the executable.

Therefore, what remains is the (rare) case of an indirect jump being encountered which is not part of a switch statement or its jump table cannot be located. To solve this, a slightly unusual method can be employed. Small pieces of code are inserted before problematic indirect branches. This code performs a runtime analysis of the indirect destination and adjusts it where necessary. In order for this to work in the general case, a complete mapping from the original addresses to the new addresses is required and this imposes a serious overhead on the instrumentation.

The instrumentation code inserted normally saves and restores the entire state of the machine. This effectively hides it from the program being executed. However, this has a significant overhead. To overcome this, QP/QPT performs a live register analysis whilst constructing the control flow graphs. In some cases, this allows for a reduction in the cost of instrumentation. It should now be clear why the detailed information discussed before, which is discarded by the compiler, is so valuable to an instrumentation tool.

2.1.4 Runtime Code Splicing

This is really a novel way of applying the techniques used in binary rewriters. The idea is still to insert instrumentation code into the program at the machine code level. However, instead of statically rewriting the executable, the target program is dynamically modified whilst executing on the system. The problems of indirect branches, control flow graph generation and live register analysis, which are faced by the binary rewriter, must also be faced by the runtime code splicer. However, there is the additional complexity of adding instrumentation whilst preserving the program's integrity at all times. An excellent discussion of this method can be found in a paper by Barton Miller and Ariel Tamches [2]. This paper focuses on their tool, called KernInst. However, since writing that paper a more advanced performance tool, called Paradyn, has been developed and documented by Miller, et al [8] and [9]. Another system which uses this idea is called Aprobe [10].

This approach has several advantages over binary rewriting. Firstly, instrumentation can be performed on the program's original executable. This is significant because it paves the way for instrumentation of running systems that cannot be restarted. Secondly, the instrumentation can be changed during execution of the program. This means that the overhead can be reduced by staggering instrumentation throughout execution. In a binary rewriting system, the instrumentation is active for the duration of execution.

2.1.4.1 Theory of runtime splicing

Runtime code splicing is achieved by using branch instructions (called splices) to divert the normal flow of control, so that it passes through the desired instrumentation code. These branches overwrite existing instructions in the program. To preserve behaviour, the overwritten instruction(s) are “relocated” to the instrumentation patch (sometimes called a base trampoline). The act of relocation must adjust any PC relative instructions. The instrumentation patch must also insure that control is always returned to the correct place. This is illustrated in figure 2.3.

The above is really just a binary rewriting technique. The focal point of doing the it at runtime is the moment when the splice is written to memory. This is critical because another process (or thread) may be about to execute or is executing the instruction(s) overwritten. If the splice is not atomically written, a sleeping process/thread, whose PC points to the instruction being overwritten, may be scheduled CPU time during the write. The result is the process/thread executing a semi-overwritten instruction which, in turn, leads to unpredictable behaviour.

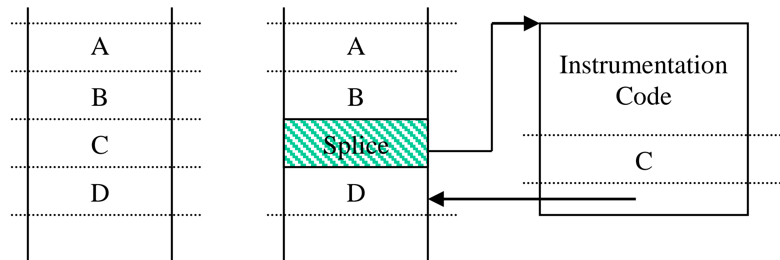

Figure 2.3 - Shows part of a program before (left) and after (right) splicing has been performed. The letters A-D represent arbitrary machine instructions.

Clearly, the above scenario must be prevented from taking place. The KernInst tool [2] overcomes this by ensuring that the splice is inserted using an atomic write. This only solves the problem for the case when the splice instruction overwrites a single instruction. On fixed length instruction architectures, such as RISC, this property always holds, thus splicing is safe.

Variable length instruction sets pose an additional complication, as a splice may overwrite multiple instructions. This is a problem for two reasons:

1. The PC of a process/thread may point to one of the instructions (other than the first) overwritten by the splice. Then, when the process/thread gets CPU time, its PC will no longer be aligned with an instruction boundary.

2. The destination of a branch, located elsewhere in the program, may be one of the instructions (other that the first) which gets overwritten by the splice. Then, when the branch is taken, the PC will become misaligned as before.

These are illustrated in figure 2.4. The second point is quite easily resolved by building a control flow graph of the program. Each node in the control flow graph represents a basic block, which is a consecutive sequence of instructions where control always enters at the beginning and can only branch at the end. A good source of information on control flow graphs and basic blocks is the “Dragon” book [4]. The point of doing this stems from noting that the situation described in point two (above) arises from overwriting a boundary between two basic blocks. This can either be prevented from happening, meaning some instructions cannot be spliced, or the problematic branches can be adjusted in some way.

Finally, there is an issue of splice range. In some circumstances, the splice instruction used may have insufficient range to reach the instrumentation patch. The solutions employed in [2] and [11] use branch bouncing to overcome this. A bounce is simply another branch instruction located elsewhere in the program. The splice instruction then uses it as a stepping stone to the instrumentation patch. The only complication with this is finding a place to locate the bounce. One solution is to identify sections of code that are never executed again and, hence, can be overwritten. This is the bounce location policy used by KernInst [2].

2.1.4.2 Use of the Stack

There exists a simple method for providing pre and post function instrumentation, proposed by Ariel Butron. This works through a slight abuse of the stack. The key idea is to arrange for the return address of the current stack frame

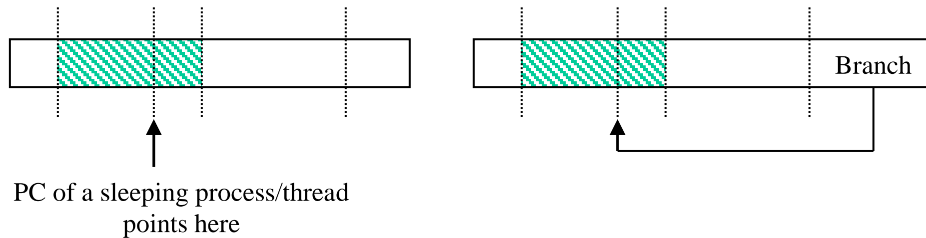

Figure 2.4 - Illustrates the two problems of splicing on a variable length instruction set. On the left, instruction misalignment occurs because a process has been put to sleep on the second instruction overwritten. When it awakes, it will find itself in the middle of a branch instruction! On the right, a branch in the program points to the second instruction overwritten. When that branch is taken control will be transferred to the middle of the splice instruction. In both diagrams, the shaded area represents the instructions overwritten by the splice and the dotted lines represent instruction boundaries.

to point to the desired post-function instrumentation code. Then, when the function terminates (via a return instruction) the post function instrumentation will be activated.

In order to set this up, a splice is written over the first instruction of the function. This points to a code patch that performs the stack manipulation, any desired instrumentation, the execution of overwritten instructions and then returns to the correct location in the original function.

A few possibilities arise from this. Firstly, the overwritten instructions could be written back to their original location in the function. Control would then be returned to the start of the function, which would proceed as normal. This prevents any instruction relocation and also helps alleviate the problem of overwriting more than one instruction with the splice. Then, when the post-function patch gets activated the splice could be rewritten back over the function entry point, allowing instrumentation of the next call to the function. However, recursive calls would be lost. Also, for variable length instruction sets, care must be taken to either ensure control is never within the function when the splice is written or that no branch points to any overwritten instructions. Failure to do this would risk the problems already discussed with splicing for variable length instruction sets.

Secondly, the actual return address of the function can be saved in a couple of ways. It could simply be stored in a variable by the pre-function patch which is then read by the post-function patch. However, recursive calls and multiple threads pose a serious obstacle. Alternatively, the current stack frame could be duplicated with the return address being modified. This provides for recursive calls and multiple threads, but has the disadvantage of requiring knowledge of the prototype of the function being instrumented.

At the time of writing, however, know known implementation of this technique exists.

2.1.5 Non-intrusive instrumentation

All of the instrumentation techniques mentioned so far have attempted to insert code into the program, which makes the necessary measurements. However, statistical sampling methods provide a way of gathering information about the behaviour of a program without modifying it at any level. Good examples of this are Morph [12], DCPI [5], gprof[13] and KernProf[14]. What these programs do is arrange for the target program to be periodically interrupted. The value of the program counter (or other registers) before interruption can then be logged along with a timestamp. This provides information on how the program counter changes over time, which can be used to approximate function duration, find frequently executed program points and much more.

Morph uses the hardware timer interrupt to generate samples. This means that sampling occurs at regular intervals, which can be slightly disadvantageous because events that are coherent with the timer interrupt will always be missed. DCPI, makes use of the Alpha's hardware performance counters to generate the required interrupt. These counters can be configured to cause an interrupt on overflow. The period between samples is changed after each interrupt by initialising the hardware counter with a pseudo-random number, which results in it overrunning after a varying number of events.

The main advantage of this method comes from the large number of samples generated at only a small cost on system performance (estimated at 1-3% for DCPI and less than 0.3% for Morph).

The disadvantage comes from the accuracy of results generated. For example, the time taken for a function could be measured by finding a sequence of consecutive samples whose PC values are within the function. The time between the first and last sample of the sequence would then be the estimated value for duration. Clearly, the accuracy of the result obtained depends upon the frequency of sample generation. This can be improved by averaging such approximations, but much information on how duration changes with time is then lost.

2.2 Types of Instrumentation and their representation

This section will attempt to identify some types of information, useful to performance optimisation, and also how they can efficiently be represented.

2.2.1 Timers and Counters

The simplest type of information that can still be of use is an event count, over time. Interesting events for counting include function entry and exit, cache misses, branch mispredictions and I/O writes/reads. Counters provide a relatively efficient way of representing data. Typically, they are used in a time histogram [16], which is a fixed length array of counters (also called buckets). Each counter maintains the event count for a specific time interval, hence the time histogram represents a finite amount of time. If the time histogram overflows its capacity can easily be increased by changing the time interval each counter represents and merging counter values, which reduces the granularity of results. This arrangement of counters is used by Paradyn and DCPI.

Another useful instrumentation is the execution time taken for a code section. This type of measurement requires starting a timer and then awaiting a trigger to stop the timer. Storage of this type of data is a little harder. The obvious solution is to log every time value recorded. For code segments that have a very high frequency of execution, this results in a large amount of data being produced. It is not desirable to deal with such large amounts of data for a number of reasons. Firstly, it is often difficult to extract useful information from a large data set. Secondly, moving copious amounts of data through the system will hit on performance and possibly affect the validity of the results. This can be overcome by averaging the results, possibly over time intervals in a fashion similar to the time histogram. A better solution, used in Paradyn, is to periodically sample the timer value itself. This reduces the amount of data produced, but does not lose the accuracy of the timings taken. So long as the period used is not regular, a statistical picture of how the execution time varies with time will be generated. This is pointless for programs that are completely irregular (unless data is captured over a set of runs), but as most programs exhibit some regularity in their control flow, it can be quite useful.

2.2.2 Path Profiling and Critical Paths

Another interesting aspect of a program is the execution path taken through it. Identifying frequently executed paths allows the programmer to concentrate his optimisation efforts. Alternatively, finding individual paths that take an unreasonable amount of time to execute can also lead to important optimisations.

An efficient method for computing the execution frequency of basic blocks, which only adds instrumentation to arcs on the minimal spanning tree, is described in a paper by Larus and Ball [17]. The paper also discusses the problem of efficient path representation. A simple solution to this is to instrument each basic block so that it logs its identification number upon execution. This produces a sequence of block identification numbers which represents the path taken through the control graph. However, many logs are made for the identification of one path which is somewhat inefficient. The optimal solution is to assign each unique path with a unique number. This number can be

logged at the terminating nodes of the graph to produce a profile of the paths taken through the control flow graph over time. This is achieved in [17] by use of a variable, initialised at the entry point, which is incremented on traversal of certain arcs in the control flow graph. The net result is that a value is available at the terminating nodes, which uniquely identifies the path taken.

2.2.3 Usage Patterns

The methods described above are normally used to locate generic bottlenecks and inefficiency within a program. However, knowledge of the typical/desired usage of the program can open up new optimisation possibilities. For example, recording function parameter values can provide useful information. It is often the case that certain parameter values occur frequently. Hence, the function can be optimised for these cases. Better still, specialised versions of the function can be produced for specific parameter values. The representation of this type of information is dependant upon the function parameters being logged and as such, has no optimal storage method.

2.3 Additional Background

The following section covers a few pieces of additional information that are relevant to the remainder of this paper.

2.3.1 Methodology in the Linux Kernel

As already mentioned, runtime splicing with a variable length instruction set poses a few problems. For it to be safe, it must be possible to atomically write the splice instruction to memory. Also, the possibility of instruction misalignment must be prevented. This last point can be partially achieved by constructing a control flow graph and all that remains of it is to ensure no process can sleep at an instruction overwritten by the splice.

To this end, the methodology of the Linux Kernel provides some help. The main points are:

- **A process executing in kernel space must run to completion unless it voluntarily relinquishes control**. This means that, when it is in kernel space, it won't be scheduled off the processor even if its timeslice expires.

- **Processes running in kernel space may be interrupted by hardware interrupts**. This may appear to contradict the first rule. But, control is always returned to the process executing in kernel space and it remains possible for processes to prevent interruption over specified segments of code.

- **An interrupt handler cannot be interrupted by a process running in kernel space**. At first, this may seem a little irrelevant. However, this fact ensures that any process (executing in kernel space) that is interrupted will always regain control after the interrupt *and before any other process*.

The three points above, taken together, allow kernel functions to overcome the remaining problems of a variable length instruction set. Firstly, a kernel function can block interrupts whilst the splice instruction is being written to memory. This guarantees that the splice is written atomically[1].

Secondly, a process can only sleep whilst in kernel space if it voluntarily releases control. This is performed by (indirectly) calling the schedule function of the kernel. Therefore, the PC of a sleeping process will always be within the schedule function itself. This fact solves the second half of the instruction misalignment problem. Although it does mean that the critical part of the schedule function cannot safely be instrumented.

Further discussions on these topics can be found in "Linux Kernel Internals" [3].

[1] Actually, it only guarantees it for a single processor compile of the kernel. However, the use of spinlocks guarantees this for SMP versions of the kernel. Also, use of spinlocks on a single CPU version evaluates to blocking interrupts, thus the code is portable.

3 Overview of the GILK instrumentation tool

The GILK tool is the implementation part of the project. It provides a powerful runtime instrumentation tool along with a Graphical User Interface to drive it. It has been designed to run under a GNU/Linux based operating system on an Intel processor. It also requires the GTK widget library. This section will cover the main topics regarding the implementation of GILK. The focus will be on the implementation of runtime code splicing used in GILK, as this is the important part of the tool.

3.1 Design Criteria

3.1.1 Objective

As stated in the introduction, the aim of this project was to find a way to instrument the kernel that might provide unique insight into its behavior. In order to achieve this, an instrumentation tool had to be developed. Hence, a design decision arose regarding the instrumentation technique that would be used by this tool. The objective of this subsection is to identify the main factors that influenced this decision.

3.1.2 Requirements

Having determined that an instrumentation tool was to be developed, the next stage was to identify the minimal requirements of such a tool. These requirements revolve around two main points. Firstly, new kernel versions are released very regularly. Secondly, the purpose of the tool is to provide accurate results. The requirements of the tool were

- **No source modification -** Modifying the kernel source to add instrumentation is not an attractive approach. This is mainly due to the need to update the tool for each kernel release (see section 2.1.1). This problem is very pertinent to Linux because of the relatively small time between kernel releases.

- **Minimal Impact to Overall Performance** - The ideal instrumentation tool would take measurements without affecting the performance of the system. Although this ideal is unachievable (with software instrumentation, atleast), it is still desirable to get as close to it as possible. The reason is simple: If the act of measuring a system dramatically affects its performance/behaviour, any measurements taken will reflect this and, as a result, maybe misleading.

- **Safe Instrumentation** – Obtaining information on real life applications can provide insight on kernel usage. However, Instrumenting real applications poses a problem. For example, data on a file server could be generated through instrumentation of an identically configured machine. It then remains to carefully simulate the input to the original server. This is not a trivial task and it would be simpler to directly instrument the file server. Clearly, no system administrator would allow this without some assurance regarding the safety of the instrumentation process. Therefore, the tool should be able to instrument without risking the integrity of the target system.

Other requirements could have been adopted. These include requirements on security related issues, the types of instruments possible and on the portions of the kernel, such as dynamically loaded components, which are available for instrumentation. However, in an effort to reduce the complexity of the tool, it was decided to concentrate only on the three requirements listed above.

3.1.3 Design Choice

The three methods that satisfy the above requirements are binary rewriting, runtime code splicing and statistical sampling. The approach chosen for the instrumentation tool was runtime code splicing. This technique, discussed in section 2.1.4, fits the requirements of the instrumentation tool very well. It doesn't require any source code modification and the overhead, while not as low as that of statistical sampling methods, is still small. In particular, the ability to instrument without restarting the system makes this a very attractive approach for kernel instrumentation.

Runtime code splicing falls down in one area: the safety of the system being instrumented. This is because, although runtime splicing is theoretically safe, the complexity of the technique (particularly on variable length architectures) makes it difficult to realize this.

The final factor in the decision to use this technique was the limited amount of existing work available, especially with respect to variable length instruction sets, when compared with the alternatives.

3.2 Terminology

Under GILK, an instrument may be applied to a basic block from the control flow graph of a kernel function. There are two types of instrument, the **pre-hook** instrument and the **post-hook** instrument. A pre-hook instrument will be executed before the first instruction of the basic block. In contrast, a post-hook instrument is executed after the last **sequential instruction** of the block. A sequential instruction being neither a branch nor function call return. In other words, for a sequential instruction execution always proceeds to the instruction located physically after it.

An instrument is a triple consisting of a unique identifier, a call sequence and an instrument function. An instrumentation is a set of instruments that are applied in some order. Under GILK, the order that instruments are applied is defined in terms of time. The **launch time** of an instrument refers to the time, in seconds, from the start of the instrumentation that the instrument is activated. Conversely, the **splash time** is the time at which the instrument is deactivated, taken in seconds, from the start of the instrumentation.

A **call sequence** is a sequence of assembly language instructions. It is responsible for calling the instrument function., which involves setting up required parameters on the stack, making the function call and restoring the state of the stack. Different call sequences may be used for different instrument functions, allowing different prototypes for the instrument functions.

The **instrument function** is a normal function, written in C. Its prototype is determined by the designer of the function. However, it must be called by an appropriate call sequence. It is usual for the instrument function to accept at least one parameter, which is the unique instrument identifier.

Each instrument will generate samples. Hence, it must be possible to identify the instrument responsible for each sample. This is achieved through the **unique instrument identifier**. This is more of a fingerprint for the instrument. It consists of the **major number**, the **instrument type**, the **block index** and the **instrument order**. The major number identifies the kernel function the instrument is located in. The instrument type indicates whether the instrument is a pre or post-hook instrument. The block index identifies the basic block within the control flow graph of the kernel function. Finally, the order index indicates the relative execution order of the instrument with respect to other instruments of the same type that are applied to the same basic block. Hence, an order index of zero indicates that the instrument will be executed before all other instruments of identical type and block index. It is not possible for two instruments to have the same identifier.

A **sample** is a tuple consisting of a unique identifier and a value. The meaning of the value component is determined by the instrument function that produced it.

The word **splice** represents a branch instruction used to overwrite existing instructions, in order to insert additional instructions into the program, whereas a **bounce** is a branch instruction that is used by a splice as a stepping stone to a destination out of the splice's range. These ideas have already been introduced in section 2.1.4. Finally, a **bounce slot** refers to an area of kernel memory which may be used for a bounce. A splice may "expose" a number of bounce slots, which are then "allocated" to specific bounces where required.

3.3 The structure of GILK

An overview of the internal structure of GILK is shown in figure 3.1. This diagram serves only to highlight the main components of the instrumentation tool.

A custom disassembler is used because additional instruction information is required for generating the control flow graphs used by GILK. The instrumentation manager is responsible for constructing and installing the splices, bounces and code patches. The kernel analyzer is used to generate control flow graphs from the set of available kernel symbols. If the kernel analyzer cannot completely construct the control flow graph for a symbol, it will mark the symbol as "not instrumentable". This happens, for example, when a function containing a jump table is encountered, as GILK does not attempt a more aggressive analysis in these situations. However, this does not turn out to be much of a limitation, as only a very small percentage of the kernel functions make use of jump tables.

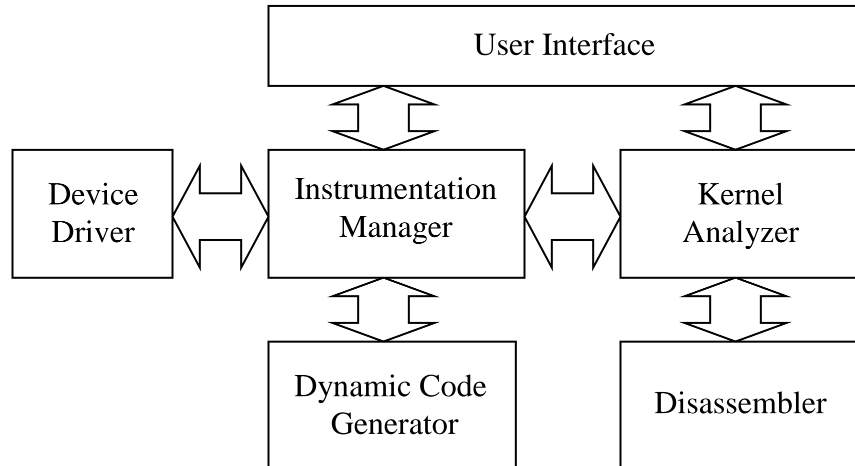

Figure 3.1, showing a structural overview of the GILK instrumentation

3.4 Valid Splice Points

As already mentioned, a control flow graph is generated for each kernel symbol. This is necessary to identify instruction boundaries that are unsafe to overwrite. The problems involved in splicing with a variable length instruction set have already been mentioned (see section 2.1.4) and it should be made clear that a splice cannot be placed across a basic block boundary. This is because a basic block boundary is necessarily the target of a branch or call instruction.

It is, however, safe to place a splice at any point within a basic block, so long as it begins on an instruction boundary. In order to make the tool more practical GILK does not allow such fine grained splicing. Instead, it allows a "pre-hook" and/or "post-hook" splice to be added to a basic block.

A **pre-hook splice** is placed at the beginning of a basic block whereas a **post-hook splice** is placed at the end of a basic block. A post-hook splice must still begin on an instruction boundary, and so may not always be exactly at the end of the block.

3.4.1 Local Bounce Allocation

There is a slight complication with the above. On an Intel x86 processor a 32-bit branch instruction is five bytes long. But, a basic block may be only a single byte in length. Clearly, it would be unsafe to splice such a small basic block using a five byte branch instruction, as this would result in the splice straddling a block boundary.

GILK uses a simple method to expose space for **local bounces**. A local bounce is a bounce that is located within the same kernel function as the splice which uses it.

When asked to instrument a basic block, GILK examines the size of the block. If it is less than thirty-two bytes in size[2], it will relocate the entire block into a code patch. Otherwise, it will just relocate those instructions overwritten by the splice.

[2] 32 was chosen because it meant the every basic block that wasn't relocated would have to contain atleast two instructions. This saved some complication in design of the tool.

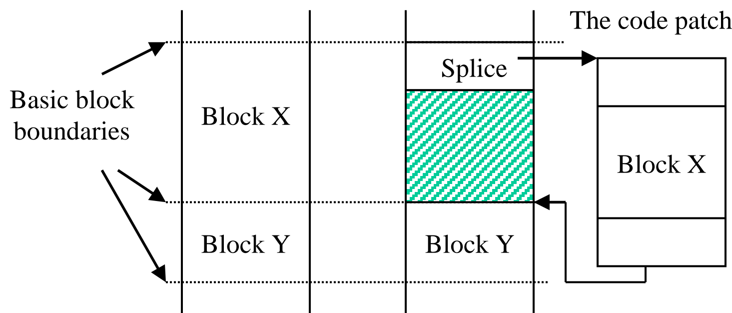

Figure 3.2, showing part of a kernel symbol before (left) and after (right) a splice has been made. The shaded area represents space exposed by relocating the whole of block X. The exact contents of the code patch is not important for this example.

Now, if a basic block that is ten bytes or more in size is completely relocated it will expose at least five bytes within the kernel function that are never executed. Figure 3.2 helps make this clearer. Therefore, the exposed bytes are available for use as bounces.

There remains one point to note about GILK's solution to the awkward block problem. It does not permit the instrumentation of single-byte basic blocks, as there is no single byte branch instruction in the Intel x86 architecture.

3.4.2 A More Aggressive Bounce allocation Strategy

The simple strategy outlined above allows splicing of awkward blocks in many situations. However, GILK was designed with a more aggressive strategy in mind and only time has prevented its full potential from being realized.

It is straightforward to see that, relocating instructions from a basic block exposes space in that block that can be safely overwritten. Therefore, instead of just relocating the instructions overwritten by a splice, extra instructions after the splice could also be relocated in order to expose additional free space for local bounces.

This could be taken a stage further. If a bounce is required for a particular splice and the previous methods cannot be applied, then an entire basic block could be relocated just for the purpose of exposing space for a bounce. This should be used as a last resort, because it introduces two unnecessary branches into the instruction stream.

All of this raises the obvious question: Why not just relocate the entire function and apply existing binary rewrite methods to insert the required instrumentation?

The answer to this question depends upon the intended use of the instrumentation tool. If a large number of instruments are to be made simultaneously, then relocating every function that is to be instrumented would consume a lot of kernel memory. Kernel memory cannot be swapped to disk, and thus the amount of available physical memory for user space processes would be reduced, which may impact upon system performance.

However, if only a few instruments are to be made simultaneously then it makes sense to relocate the entire function, particularly as it would permit instrumentation of all basic blocks (including single byte blocks) without complication.

3.5 The Relocated Sequence

As discussed above, GILK determines which bytes are to be relocated based upon the size of the block and what instrument(s) are being applied to the block. The set of instructions actually relocated for an individual splice is termed the "relocated sequence".

The instructions that make up the relocated sequence are not always identical to those in the original block. GILK must modify any instructions that use relative addressing. In some cases, this requires a new instruction and others simply an adjustment of the relative address, taking into account the new location of the instruction.

3.6 The Code Patch

Each splice connects to a unique code patch (possibly via a bounce). The contents of the code patch are determined by the instrument(s) being applied to the basic block and the relocated sequence.

For example, suppose that some pre-hook instruments are being applied to a basic block and the block is large enough to avoid complete relocation. In this case, the call sequences of the instruments are placed at the start of the code patch. In this way, the call sequences and the instrument functions they connect to will execute before the first instruction of the basic block.

The procedure for a post-hook instrument applied to a large basic block is similar, except that the call sequences are placed after the relocated sequence. There is one complication, as the last instruction of a basic block may be a branch or function call return. This is a problem because the presence of such an instruction may prevent the call sequences (located after it in the code patch) from being executed. In this case, GILK "splits the block end". What this means is that the offending instruction is separated away from the relocated sequence. The call sequences are then sandwiched between the remainder of the relocated sequence and the offending instruction. This results in the call sequences being executed after the last sequential instruction of the basic block. Figure 3.4 shows a good example of a post-hook instrument being applied.

The final case to cover is the application of instruments to a basic block that needs complete relocation. In this situation, the splice instruction is always written over the start of the basic block and the relocated sequence consists of all the instructions that make up the basic block. If a pre-hook instrument is requested for the block then its call sequence is located before the relocated sequence. Also, if a post-hook instrument is requested for the block, then its call sequence is located after the relocated sequence, with the block end being split when necessary. Figure 3.3 shows the three possible code patch configurations.

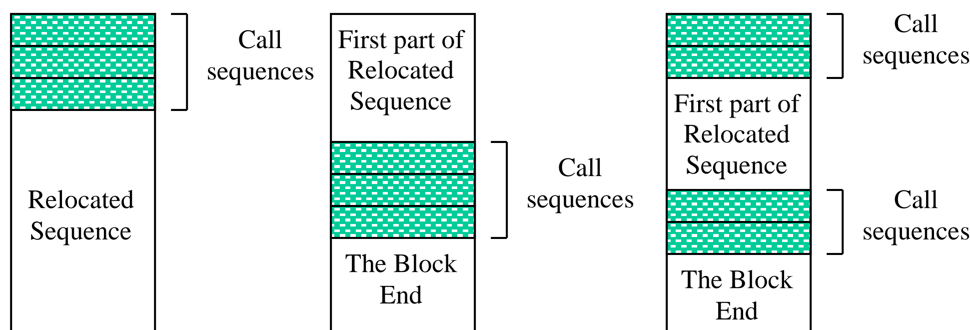

Figure 3.3 - The three possible code patch configurations.
Note that the block end will be empty in some cases

3.6.1 State Saving

In order for the kernel symbol being instrumented to behave exactly as it did before, it is necessary for the state of the machine to be preserved across all call sequences in a code patch. On the Intel architecture, preserving the state of the machine consists of preserving the values of all the registers, including the flags register.

3.6.2 The Return Point

There is one aspect of the code patch that has not yet been mentioned. That is, where control goes after the code patch has been executed. Clearly, it must return to the instruction that originally followed the last instruction of the relocated sequence.

There are four types of basic block that are of concern when considering this problem:

1. The block has no arcs leading from it. This occurs when the final instruction of the block is a return instruction.

2. The block has a single "follow-on" arc. A follow on arc occurs when a sequence of instructions produces two basic blocks in the control flow graph because one of the instructions is the destination of some branch located

elsewhere. The follow-on arc is therefore the arc that connects these two blocks together. It represents the fact

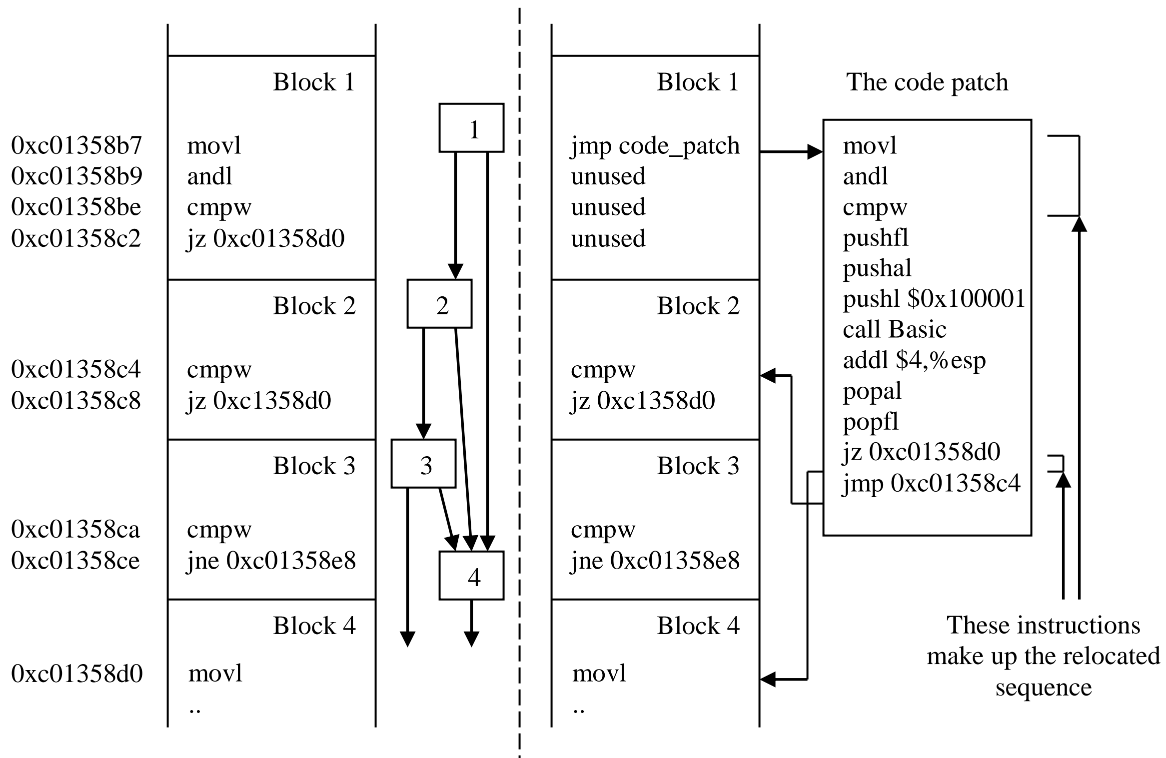

Figure 3.4 - On the left we see part of a kernel function and its control flow graph. On the right we see the result of applying a post-hook instrument to block 1. Some of the instruction operands have been omitted purely for simplification. In the code patch, notice that the relocated sequence has been separated as a result of splitting the block end and that the conditional branch, although relocated to the code patch, has been fixed so that it still points to its original destination. Also, the destination of the terminating branch is the instruction that originally followed on from the conditional branch. As a final note, the unused slots in the new block 1 are available to be allocated for local bounces.

that the two basic blocks are physically located next to each other. An example of a follow-on arc is the arc from block two to block three in the control flow graph of figure 3.4.

3. The block has a single outgoing arc that is not a follow-on arc. This occurs when the final instruction of the block is an unconditional branch.

4. The block has two outgoing arcs. This occurs when the block's final instruction is a conditional branch. In this case, one arc is effectively a follow-on arc and the other points elsewhere in the control flow graph.

There are also a few more points that should be noted. Firstly, there can be at most one branch or return instruction in the relocated sequence. Secondly, if a branch or return instruction is present in the relocated sequence then it is always the last instruction. Hence, splitting the block end guarantees that all the call sequences will be executed before any branch or return instruction in the relocated sequence.

GILK ensures that control returns to the correct location in the original function by making the last instruction of the code patch an unconditional branch, called the **terminating branch**. This points to the instruction that would normally be executed after the last instruction of the relocated sequence.

To see that this works, consider the four cases outlined above. The first case is trivial as a return instruction will always return to its call site. Hence, in this case the terminating branch is unnecessary.

In the second case, there are no branch instructions in the relocated sequence. Therefore, the terminating instruction will eventually be executed and control will return to the correct point.

In the third case, the mechanism for generating the relocated sequence will ensure that the unconditional branch at the end of the relocated sequence still points to its original destination. Again, the terminating branch is unnecessary here.

The final case is a mixture of the above. The conditional branch is adjusted when generating the relocated sequence so that the taken destination is still the original destination. If the branch is not taken, control will fall through to the terminating branch and the follow on arc will effectively be taken. This case is illustrated in figure 3.4.

3.7 Splice Dependence

The use of local bounces introduces a dependence problem. This occurs because a bounce is dependent upon the space it overwrites being unused. As an example, suppose that splice SB is a splice that requires the local bounce B1. Additionally, the splice SE exposes the bounce slot allocated to B1. This is shown in figure 3.5.

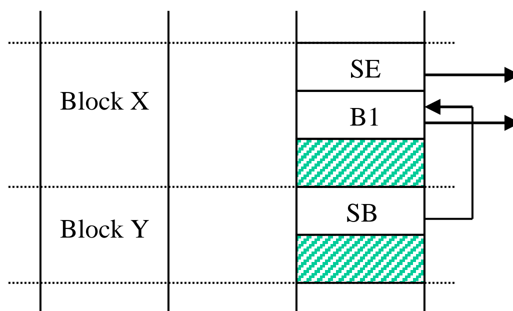

Figure 3.5, showing part of a kernel function before (left) and after (right) an instrumentation has been applied. In this case, splice SB is dependent upon bounce B1, which is dependent upon splice SE.

Now, B1 is dependent upon SE because it cannot be installed (written to kernel memory) unless SE is already been installed. This is because the bounce slot used by B1 is only available when SE is installed. Also, SB is dependent upon B1 as it cannot be installed before B1. The best solution to this problem is to atomically install all splices and bounces together. At the current time, however, GILK does not install them all together. Instead, GILK installs splices and bounces in an order which preserves the dependencies among them.

3.8 Dynamic Code Generation

The code patch is actually assembled in GILK from strings of assembly language instructions that represent the call sequences, some state saving instructions and the relocated sequence. The assembly language instructions for individual call sequences are obtained from separate files on disk. The assembly language for the relocated sequence is obtained by first disassembling the instructions marked for relocation and then passing them through a filter, which amends any program counter relative instructions, and generates the actual relocated sequence.

All parts of the patch are then concatenated together into a single string and fed into the dynamic code generation module. This module first assembles the instructions into object code using the GNU assembler “as” and then links them using the GNU BFD object file library to produce the actual machine code representation of the patch. This patch is then uploaded into kernel memory via the device driver, which is discussed later.

3.9 The Instrument Function

In order to get an idea of what an instruction function is and what it does, a simple instrument function that counts the number of times it has been called is presented:

```
void counter_instrument(unsigned int iid)
{
  static int i=0;
  ilk_log_integer_sample(i++,iid);
}
```

The function accepts a single parameter, the unique instrument identifier. It then generates a sample by logging the current call count with the identifier. The instrument function is loaded into the kernel as a kernel module. What this means is that the task of dynamically linking its symbols is performed by the kernel, which saves some complication in GILK.

The instrument functions are free to use any kernel functions available to normal device drivers. This raises a potential pitfall, which GILK fails to prevent. If an instrument function makes use of a kernel symbol that is being instrumented using itself, an infinite loop will be setup which may cause the kernel to grind to a halt.

3.10 The ILK Device Driver

ILK is the name given to the device driver component of GILK. The job of this component is to collect samples from all the active instruments and to provide a means for those results to be copied from kernel space to user space. It also provides an interface to some kernel functions, such as kmalloc which is used for allocating the kernel memory for the code patches.

This is an important part of the instrumentation tool. Throughput is an issue, so it must be able to cope with a reasonably high rate of sample generation.

In order to keep the sample collection efficient, a design decision was made that fixed the type of sample that could be logged to a tuple of the instrument identifier and a single value. It would have been nice to allow more than one value to be logged with each identifier. For example, suppose the parameters of a function were to be recorded. In this case, it would be ideal if all the parameters were logged with the instrument identifier as a single sample. Instead, each parameter must be separately logged, generating multiple samples. This introduces a certain amount of waste and complicates analysis of the samples.

ILK uses an array of fixed size structures to store the samples as they are generated. This array is used as a cyclic buffer with ILK writing samples into it and GILK reading them out
.
This results in a complication, which GILK does not attempt to address. If the sample generation rate is high, then GILK will be working hard copying data from the device driver. This is a relatively expensive procedure as it involves a context switch. This additional load on the system may well affect the samples generated. The options are limited on how to decrease the impact on performance from this copying. The samples could be stored in kernel memory for the duration of the instrumentation. However, it is not uncommon for five MegaBytes (or more) of data to be generated for only a sixty second instrumentation. Clearly, storing all of that in unpageable kernel memory would hit upon the performance of the system. Alternatively, a large buffer could be used that gets flushed with a low frequency. However, existing implementations of this method suggest that performance is still affected, as the copying of samples to user space is more furious than before.

3.11 Sample Collection

To collect the samples being generated during instrumentation, GILK forks of a separate child process whose sole purpose is to continuously copy samples from user space and write them to disk. The idea behind this is that it allows the user interface of GILK to continue working without affecting the amount of processor time devoted to copying samples from the device driver.

The file generated by the “collector” child-process contains a mapping from major instrument numbers to kernel symbols, as well as the samples generated. This file, together with the control flow graphs of the relevant kernel symbols, can be used to generate a picture of how the measured items changed throughout the instrumentation period.

4 Analysis of GILK

This section will present the results obtained from the analysis of GILK. The reason for performing the analysis on GILK was to establish a level of confidence in it. The first half of the analysis was concerned with showing that the results produced by GILK are valid. The second half was interested in obtaining a figure for the overhead imposed on system performance by instrumentation with GILK.

4.1 Correlating with IKD

In order for an instrumentation tool to be used to influence the design of a software program, there must exist a certain level of confidence in the results generated by the tool. In other words, if the tool cannot be trusted to generate valid results why would anyone use it? To this end, some considerable effort was undertaken to show that at least some of the results generated by GILK are valid. To show that all results generated by GILK were valid would require either generation of all possible results (very impractical) or a mathematical proof that showed an invalid result could not be generated. The author speculates that the latter is intractable due to the infinite combinations in which the tool can be used and the infinite set of results which can be produced for each combination.

The aim of the experiments outlined in this section was to show a correlation between measurements taken by GILK and those by another tool, called IKD, for the same sequence of events. This goes someway towards proving that measurements taken by GILK are valid.

4.1.1 A brief introduction to IKD

IKD [15] is a set of utilities that instrument certain aspects of the Linux kernel. It is supplied as a kernel patch that must be applied to the source tree before instrumentation can be performed. The tool is capable of a number of different instrumentation modes. However, for the purposes of this section only the "ktrace" mode is of interest as only this mode can produce results which can be compared with those of GILK. Further information regarding configuration of the tool to produce data comparable with GILK can be found in Appendix A.

4.1.2 Design of the experiment

The idea behind this experiment was to get both tools to record the time at which each event, from a known sequence of events, took place. This would allow a direct comparison of the timings made by each tool and hence, a conclusion regarding any correlation between them could be made. This **event sequence** would ideally be free from any random factors, as the presence of such factors could prevent a comparison being made.

Therefore, the outline of the experiment was to boot into a clean system, set the tool measuring and then play the event sequence. This was performed for both GILK and IKD and produced two data sets (one for each tool) that were directly compared for discrepancies. The period of time during which measurements were being taken is, henceforth, to be termed the **measurement period**. Thus, the experiment contained two measurement periods: one for GILK and one for IKD.

4.1.3 The Event Sequence

As already mentioned, the IKD tool has only one mode of operation, the ktrace mode, which can produce results comparable with those of GILK. This mode records timestamps for **entry events**. An entry event occurs when control enters a function. GILK is capable of measuring many types of event, including the entry event. The entry event was therefore the common event type that could be measured by both GILK and IKD. Therefore, the event sequence had to be comprised purely of this event type. To simplify matters, only the entry events for a single function were to be timed.

The function selected for this was "sys_fork", which is called when a process splits itself in two. The exact details of this function are not relevant, and a concise account of its inner workings can be found in [3]. Sys_fork is particularly suitable because it has a certain property. This property is that, when the system is idle, sys_fork is

never called unless a user (indirectly) requests it to be[3]. This means that during the measurement periods and with user interaction prohibited, a single program can be used to generate the event sequence by calling sys_fork via the Linux ‘C’ function “fork”.

The program used for this purpose was called Pulsar and its source code is shown in figure 4.1. Pulsar consists of five iterations of a loop body, which has three distinct phases of execution. Firstly, thirty-two processes are forked off one after another (hence, thirty-two calls to sys_fork are made). Then, Pulsar “sleeps” for five seconds. Finally, it collects up all the completed processes and begins a new iteration. These shall henceforth be respectively referred to as the working, sleeping and collection phases.

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
  int pidarray[32],i,j,status;

  for(j = 0; j < 5; j++)
  {
       for(i = 0; i < 32; i++) {
       if((pidarray[i] = fork()) == 0)
            exit(0);
    }

  sleep(5);

  for(i = 0; i < 32; i++) {
       waitpid(pidarray[i],&status,0);
    }
  }
exit(0);
}
```

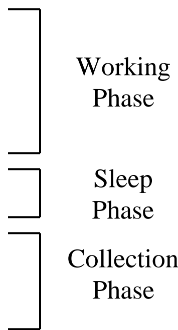

Figure 4.1 - Showing the ‘C’ code for the Pulsar program. The three phases of the inner loop are marked on the right hand side.

4.1.4 Examining the Results

The results from the experiment (described above) are illustrated in figure 4.2. They consist of two sets of samples. Each sample is a timestamp for an entry event of sys_fork and a sequence number. The sequence number identifies the exact execution of the fork statement (in pulsar) that caused the entry event to sys_fork. Pulsar will generate 160 entry events as there are 160 executions of its fork() statement. Therefore, the sequence numbers range from 0 to 159 with 0 being the first execution and 159 being the last execution of fork. The sequence number can be related to the loop in Pulsar through the formula sequence number = (j*32) + i, where j is the outer loop variable and i is the loop variable for the first inner loop.

In other words, each execution of the fork statement in Pulsar results in sys_fork being invoked. This, in turn, causes the instrumentation tool to generate a timestamp, which is then combined with the fork statement’s sequence number to produce the final sample. Both sample sets have been plotted on the graph, so that they can be compared. Hence, by examining figure 4.2, it should be clear that, although the two sample sets are very similar, there exists a slight discrepancy between them.

[3] Sys_fork is actually called by the shell during system initialisation and also by various system daemons such as cron. However, the daemons can be prevented from disrupting the experiment and the measurement period always takes place after the system has been initialised. This means that these variables can be safely ignored.

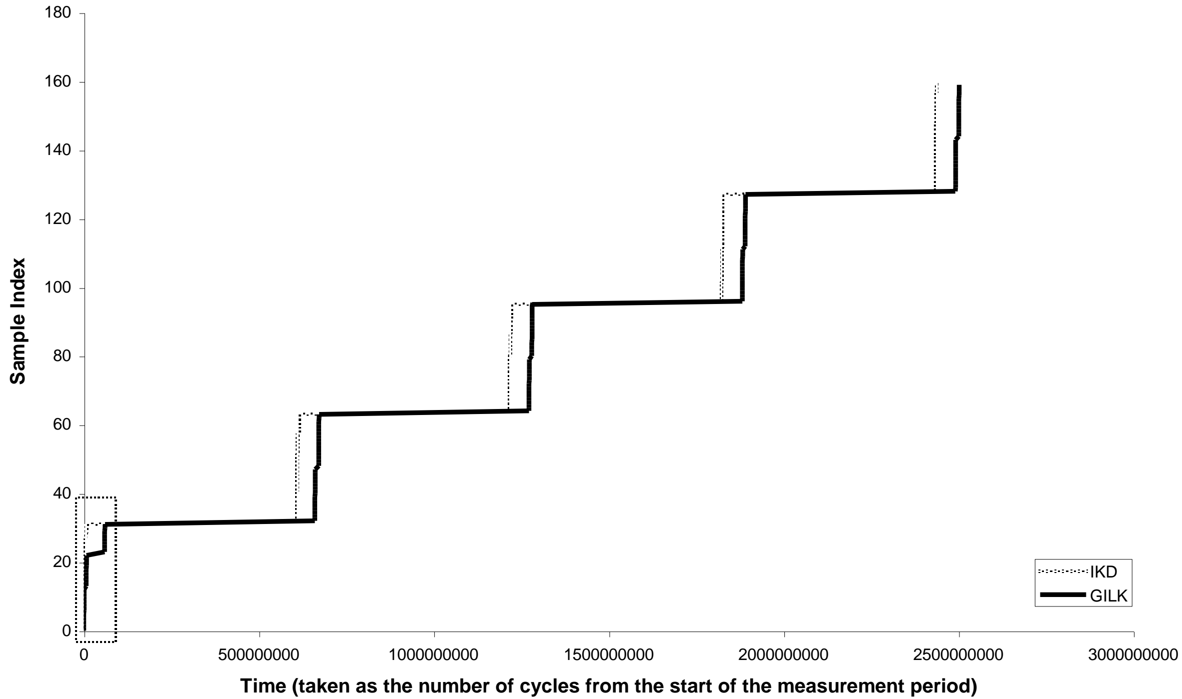

Figure 4.2 - Showing the two (complete) sample sets, generated by IKD and GILK during their respective measurement periods, superimposed onto each other. For each curve the five main vertical lines represent the five working phases of Pulsar. The four main horizontal lines represent the first four sleep phases of Pulsar. The region enclosed by the dashed line is displayed in figure 4.3.

Figure 4.3 highlights this discrepancy by showing the samples from both sets that correspond to the first working phase of Pulsar. What this tells us is that GILK has observed something different from that observed by IKD. The question is: why? Two hypotheses were developed as possible answers to this question. Firstly, the instrumentation method employed by GILK could be introducing these differences. The focal point of this hypothesis revolves around the sample logging mechanism, as this accounts for the majority of the overhead caused by GILK's instrumentation method.

The alternative hypothesis came from noticing that in figure 4.3, IKD actually has a similar, albeit less dramatic, curve to that of GILK. This leads to the conclusion that the effect causing the discrepancy is also measured by IKD. However, the size of the effect is much greater when GILK is being used for measuring. The suggestion is, therefore, that the effect being measured is that of process scheduling. In other words, Pulsar is scheduled off the processor before it can complete the work phase. However, to account for the discrepancy there would need to be a process present whilst measuring with GILK, but not with IKD, that was consuming a lot of processor time. If this were the case, the time between Pulsar being scheduled off and on the processor would be larger under GILK than IKD.

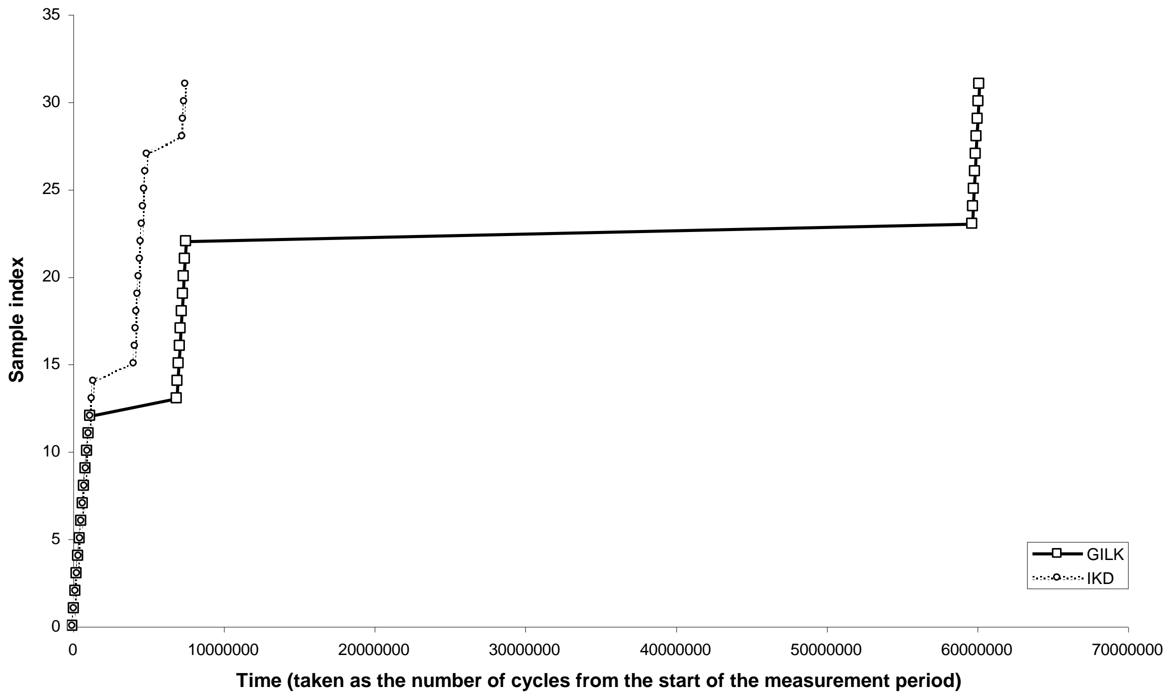

Figure 4.3 - Showing a close up of the first thirty-two samples from each of the two sample sets presented in figure 4.2. This graph highlights one of the discrepancies observed in figure 4.2.

This second theory, being the preferred one, was further investigated by configuring GILK to record the times when process scheduling takes place, as well as the entry events of sys_fork. It would have been ideal to perform this second experiment with IKD, as well. However, IKD is in no way suitable for such a sensitive measurement of the Linux kernel. The results of this investigation are illustrated in figure 4.4. This graph, more or less, verifies the second hypothesis. Firstly, examination of the graph reveals that scheduling is taking place during the first working phase of Pulsar. Secondly, there are large periods of time when no scheduling occurs and Pulsar does not hold the processor[4].

The significance is that these periods represent processor time allocated to other processes and because the intervals are large, it can be concluded that these processes are intensively using the processor.

A little bit of detective work soon uncovered the source of the problem. The GILK tool was forking of a child process to collect the generated results. A small piece of inter-process communication was occurring between GILK and its child to signal the end of measurement. This communication was using, for various reasons, non-blocking I/O which resulted in a busy-wait loop in both GILK and its child. A minor revision to GILK was made to remove these loops and the original experiment was performed again to confirm that this was the source of the original discrepancy. The new results are presented in figure 4.5. It should now be clear that there exists a correlation between the results produced between IKD and those from the modified version of GILK.

[4] The time intervals that Pulsar holds the processor are those within which samples are clustered. So, from figure 4.4 it can be deduced that Pulsar holds the processor for three separate time intervals during its first working phase because there are three sample clusters.

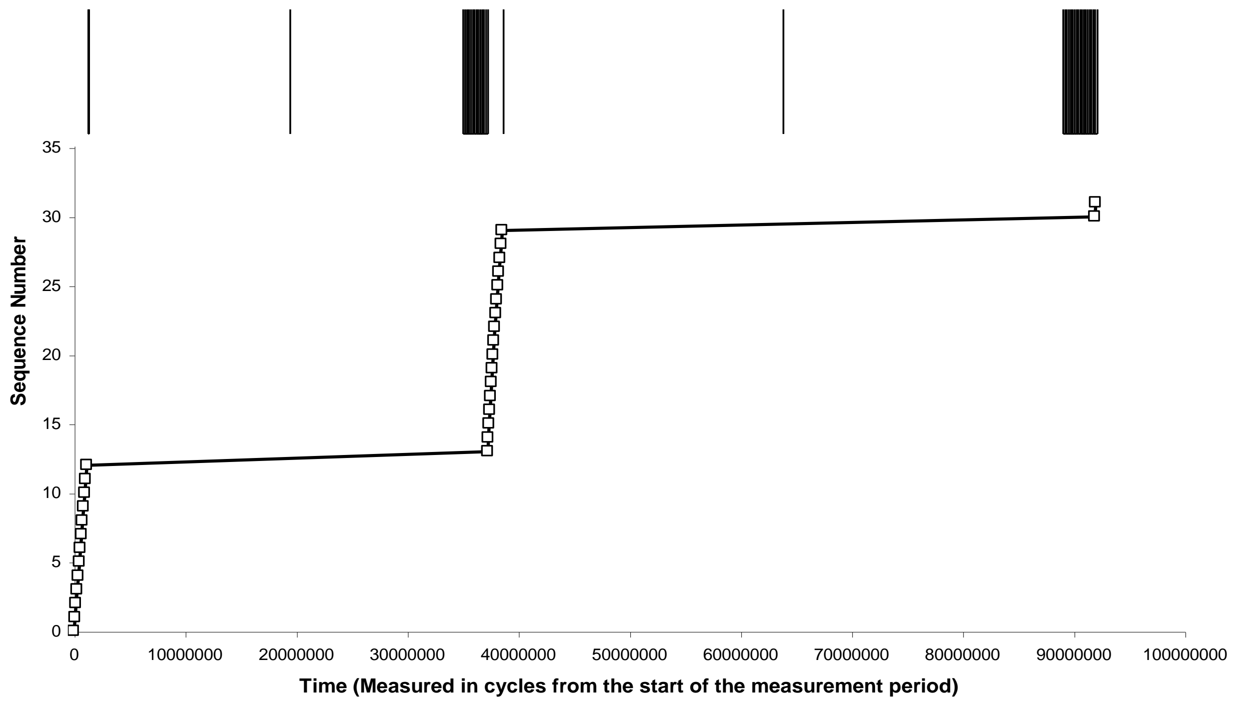

Figure 4.4 - showing the sample set generated by GILK for the second experiment performed. As for Figure 4.3 the curve shows only the samples that correspond to the first working phase of Pulsar. The lines plotted above the graph indicate the times at which process scheduling took place These are plotted with respect to the X-axis but have no meaningful Y value.

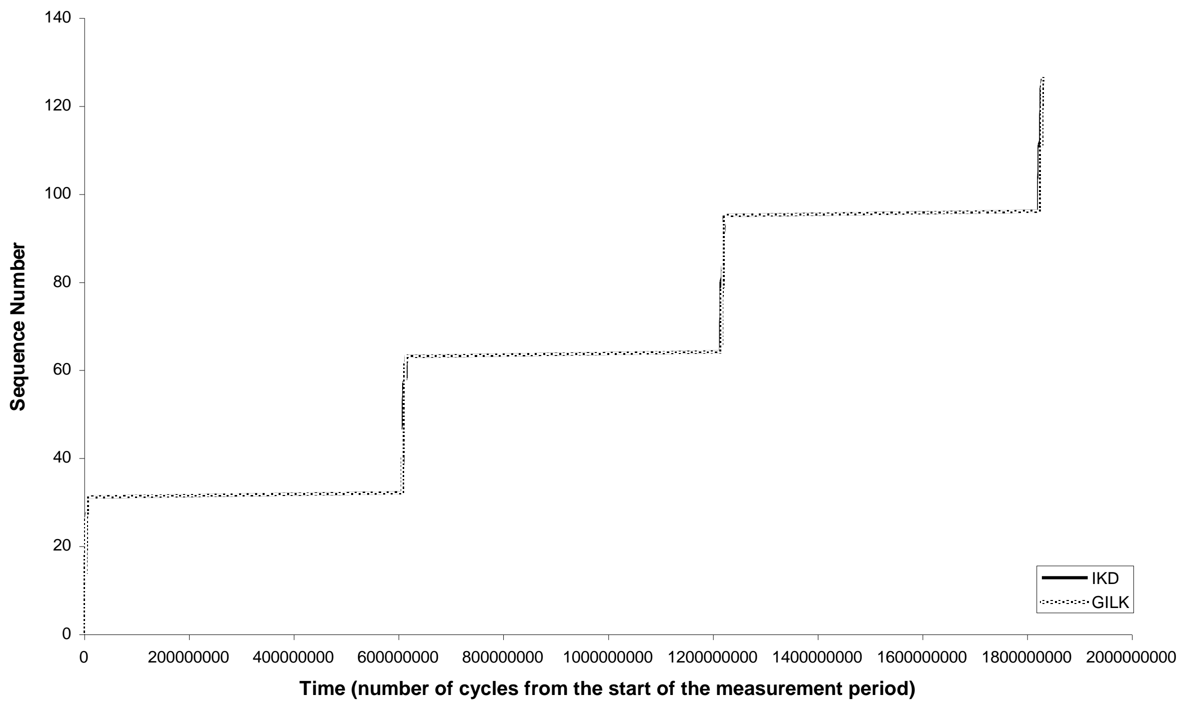

Figure 4.5 - showing that the measurements taken from the modified version of GILK match almost exactly with those taken by IKD

4.2 Measuring the overhead of Instrumentation with GILK

For an instrumentation tool, the act of taking measurements will have an impact upon system performance. If this is very large, the results generated by the tool are likely to be skewed. Therefore, it is desirable to have an idea of the magnitude of overhead imposed by the instrumentation tool. To this end, the experiments outlined in this section attempt to measure this overhead. In fact, the following experiments only measure the effect of inserting an instrumentation into the kernel. They do not attempt to account for the overhead caused by the data collection part of GILK and this would need to be done separately.

4.2.1 Design of the Experiment

The idea behind this experiment was to measure the time taken to execute a section of code with and without an instrumentation applied to it. The section of code chosen to be timed was (again) sys_fork. The reason for this choice was that a low variance in the execution time of this function had already been observed whilst undertaking the previous experiments.

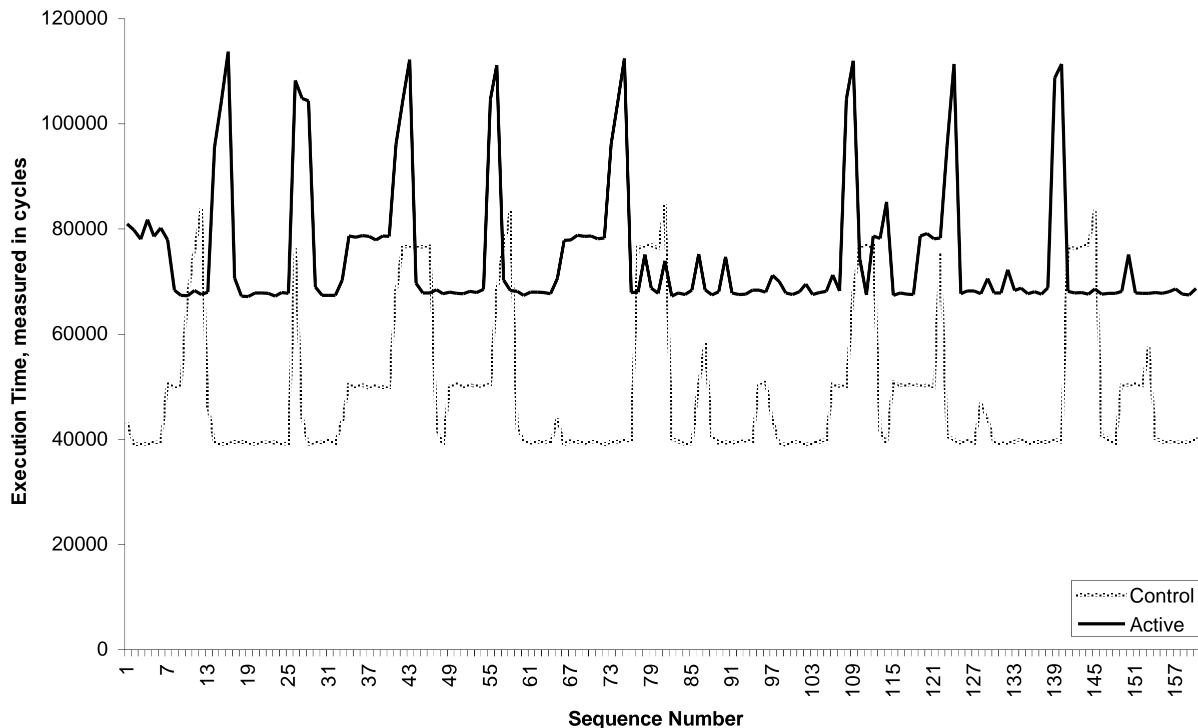

Figure 4.6 - showing the results from run three of both the control and active phases.
The Y axis marks execution times for the function sys_fork

The experiment consisted of the control and active phases. In the control phase, GILK was used to record the execution time of sys_fork and no other instrumentation was applied. In the active phase, GILK was again used to record the execution time of sys_fork. However, in this phase a simple instrument was also added to every basic block of the function "do_fork", which is the main worker function of sys_fork. In both phases, sys_fork was executed a number of times by running Pulsar. Each phase consisted of booting into a clean system and then measuring the execution time for six consecutive runs of Pulsar. The purpose of using Pulsar was merely to ensure that sys_fork was executed a sufficient number of times.

4.2.2 Examining the Results

The results from run three of each phase are plotted onto the graph in figure 4.6. The purpose of this is to highlight that a definite increase in the execution time of sys_fork was observed during the active phase. However, there are

many (random) peaks present in the graph, which are likely to be caused by various effects such as caching. Hence, it was decided that the overhead of instrumenting do_fork was represented in the graph by the difference between the baselines of both curves. This is not unreasonable as the magnitude of the peaks is the same in both curves

meaning that they are not attributable to the instrumentation. Moreover, they introduce a random element to the execution time of sys_fork and it is desirable to ignore them in order to reduce the variance of the execution times being compared.

Therefore, the minimal value recorded for both phases was determined and the difference between them attributed to the cost of instrumentation. These results are present in figure 4.7. It should be pointed out that the active phase added ninety-four instruments to the do_fork function. This is an unusually large number of instruments, which accounts for the large overhead. An estimate of the cost for an individual instrument was determined by dividing the measured difference by ninety-four. This gives the value of cost, per instrument added to do_fork, as 0.76%.

Results	
Measured Difference (in cycles)	27808
Overal Percentage Increase	71.82560182
Overhead per instrument (in cycles)	295.8297872
Percentage increase per instrument	0.764102147

Figure 4.7 - Showing the measured overhead for the instrumentation of do_fork. There were 94 basic blocks of do_fork that were instrumented. Hence the overhead per instrument is obtained by dividing measured difference by 94.

5 Reflection and Future Work

This section will examine the strengths and weaknesses of the GILK instrumentation tool. Also, any future work that could be undertaken on this tool and its instrumentation technique will be discussed. This will include the suggested improvements on GILK that have arisen as a result of creating and using it.

5.1 The strong points

The potential uses of GILK have only been touched upon in this document. For example, most of the experiments previously discussed only use one type of instrument, which records timestamps. The type of instrument is not limited to this, and can cover logging of function parameters, cache miss rates[5], T.L.B[1] miss rates, memory usage, stack usage, disk usage, processor usage and much more. In short, just about every aspect of the kernel can be monitored through clever use of this tool. No other tool has been encountered that can provide this range of performance monitoring solutions for the Linux kernel. In fact, the instrumentation methods actually used by the kernel developers are primitive by comparison.

There are two main features that make up the backbone of GILK. These are the dynamic code generation component and the specialised disassembler.

5.1.1 The specialised disassembler

The specialised disassembler provides the platform that makes the runtime code splicing method possible. It is used extensively to analyse the machine instructions that make up the Linux kernel and provides invaluable information, which could not be obtained from a standard disassembler.

5.1.2 Dynamic code generation

The method used by GILK to dynamically generate code is particularly interesting. By making good use of the GNU assembler 'as' and the GNU BFD object file manipulation library, it is capable of relocating any of the machine instructions that make up the kernel, with minimal effort. Additionally, instrumentation code accepted as a string of assembly language can be inserted at any point within a relocated chunk of instructions. This functionality is essential for the implementation of the runtime code splicing technique.

5.2 Improvements to GILK

The experience gained from using GILK to perform instrumentations has provided many ideas for suggested improvements. The main areas for improvement are sample generation and collection, co-ordination between instruments and presentation of results.

5.2.1 Sample generation and collection

The sample generation method has many problems and limitations, the major fault being the inability to cope gracefully with instruments that produce a large amount of data. This problem is twofold. Firstly, GILK doesn't support the types of sampling that can produce less data. For example, the periodic sampling technique, discussed in section X.Y, is used in Paradyn[8][9] to reduce the amount of data without affecting the accuracy of the samples. This, and other sampling strategies, should be incorporated into GILK to provide a more rounded instrumentation tool.

The second aspect of this fault is the poor strategy for collecting samples. At the moment, a process is forked off to continuously read the sample data. When the sample generation rate is high, this process will be consuming a large amount of processor time and its effect, already observed in the correlation experiment, will again be prominent. There are many different ways for dealing with this problem. However, none of the solutions will be ideal for all situations. Instead, it would be wise for GILK to provide the user with the ability to select the sample collection

[5] Through use of hardware performance registers specific to the Intel x86 processor line

strategy. This would mean that the optimal strategy could always be chosen. An additional benefit would be that the user could vary the collection strategy in an effort to decide whether or not it was responsible for an observed behaviour.

5.2.2 Instrument co-ordination

Another important fault with GILK is the complete lack of co-ordination between instruments. For example, suppose it was desired to measure the execution time of a function A, but only when it had been called by a function B. This could be achieved by instrumenting both functions and providing a shared (boolean) variable between them. The instruments in function B would then be configured to set the shared variable on entry to B and to reset it upon exit from B. Then, when the instruments in A were executed, they would check to see if that variable was set. If it was, they would record the time taken for A. If it wasn't they would simply do nothing. There are a few extra complications with this example, but it still serves to highlight the problem. Under GILK, this type of (very useful) instrument co-ordination is not possible.

5.2.3 Presentation of Results

This is a major flaw with GILK as it does not provide any capability of presenting the results. However, it is not really the inability of GILK to draw graphs that is the problem. This feature is probably best left to specific drawing packages. The real problem lies in the fact that GILK attributes no meaning to the data it produces.
Different types of instrument produce different types of data. For example, certain types of instrument produce timestamps whilst others produce counts or variable contents. The meaning of the data produced is not currently reflected in the results file produced. This leaves the job of first deciphering the meaning of the data and then choosing the best way of displaying it up to the user. If the user has a great deal of knowledge on the workings of GILK this won't be a problem, although it still takes considerable time. However, if the user doesn't have this knowledge then the task becomes very difficult.

Some instrumentation tools get around this problem by fixing the types of instrumentation that can be performed. A special filter is then provided for each type, which typically draws the data in an appropriate way on the display. The disadvantage with this is that a limitation has been imposed upon what the tool can actually be used for. Another approach would be to include "metadata" with the results produced. However, the ideal solution remains unclear and it is therefore, labelled "future work".

5.2.4 The Disassembler

The disassembler provides information about machine instructions that cannot otherwise be obtained. One of its uses in GILK is to disassemble machine instructions that are to be relocated. The assembly language output from it is eventually fed back into an assembler and turned back into machine instructions. This is the source of subtle bugs. If the disassembler produces the wrong assembly language string for a machine instruction, but the string is still valid assembly language, then the assembler will compile this without warning and the original error will go undetected. This wouldn't be a problem if the diassembler was perfect. However, this is quite difficult to achieve.

A simple solution to this problem presents itself. Under Linux there already exists a well tested disassembler. However, it cannot produce the extended information that is required by GILK. So, the output of GILK's internal disassembler should be compared with that of the standard disassembler to identify error cases.

5.3 Future Work

There are several aspects of the instrumentation method employed by GILK that could be extended upon. The main areas are:

- **Complete function relocation** – The idea is to completely relocate a function, instead of only parts of it. This would offer much more freedom as the function could then be completely rewritten to insert the instrumentation. However, there remains a large question regarding the possible problems and side effects of doing this. For example, instrumentation of a large number of functions would result in a lot of (unpageable)

kernel memory being allocated. Additionally, the space originally occupied by each function would, presumably, be wasted. Hence, work could be done to investigate the practicality of this idea.

- **Sample Collection** – As previously suggested, the sample collection strategy which has the smallest amount of system overhead remains undetermined. An investigation into the set of possible sample collection techniques could be performed, with an aim to identifying the best ones. This would provide GILK with a set of optimal strategies that could be chosen from for a given situation, and hence the overhead of instrumentation could be reduced.
- **Extension to user space** – Currently, GILK is limited to instrumenting the Linux kernel. However, many of the techniques it employs could be applied to user space processes. However, there remains some complexity regarding safe instrumentation and threaded applications. Both of these problems are currently solved by GILK with the aid of the methodology of the Linux kernel. In order for this extension to work, solutions to these problems need to be found.
- **Bounce Allocation Strategy** – As previously mentioned, the bounce allocation strategy employed by GILK is not as thorough as it could be. Work could be done to improve this. For example, bounces are not placed optimally and, as a result of this, the situation can arise in GILK where an instrument cannot be performed even though it is physically possible. Much of the framework for this already exists in GILK and it would not be a massive undertaking.
- **Indirect Branches** – Currently, GILK cannot instrument functions that contain indirect branches. Much work has been done on the problem of indirect branches and there remains scope for implementing these solutions into GILK. However, the number of kernel functions that actually contain indirect branches is very small and the advantage gained may be negligible.

6 Conclusion

This project has developed a sophisticated instrumentation tool for the Linux kernel called GILK, which is capable of inserting arbitrary instrumentation code before and after almost every basic block. Further, GILK has been shown to produce results that correlate with those of another instrumentation tool, IKD. GILK has however proved to be a dramatic improvement upon this tool.

The full potential of GILK has yet to be realised. Much time could now be spent using the tool to investigate the Linux kernel and uncovering any performance bottlenecks. Furthermore, there are a number of small tweaks that, if implemented, would make the tool a well-rounded performance monitoring system.

References

[1] J.R. Larus and T. Ball: *Rewriting Executable Files to Measure Program Behaviour*, University of Wisconsin Computer Sciences Technical Report 1083, March 25, 1992.

[2] Ariel Tamches and Barton P. Miller: *Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels*. Third Symposium on Operating System design and implementation, Febuary, 1999.

[3] M Beck, H Böhme, M Dziadzka, U Kunitz, R Magnus and D Verworner: *Linux Kernel Internals (second edition).* Addison-Wesley, 1997, ISBN: 0-201-33143-8

[4] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman: *Compilers, Principles, Techniques and Tools*. Addison-Wesley, 1986, ISBN: 0-201-10194-7

[5] J.M. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.-T.A. Leung, R.L. Sites, M.T. Vandervoorder, C.A. Waldspurger and W.E. Weihi. Continuous Profiling: *Where have all the cycles gone?* 16th ACM Symposium on Operating System Principles (SOSP), Saint-Malo, France, Oct. 1997.

[6] J. Larus and E.Schnarr: *EEL: Machine-Independent Executable Editing*. In the proceedings of the ACM SIGPLAN '95 Conference on Programming Languages Design and Implementation, pages 291-300, June 1995.

[7] Han Bok Lee and Benjamin G. Zorn, *BIT: A Tool for Instrumenting Java Bytecodes*. In the proceedings of USENIX Symposium on Internet Technologies and Systems pages 73-82, December 1997.

[8] Barton P. Miller, Jeffrey K. Hollingsworth and Mark D. Callaghan: *The Paradyn Performance Tools and PVM*. www.cs.wisc.edu/~paradyn/

[9] Barton P. Miller and Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam and Tia Newhall: *The Paradyn Parallel Performance Measurement Tools*, www.cs.wisc.edu/~paradyn/

[10] *Aprobe: A Unique Testing & Debugging Tool*, www.ocsystems.com

[11] Daneil F. Zucker and Alan H. Karp: *RYO: a Versatile Instruction Instrumentation Tool for PA-RISC*, Technical Report: CSL-TR-95-658, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, January 1995

[12] Xiaolan Zhang, Zheng Wang, Nicholas Gloy, J. Bradley Chen and Michael D. Smith*: Operating System Support for Automated Profiling and Optimization*, In the proceedings of the 16th ACM Symposium on Operating Systems Principles, St. Malo, France, October 1997

[13] Gprof, the free software foundation, www.gnu.org.

[14] KernProf: *A set of facilities for profiling the Linux kernel*, http://oss.sgi.com/projects/kernprof/

[15] IKD: *Assorted tools for debugging the Linux Kernel*, ftp://e-mind.com/pub/linux/patch-ikd-arca/

[16] Barton P. Miller, R. Bruce Irvin and Jeffrey Hollingsworth: *The Integration of Application and System Based Metrics in a Parallel Program Performance Tools,* In the proceeds of ACM SIGPLAN 1991, Symposium on Principles and Practise of Parallel Programming.

[17] James R. Larus and Thomas Ball: *Efficient Path Profiling*, 1996.

7 Appendix A

7.1 Ktrace

The ktrace functionality of IKD provides call tracing of all kernel functions, except those that relate to scheduling. For each function call made, ktrace records the program counter, a timestamp and (optionally) a process identifier. The accuracy of the timestamp depends upon the host architecture. On an Intel x86 system, the RDTSC instruction is used and this provides timestamps measured in cycles.

Ktrace is implemented through a direct augmentation to the kernel and makes good use of GCC's "-pg" switch. Recall from section 2.1.2 that this compiler switch automatically inserts a call to "mcount" at the beginning of each function. The body of mcount is normally obtained by linking with the standard C Libraries. However, the Linux kernel is not linked with these libraries. Instead, IKD provides an implementation of this function, which records the required data into a cyclic buffer. The size of this buffer is determined at compile time and overflow results in the oldest samples being overwritten.

The buffer is read through a special file, "/proc/trace". A small utility is provided that reads the current state of the buffer and formats it for output to a terminal.

7.2 Adapting ktrace for accurate measurement

The instrumentation technique used by IKD has some implications upon the granularity of functions which can be instrumented. Out-of-the-box, IKD is configured to instrument every function of the Linux kernel. This generates a tremendous amount of data and places a significant overhead on the kernel. More importantly, the internal buffer overflows very regularly, which means that a significant number of results are lost. A statistical sampling method could be employed here to cope with the arbitrary loss of results. However, GILK does not use a statistical sampling method, hence, a direct comparison would require a lossless data set from IKD. To achieve this, it was necessary to reduce the number of functions being instrumented by IKD to only those required.

The set of kernel functions instrumented by IKD is mostly determined by which source files are actually compiled with the '-pg' switch. Additionally, there is a small number of kernel functions which have had the calls to mcount manually applied. This, along with the fact that a single source file typically contains several functions, makes instrumenting a single function quite difficult. The solution employed was to disable instrumentation on all functions, including the exceptional cases. Then, the required instruments were inserted by manually adding the calls to mcount. Now, assuming the instrumented functions do not have a high call frequency, the buffer doesn't overflow regularly. This allowed a complete sequence of the (relevant) function calls, along with their timestamps, to be recorded by IKD.