GILK: A dynamic instrumentation tool for the Linux Kernel

A. Nonymous, B. Nonymous, C.Nonymous

No Institute Given

Abstract. *This document describes a novel instrumentation tool for the Linux Kernel and Intel x86 architecture. We describe how the tool allows a stock Linux kernel to be modified while in execution. A convenient graphical interface allows the user to browse the control-flow graph of each kernel function, and insert user-specified instrumentation before or after any basic block. The instruments are implemented as kernel modules and, as such, are written in 'C' with access to all parts of the kernel. The Intel x86 architecture poses a particular problem, due to variable length instructions, which this paper addresses for the first time. We evaluate the potential of the tool for performance instrumentation, and compare it with an alternative. Finally we present a short case study illustrating its use in understanding i/o behaviour in the kernel. The source code is freely available for download.*

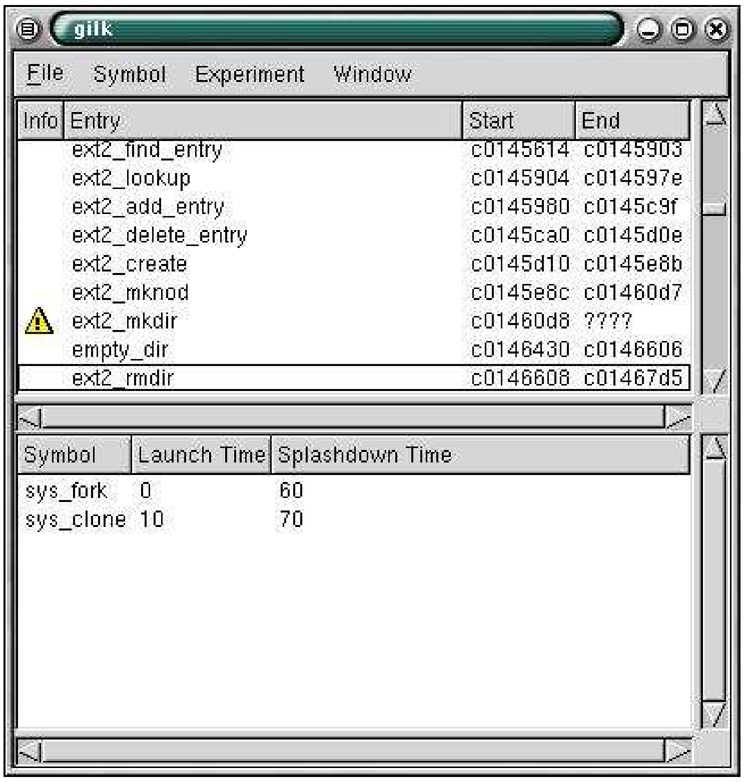
1 Introduction

In this paper we describe an instrumentation tool called GILK that has been developed specifically for the Linux Kernel. It permits sensitive instrumentation code to be added to an unmodified kernel in execution with low instrumentation overhead. This is achieved through an implementation of *runtime code splicing*, which allows arbitrary code to be inserted before most machine instructions in the kernel without affecting its behaviour.

Currently the tool works only for kernels running on the Intel x86 architecture, although in principle there is no reason why it could not work on others. Through a graphical based interface (see Figures 1,2), the user may choose how and where to instrument, when to begin and end individual instruments and what to do with the information produced by them.

We have found it to be useful in a variety of applications, including measurement of disk and network activity. In the latter, it has been used to show that a "power law" exists in network traffic (see section 4.4). We have also verified the correctness of the tool by comparing results with those generated by another instrumentation package, IKD [1]. We make the following contributions:

- An implementation of runtime code splicing for the Intel x86 architecture is outlined.
- A new technique for code splicing, called *local bounce allocation*, is described.



Info	Entry	Start	End
	ext2_find_entry	c0145614	c0145903
	ext2_lookup	c0145904	c014597e
	ext2_add_entry	c0145980	c0145c9f
	ext2_delete_entry	c0145ca0	c0145d0e
	ext2_create	c0145d10	c0145e8b
	ext2_mknod	c0145e8c	c01460d7
⚠	ext2_mkdir	c01460d8	????
	empty_dir	c0146430	c0146606
	ext2_rmdir	c0146608	c01467d5

Symbol	Launch Time	Splashdown Time
sys_fork	0	60
sys_clone	10	70

Fig. 1: Illustrating the main window of GILK. In the upper pane, the kernel symbols are shown with markers indicating the result of the analysis performed on them. The triangle marker shown indicates that the analysis was unable to complete. This may be, for example, as a result of finding an instruction that couldn't be understood by GILK's disassembler. In the bottom pane, we see the instruments that make up the current experiment. They will not be activated until some point after the experiment is begun.

– The accuracy and overhead of the splicing method is evaluated through experimental work.

The remainder of this paper is as follows. Section 2 will briefly examine some of the related work. Section 3 will overview the implementation behind the instrumentation tool, while section 4 will examine some experiments performed using the tool. Section 5 will conclude.

2 Related Work

Much of the foundation for this project has been laid by Tamches, *et al.* [15, 16, 14]. They have developed the *KernInst* dynamic instrumentation tool for the

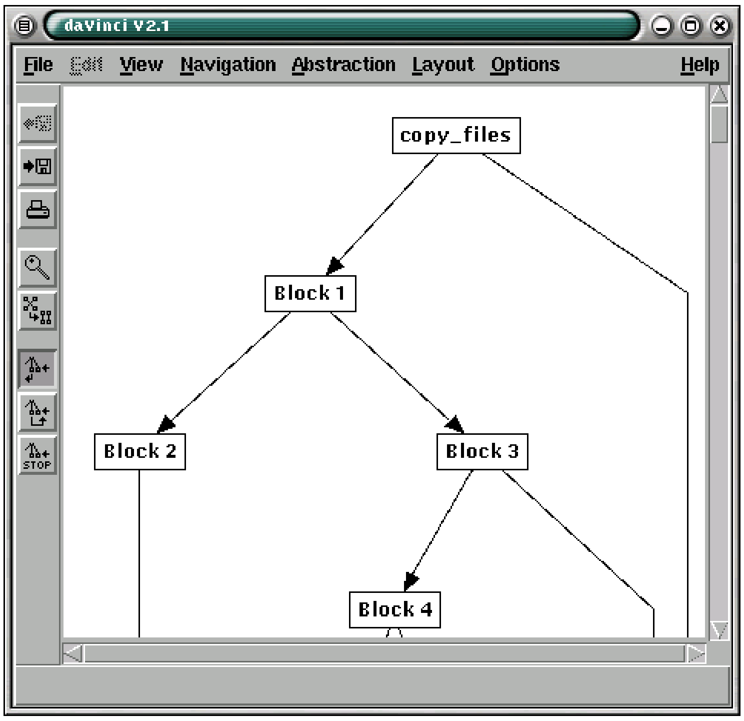


Fig. 2: Illustrating a control-flow graph generated by GILK. It is displayed using the graph visualization tool daVinci [5]. The graph for any kernel symbol is accessible via a menu option on the main window.

Solaris kernel and UltraSparc architecture. It works in a very similar fashion to GILK, with two notable exceptions. Firstly, it performs a more aggressive analysis of kernel symbols, in particular an interprocedural live register analysis. This attempts to minimise the impact of instrumentation through unused registers. Secondly, the techniques it uses are geared towards a fixed length instruction set and a multi-threaded kernel. In contrast, the Intel x86 architecture uses a variable length instruction set and the Linux kernel is single threaded. These topics will be discussed in more detail in section 3.1. The main illustration of their tool in use is through the performance study of a web proxy server, where a design flaw is identified.

Another dynamic instrumentation tool is Paradyn [11, 6], which instruments user programs. Paradyn's approach is somewhat different to KernInst, in that it ensures safety by walking the stack before each code splice. This causes a performance overhead, which would be a problem only if instruments are activated frequently.

Binary rewriters such as QP/QPT [8], EEL [9], BIT [10] and ATOM [13] introduce instrumentation code by modifying the executable statically. This is, arguably, a safer approach than runtime modification. But it is more cumbersome, especially for kernels, as it is often unclear what needs to be instrumented beforehand. With a dynamic system one can quickly switch between different instruments before settling on the right one. On the other hand, for a binary rewriter operating on a kernel one either instruments many symbols in one go, hoping to catch some information of interest, or endures a large number of system reboots as the exploration proceeds. In the former case, the experiment may also need to be rerun with a narrower instrument set in order to reduce the performance impact imposed. Finally, the static approach prevents instrumentation of systems that cannot be restarted.

It is our belief that GILK provides a more practical solution to instrumentation than Binary Rewriting. This stems from the fact that, being a *dynamic* system, it is more suited to the exploratory nature of performance monitoring and debugging. Nonetheless, the techniques used in static rewriters are extremely relevant to this work.

There are a number of existing instrumentation packages for the Linux Kernel. These include the IKD patch [1] and KernProf [2]. These are both implemented as kernel patches and, therefore, need to be applied to the kernel source and then compiled. In both cases, instrumentation is inserted by compiling with the '-pg' switch of GCC. This provides a rather coarse form of instrumentation as all symbols within a file compiled with '-pg' will be instrumented. Additionally, it is difficult to target instrumentation without fiddling around with the kernel's build procedure. KernProf is the more powerful of the two and is capable of *statistical sampling*, call path tracing, call counting and more.

Statistical sampling is another valuable technique for instrumentation. DCPI [3] provides a good example of this. Such systems arrange for the target to be interrupted infrequently, so that the Program Counter (or other registers) can be recorded along with a timestamp. Thus, a picture of where the program is spending its time will emerge as more samples are collected. Typically this method has a very low overhead and it is true that this will almost always be less than that of GILK. However, statistical sampling is also somewhat limited in the data that it can record. For example, function execution times can only be approximated. But, we do not dismiss statistical sampling. Instead, we feel that it is complimentary to dynamic instrumentation and GILK.

3 GILK Overview

The GILK tool consists of two main components: the device driver (called ILK) and the client. The client does the majority of the work, whilst the device driver simply provides the client with access to kernel space. The client begins with a scanning phase where all kernel symbols are examined in turn. For each symbol, the tool attempts to determine whether it is safe to instrument or not, which involves generating the Control Flow Graph and performing some rudimentary analysis (see section 3.5). This also helps identify any data symbols which cannot otherwise be distinguished. Those which are determined to be unsafe are then marked in the GUI and blocked from instrumentation. The user is then permitted

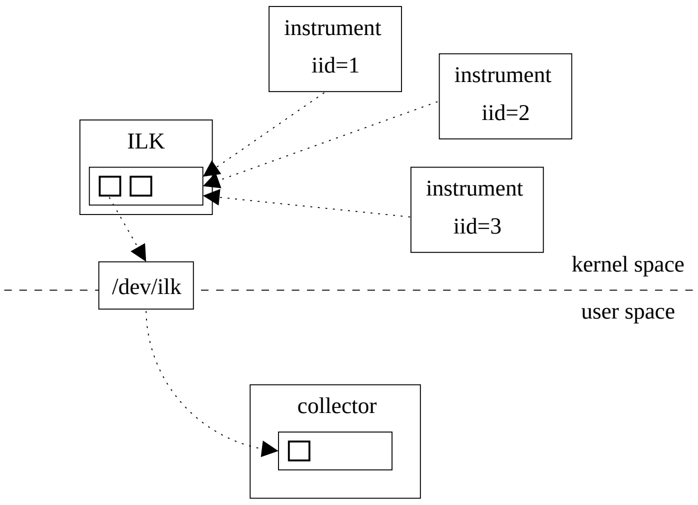


Fig. 3: Illustrating how samples flow through the system. As the kernel executes, the instruments obtain control and log their samples with the device driver. A user space process, the collector, periodically empties this buffer into user space via the device interface /dev/ilk. In turn, the collector writes the samples to disk.

to specify what instrumentation should take place. This amounts to choosing target locations, selecting instruments for them and specifying start and finish times. The tool supports staggered launching and termination of instruments to provide greater flexibility and to help reduce the impact on kernel performance.

The tool itself is capable of adding a *pre-hook* or a *post-hook* instrument to any basic block within a function. A pre-hook instrument is one that gets executed before the first instruction of a basic block. A post-hook instrument is executed after the last non-branching statement of a basic block. Hence, a target location consists of a symbol, an identifier and a flag indicating a pre- or post-hook instrument. The process of assigning instruments to locations is helped by the use of strategies, which capture typical instrumentation plans. For example, one strategy is to insert the same pre-hook instrument to each basic block. This would be useful for execution path tracing. Another is to place a pre-hook instrument before the first basic block and a post-hook instrument after each terminating block. If the instrument recorded timestamps we would be able to measure execution time for the function in question.

Figure 3 provides an overview of the sample generation and collection phase. Each of the instruments is assigned a unique identifier (iid) which they log, along with any additional data, to the device driver. The device driver buffers these samples and a user space process, the collector, periodically copies them from kernel space via the standard UNIX device interface. They are then written to

before	after
movl 60(%esp,1),%ecx	jmp patch
testb $0x10,%ch	db c5
jz 24	db 10
...	next:
	jz 24
	...
	patch:
	pushfl
	pushal
	pushl $iid
	call instrument
	addl $4,%esp
	popal
	popfl
	movl 60(%esp,1),%ecx
	testb $0x10,%ch
	jmp next

Fig. 4: Illustrating before and after a code splice. What we can see is that in the second code sequence the branch instruction has been placed over the first two instructions of the original. This has made two bytes redundant, as they are no longer in the flow of control. The patch itself forms a pre-hook instrument with the two overwritten instructions relocated to the back.

disk. The user space client keeps a record of the active instruments so that the kernel can be safely restored to its original form.

3.1 Code Splicing

The aim behind code splicing is to place a branch instruction (the splice) at the target location (the splice point) connecting to the instrumentation code. Clearly, the branch will overwrite a number of instructions and, therefore, those affected are first relocated into a *code patch*. The splice points to this code patch which contains any relocated instructions, code to save and restore the machine state and a call to the instrument function itself. This is illustrated in Figure 4.

For a fixed length instruction set, this works pretty much as is. However, for a variable length instruction set, such as the Intel x86, there is a slight complication. The branch instruction used is 5 bytes in length. However, an instruction may be as small as a single byte in length. Therefore, it is entirely possible that the branch may overwrite more than one instruction. Now, consider what will happen if, for example, the second instruction is the target of a branch elsewhere in the code. When that branch is taken, control will be passed into the middle of the splice and not to the original instruction! It is for this reason that GILK must generate the Control Flow Graph for each symbol. With this knowledge the above problem can be reduced to saying that it is unsafe to place the splice across a basic block boundary. In fact, GILK is slightly more

before		after	
opcode	instruction	opcode	instruction
`55`	pushl %ebp	`e9b1080000`	jmp c0116918
`57`	pushl %edi	`4424`	(unused)
`56`	pushl %esi	`244f`	(unused)
`53`	pushl %ebx	`4fff`	(unused)
`c74424244f4ffffff`	move $fffffff4, 36(%esp,1)	`ffff`	(unused)

Fig. 5: Illustrating eight bytes that are free for use as a local bounce. These unused bytes stem from the fact that the branch instruction has completely overwritten the first four instructions and the first byte of the fifth.

restrictive than this, by allowing only pre-hook and post-hook splices. This is simply to make the number of target locations more manageable.

There is, however, a second problem with variable length architectures that is similar to the first. This time, consider what happens if a thread is suspended at an overwritten instruction. Again, when the thread awakens, control could be passed into the middle of a branch. At this point, the methodology of the Linux Kernel comes to the rescue. There are three main points:

– *A process executing in kernel space must run to completion unless it voluntarily relinquishes control.* This means that when a process is in kernel space it will not be scheduled off the processor when its timeslice expires.
– *Processes running in kernel space may be interrupted by hardware interrupts.* Although this may appear to contradict the first rule, it does not because control is always returned to the process. Additionally, it is possible to prevent interrupts from occurring in specified sections of code.
– *An interrupt handler cannot be interrupted by a process running in kernel space.* This fact ensures that any process executing in kernel space always regains control after the interrupt *and before any other process.*

These three points, taken together, allow us to overcome the remaining problem with variable length instruction sets. Firstly, the ability to block interrupts means that the device driver can write the branch instruction to any part of the kernel atomically. Secondly, although a process may relinquish control it can only do so through indirectly calling the `schedule()` function. This means that, so long as we don't instrument this function, the sleeping thread problem can be ignored. Further discussions on these topics can be found in [4].

3.2 Local Bouncing

There is an interesting problem with the use of a five byte branch instruction for splicing: A basic block may be less than five bytes in length! In this case, we cannot use such a large branch instruction, without straddling a block boundary. However, the Intel instruction set also supports a two byte branch instruction, which has a range of only -128 or +127 bytes. In general, this is not enough to reach the code patch.

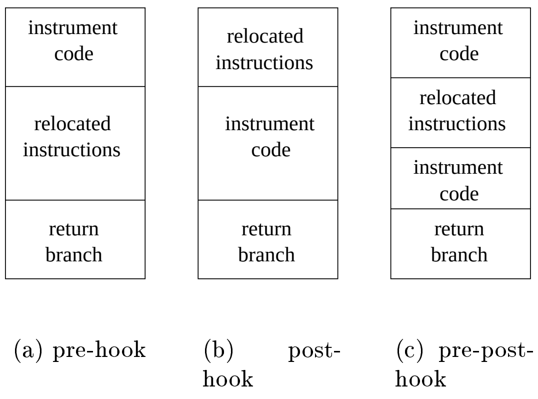


Fig. 6: Illustrating the three patch configurations. In the pre-hook case, the relocated instructions from the original code sequence are placed after the instrumentation code. In the post-hook case the reverse is true and for the pre-post-hook it sandwiches the relocated instructions.

Therefore, GILK attempts to place a five byte branch to the instrument within this limited range. This is termed *bouncing*. The problem, then, is where to position these bounces. Due to the nature of the Intel instruction set it is quite often the case that bounce sites are made available by splices for other instruments. This is illustrated in Figure 5. GILK will attempt to use any of these "free" bounce sites when possible. If none are available then GILK will relocate instructions *solely for the purpose of finding space.*

This strategy is termed *local bounce allocation* because GILK only attempts to allocate bounces within the function being instrumented. It is a novel feature of the tool.

3.3 The Code Patch

There are three configurations for the code patch: *pre-hook, post-hook* and *pre-post-hook*. In the first case, the instrumentation code is positioned before the relocated instructions. This is the configuration used in the code patch from Figure 4. In the second case, the instrumentation code is located after the relocated instructions. An exception to this occurs when the last instruction is a branch or return. In this case, the instrumentation code is sandwiched between the main body of relocated instructions and the last one. The final case occurs when the entire basic block must be relocated and both pre-hook and post-hook instruments are requested on it. Figure 6 illustrates these configurations.

The code patch is actually assembled in GILK from strings of assembly language representing the relocated instructions and the instrument code. The relocated instructions are passed through a filter which fixes up any that are Program

Counter dependent. This may be as simple as adjusting the relative address or it may require a new instruction. The final assembly string is then passed through the GNU assembler 'as' and linked using the GNU BFD object file library to produce machine code, which can then be uploaded into kernel memory.

3.4 Instrument Functions

The instruments themselves are implemented as kernel modules, as this simplifies some of the dynamic linking issues. Each instrument module initially registers itself with the ILK device driver, providing a pointer to the *instrument function*. The instrument function accepts, as parameters, at least the unique identifier and possibly other arguments depending upon which code patch template was used. An example function is:

```
void simple_instr(unsigned int iid) {
 ilk_log_integer_sample(jiffies,iid);
}
```

This function simply logs the value of the global variable "jiffies" when it is called. This is interesting because, being a kernel module, it has access to all the structures of the kernel which is can report on. Also, being written in 'C' means that quite complex instrument functions can be constructed.

3.5 Indirect branching and other issues

Indirect branching is a problem for the GILK tool. This is because a dataflow analysis is required in order to generate the Control Flow Graph. GILK performs no such analysis and, therefore, it cannot conclude anything regarding the destination of indirect branches. Hence, a symbol which contains an indirect branch cannot be safely instrumented under GILK. In practice this is not much of a problem as indirect branches are rare. These problematic symbols are uncovered in the analysis phase of the tool and marked in the GUI.

4 Experimental Results

This section will cover some of the experimental work performed with the GILK tool.

4.1 Correlation with IKD

IKD [1] 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 supports a number of different instrumentation modes. However, for the purposes of this experiment only the "ktrace" mode is of interest. This mode provides call tracing of kernel functions and records the program counter, a timestamp and (optionally) a process identifier. The accuracy of the timestamp depends upon the host architecture. On Intel x86 systems, the RDTSC instruction is used, which provides timestamps measured in clock cycles.

The mode is implemented through use of GCC's "-pg" compiler switch. This mode inserts a call to "mcount()" at the beginning of each function. Normally, the code for `mcount()` is provided by the standard C libraries. However, the Linux Kernel is not linked with them and, instead, the IKD patch provides an implementation 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 method employed 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 in the kernel. This generates a tremendous amount of data and places a significant overhead on the kernel. More importantly, the internal buffer overflows regularly, meaning a significant number of samples are lost. In order to make a comparison with GILK, it was necessary to reduce the number of functions being instrumented by IKD. This was achieved by simply removing the "-pg" switch and placing the calls to `mcount()` manually around the functions in question, which allowed an unbroken sample set to be generated.

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. Therefore, the kernel function "sys_fork()" was selected as it is called infrequently during the normal course of events. A simple 'C' program, called *Pulsar*, was written which would invoke `sys_fork()` in a controlled fashion. The code for Pulsar is illustrated in Figure 7.

The experiment was performed on a Pentium 120Mhz running Linux 2.0.36. The results are illustrated in Figure 8. It shows two sample sets, with each sample being a timestamp and a sequence number. The timestamps are relative to the start of each sampling period and indicate when a call to `sys_fork()` was made. The sequence number indicates which `fork()` statement in Pulsar caused the timestamp. In total, Pulsar generates 96 `fork()` calls and, so, the sequence numbers range between 0 and 95. The graph has a distinct step shape which is caused by the `sleep(5)` call in Pulsar and it should be fairly clear that the results are nearly identical between IKD and GILK.

4.2 Performance Overhead

The aim of this experiment was to quantify the overhead caused by instrumentation. This was achieved by measuring 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 "do_fork()", which is called by `sys_fork()`, and the instrument to be measured was a simple timestamp recorder.

The experiment consisted of the control and active phases. In the control phase, execution times for `sys_fork()` were measured when no instrumentation

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

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

  for(j = 0; j < 3; 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);
}
```

Fig. 7: Showing the 'C' code for the Pulsar program, along with the phases of execution.

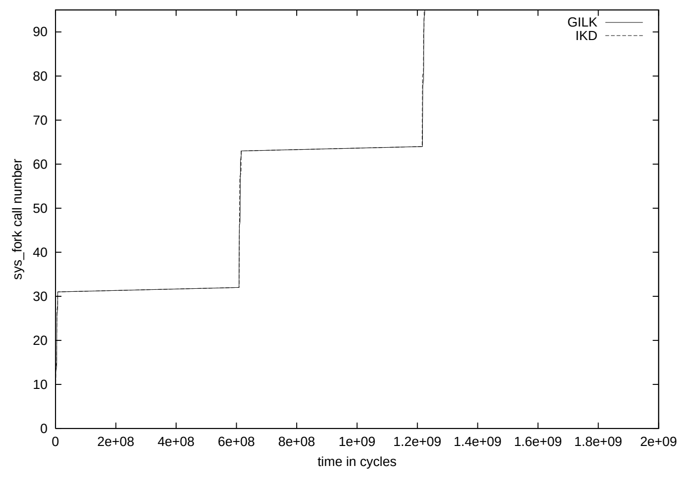


Fig. 8: Illustrating a correspondence between the results of IKD and GILK as the two lines overlap almost perfectly

was applied to do_fork. In the active phase, the simple instrument was added to every basic block of do_fork() and the execution time was recorded as before. The Pulsar program was used to drive sys_fork() and the results can be seen in Figure 9. Again, the machine used was a Pentium 120Mhz running Linux 2.0.36.

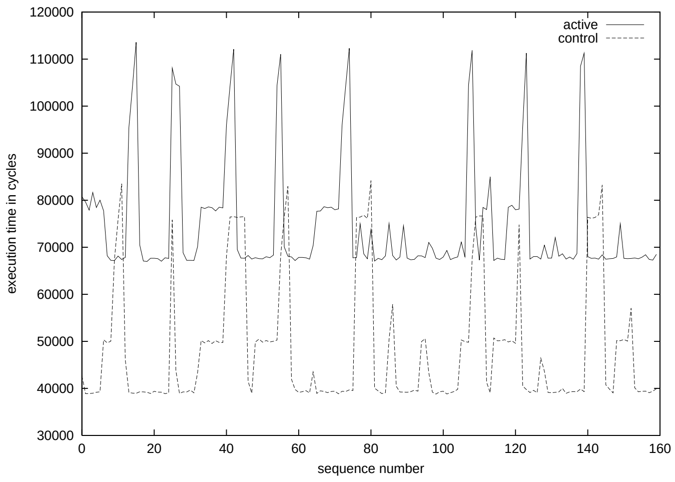


Fig. 9: Illustrating the performance overhead imposed by GILK for a particular experiment

The graph shows a distinct vertical skew between the two sample sets. Measuring between the baselines of each set, the difference is around the 30,000 cycle mark, which is slightly less than the total execution time of the function. Given that there were 94 basic blocks, we can reason that the overhead per instrument is around 300 cycles. This is reasonable for many purposes and could be improved in a more refined prototype.

To see how these cycles are spent we must examine what the simple instrument does. Firstly, it must save and restore the state of the processor which is achieved via the `pushal, pushfl, popal` and `popfl` instructions. Next, the instrument identifier and a timestamp (obtained via the `rdtsc` instruction) are placed on the stack and a call made to the instrument function. This then logs the timestamp with the device driver and returns. For a pentium system, the state saving instructions only account for seventeen cycles and the cost of the function call will be similar. Therefore, we conclude that the majority of the overhead is spent in the device driver, adding samples to the buffer. This suggests that the benefits of live register analysis would be negligible without dramatic improvements to the sample logging process.

4.3 Pipe Blocking

This experiment provides a simple case study to show GILK being used to understand kernel and process behaviour. The idea behind it was this: suppose we have a series of UNIX commands concatenated with the "pipe" operator and we wish to determine which of the commands is the bottleneck. One way of using GILK to determine this is by instrumenting the kernel symbol `pipe_write`. Part of the code for this symbol is:

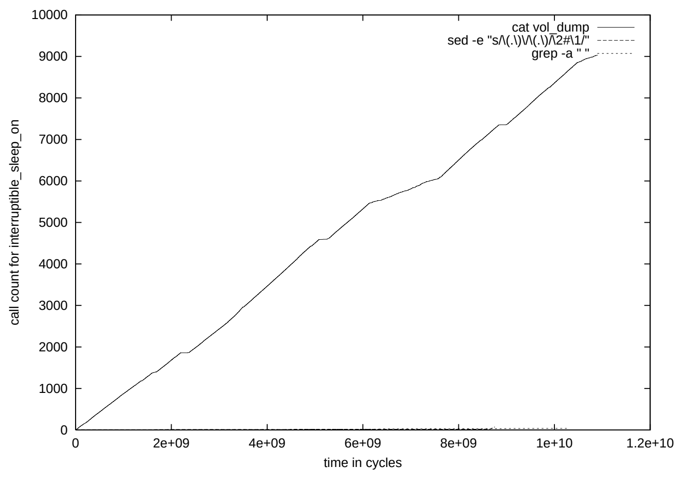


Fig. 10: Showing the results from the pipeline experiment. Firstly, note that the data for "sed" and "grep" form an almost straight line across the bottom. The graph tells us that the "cat" process is making a large number of calls to interruptible_sleep_on() in a periodic fashion, whilst the other processes are making view. This suggests that "sed" is the bottleneck for this pipeline.

```
while ((PIPE_FREE(*inode) < free) || PIPE_LOCK(*inode)) {
        ...
        interruptible_sleep_on(&PIPE_WAIT(*inode));
}
```

The function `interruptible_sleep_on()` puts the process to sleep, pending a wake up call from the pipe reader. So, the above can be simplified to saying that the process is put to sleep when there isn't enough space in the buffer *or* the pipe is locked by a reader. Therefore, it is reasonable to assume that a process in a pipeline will make a lot of calls to `interruptible_sleep_on()` if it is producing data faster that it can be consumed.

To measure this, GILK was used to place a pre-hook instrument on the basic block which makes the call to `interruptible_sleep_on()`. The instrument recorded the Process ID and a timestamp. A large file, called "vol_dump" was created with random data and the following pipeline used:

```
% cat vol_dump | sed -e "s/\(.\)\/\(.\)/\2#\1/" | grep -a " " | sort
```

The results can be seen in Figure 10. They indicate that the "cat" process is making a large number of calls to `interruptible_sleep_on()` whilst the others are making relatively little. This means that "cat" is producing data faster than it can be consumed and this is causing it to block. The suspicion is, therefore, that "sed" is the bottleneck for this pipeline.

If this was the case then we would expect that processes after it would be blocking on their read operations. To confirm this a second experiment

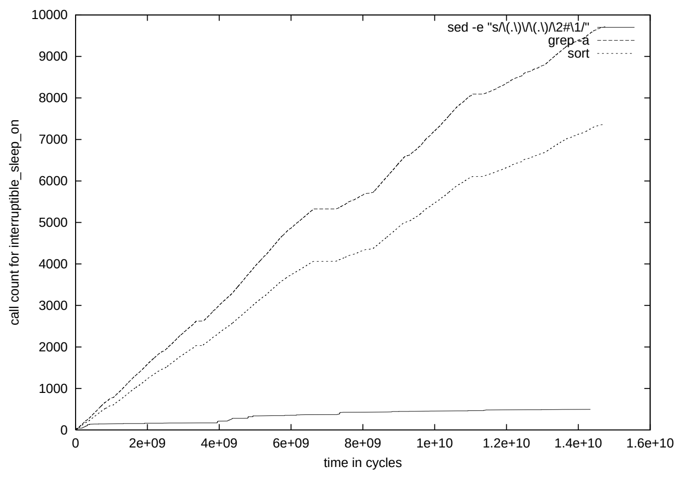


Fig. 11: Showing the results from the second pipeline experiment. This time the number of times pipe_read is blocking is being measured. What the graph shows us is that the processes after "sed" in the pipeline are blocking a lot but sed is not. This confirms the suspicion that "sed" is the bottleneck

was performed in which the pipe read operation was monitored for calls to `interruptible_sleep_on()`. The results from this are shown in Figure 11 and they show that all processes in the pipeline after "sed" are blocking whilst waiting for data to be produced. Hence, the conclusion that "sed" is the bottleneck seems reasonable.

4.4 Network Traffic Analysis

The experiments outlined in this section form part of ongoing research into self-similarity at INSTITUTE. This particular experiment used GILK to investigate the properties of artificial network traffic. For this purpose a simple multi-threaded JAVA server was constructed that transferred data across the network to a number of clients.

The experiment requires that the arrival times of network packets be recorded, so that the *inter-arrival* times can be computed. The unix utility `tcpdump` was initially used for this purpose, but it was later discovered that inter-arrival times of zero were being measured. Clearly, this is a mistake and it was unclear whether the generated power spectra were affected. Thus, GILK was deployed to confirm that inter-arrival times of zero were not real and to ascertain how this feature of `tcpdump` had affected the original data.

The flexibility that GILK provides was key to the success here. It was used to instrument functions within the Linux TCP/IP stack as well as the ethernet driver. This verified that `tcpdump` was producing some erroneous results and, through comparison of power spectra generated by GILK and tcpdump, it was

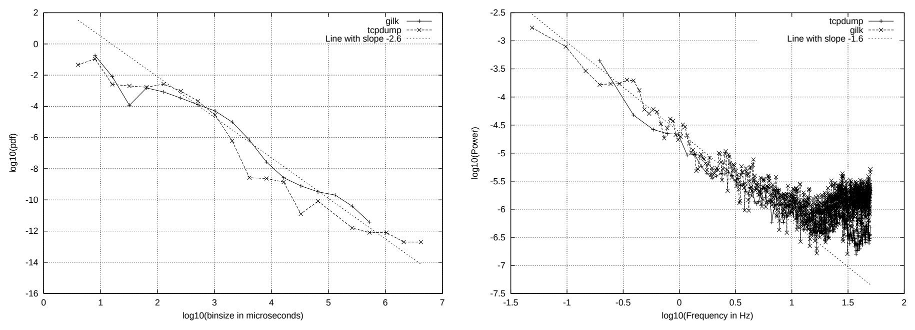


Fig. 12: These plots show a comparison of the histograms (left) and power spectra based on data from `GILK` and `tcpdump`. The graph on the right indicates the presence of a power law. This is because a reasonably straight line is present in the first half of the plot. The plots are similar enough to provide a good verification of data produced by GILK. We would not have expected them to be identical. For this experiment sixty clients were connected to the server.

concluded that there was negligible distortion on the original data. Figure 12 illustrates the comparison.

The significance of the power spectra and self-similarity in general are beyond the scope of this paper and the reader is referred to [12, 7] for more information.

5 Conclusion

The GILK tool employs some state-of-the-art technology to provide a useful instrumentation tool. It is an example implementation of runtime code splicing for a variable length instruction set, which has not been done before. Experimental evidence supports it as an accurate and reasonably low overhead way of performing instrumentation.

There remains, however, some scope for improvement. Particularly, the sample logging process appears expensive. Live register analysis could also be used to make saving the machine state when executing an instrument unnecessary. Also, the need to implement instruments as kernel modules adds to the overhead by requiring an extra function call. This could be prevented by employing a more sophisticated dynamic loader as part of the device driver. The custom disassembler could be reworked to allow easy updating for new instruction set extensions and, finally, the client interface could be extended to provide more default instrumentation strategies and easier navigation through the kernel symbols.

The source code for the tool has been placed under the GNU General Public License and is available for download from `http://www.anonymous.com/~anon/`

GILK. We are hoping to continue this work in the near future and provide a more advanced and mature prototype.

References

1. Ikd: Assorted tools for debugging the linux kernel, ftp://e-mind.com/pub/linux/ikd/.
2. Kernprof: A set of facilities for profiling the linux kernel, http://oss.sgi.com/projects/kernprof/.
3. Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390, November 1997.
4. Michael Beck et al. *Linux kernel internals*. Addison-Wesley, Reading, MA, USA, second edition, 1998. Includes CD-ROM. Translation of the German edition *Linux-Kernel-Programmierung*.
5. Michael Fröhlich and Mattias Werner. The daVinci graph visualization tool, http://www.informatik.uni-bremen.de/davinci/.
6. Jeffrey K. Hollingsworth, Barton P. Miller, J. R. Goncalves Marcelo, Oscar Naim, Zhichen Xu, and Ling Zheng. Mdl: A language and compiler for dynamic program instrumentation. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques (PACT '97)*, pages 201–212, San Francisco, California, November 10–14, 1997. IEEE Computer Society Press.
7. H. J. Jensen. Self-organised criticality, CUP, 1998.
8. James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Software - Practice and Experience*, 24(2):197–218, February 1994.
9. James R. Larus and Eric Schnarr. EEL: machine-independent executable editing. *ACM SIGPLAN Notices*, 30(6):291–300, June 1995.
10. Han Bok Lee and Benjamin G. Zorn. BIT: A tool for instrumenting Java bytecodes. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (ITS-97)*, pages 73–82, Berkeley, December 8–11 1997. USENIX Association.
11. B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn performance tools. *IEEE Computer*, 28, november 1995.
12. C. Tang P.Bak and K. Wiesenfeld. Self organised criticality: an explanation of 1/f noise. *Physical Review Letters*, 59:381, 1987.
13. Amitabh Srivastava and Alan Eustace. Atom: A system for building customized program analysis tools. *ACM SIGPLAN Notices*, 29(6):196–205, June 1994.
14. Ariel Tamches. *Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels*. PhD thesis, University of Wisconsin, 2001.
15. Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Operating Systems Design and Implementation*, pages 117–130, 1999.
16. Ariel Tamches and Barton P. Miller. Using dynamic kernel instrumentation for kernel and application tuning. *The International Journal of High Performance Computing Applications*, 13(3):263–276, Fall 1999.