GILK: A dynamic instrumentation tool for the Linux Kernel

David J. Pearce, Paul H.J. Kelly, Tony Field and Uli Harder

Imperial College of Science, Technology and Medicine
Exhibition Road, London, UK

Abstract. *This paper describes a dynamic instrumentation tool for the Linux Kernel which allows a stock Linux kernel to be modified while in execution, with instruments implemented as kernel modules. The Intel x86 architecture poses a particular problem, due to variable length instructions, which this paper addresses for the first time. 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 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 interface, the user may choose how and where to instrument, when to begin and end individual instruments and what to do with the information produced. 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.

2 Related Work

Much of the foundation for this project has been laid by Tamches, *et al.* [1–3] with the *KernInst* dynamic instrumentation tool. This works on the Solaris kernel and UltraSparc architecture and its techniques are applicable to a fixed length instruction set and multi-threaded kernel.

Binary rewriters such as QP/QPT [4], EEL [5], BIT [6] and ATOM [7] introduce instrumentation code by modifying the executable statically.

This is, arguably, a safer approach than runtime modification, but is more cumbersome. It is our belief that GILK provides a more practical solution, as its *dynamic* approach is more suited to the exploratory nature of performance monitoring and debugging.

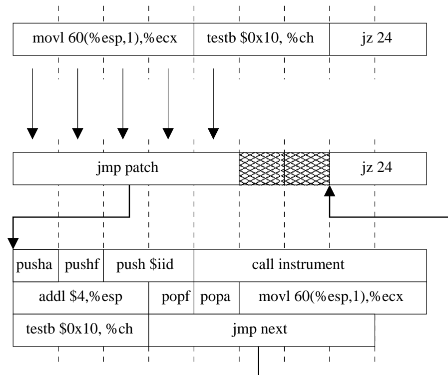

Fig. 1: *Illustrating the process of placing a code splice. In the new sequence two bytes are now redundant, as they are no longer in the flow of control. The two overwritten instructions are relocated to the patch.*

3 GILK Overview

The GILK tool consists of two components: a device driver (called ILK) and a client. The client does the bulk of the work, with the device driver providing access to kernel space. The client begins with a scanning phase to establish the set of valid instrumentation points. The user then specifies what instrumentation should take place, which amounts to selecting instruments, choosing points and specifying start and finish times. The tool supports staggered launching and termination of instruments, which provides greater flexibility.

There are two instrument points associated with each basic block of a kernel function: the pre- and post-hook. A pre-hook instrument gets executed before the first instruction of the block, while a post-hook instrument is executed after the last *non-branching* statement.

Each of the instruments is assigned a unique identifier (iid) which is logged, along with any additional data, to a buffer in the kernel, which the client periodically flushes. Eventually, they are written to disk. The 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 idea behind code splicing is to write a branch instruction or *splice* at the instrument point. Clearly, this will overwrite instructions at that point and, therefore, those affected are first relocated into a *code patch*. The splice targets this code patch, which must also save and restore the machine state and call the instrument function. This is illustrated in Figure 1.

Under the Intel x86 architecture, the splice used is 5 bytes long. However, an instruction may be a single byte in length, making it possible for the splice to overwrite more than one instruction. If an overwritten instruction (other than the first) is the target of a branch, control could be passed into the middle of the splice!

It is for this reason that GILK must generate the Control Flow Graph for each kernel symbol. With this knowledge the above problem can be reduced to saying that it is unsafe to straddle the splice across a basic block boundary.

There is 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 the splice. 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.*
- *Processes running in kernel space may be interrupted by hardware interrupts.*
- *An interrupt handler cannot be interrupted by a process running in kernel space.*

These three points, taken together, allow us to overcome this second problem. Firstly, the ability to block interrupts means the device driver can write the splice without fear of interruption. 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 [8].

3.2 Local Bouncing

It is sometimes the case that a basic block is less than five bytes in length. This means we cannot always place a splice without straddling a block boundary. However, the Intel x86 architecture also supports a two byte branch instruction with limited range. In general, this is not enough to reach the code patch directly.

Therefore, GILK attempts to place a normal splice within this range. This is termed *bouncing*. The problem, then, is where to position these bounces. Luckily, it is often the case that space is made available by splices for other instruments. To understand this better, consider again Figure 1. If the second overwritten instruction was three or more bytes longer then there would be (at least) five redundant bytes available for use as a bounce. If no other instruments are active, or there are simply not enough redundant bytes, 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.

3.3 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 instrument identifier and possibly other arguments depending upon which code patch template was used. An example function is:

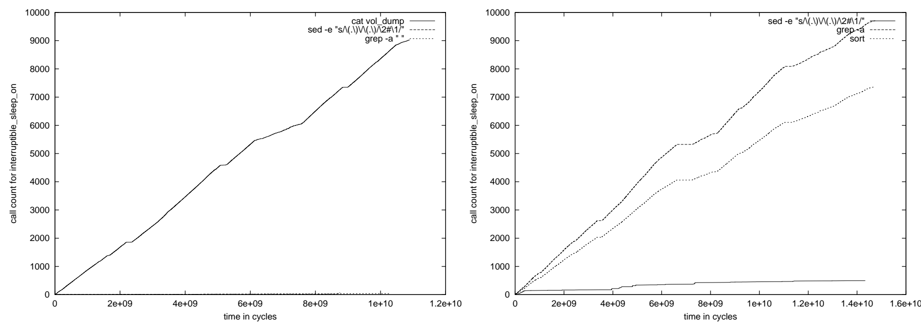

Fig. 2:

```
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. Being a kernel module, it has access to all the structures of the kernel which it can report on. Also, as a 'C' function it could easily be more sophisticated.

4 Experimental Results

4.1 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:

```
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
```

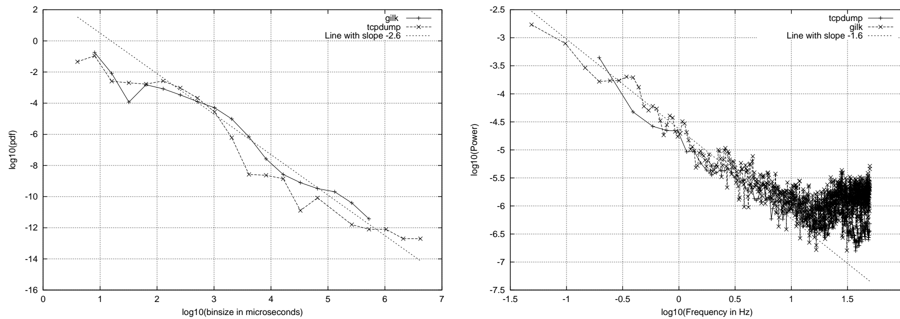

Fig. 3: These plots show a comparison of the histograms (left) and power spectra based on data from `GILK` and `tcpdump`.

The results can be seen in the left graph of Figure 2. 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 processes after it to be blocking on their read operations. To confirm this a second experiment was performed in which the pipe read operation was monitored for calls to `interruptible_sleep_on()`. The results from this are shown in right graph of Figure 2 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.

4.2 Network Traffic Analysis

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

The experiment requires *inter-arrival* times for packets to be measured. The utility `tcpdump` was initially used for this, but it occasionally reported inter-arrival times of zero. Clearly, this is a mistake and it was unclear whether the generated power spectra was being affected.

Thus, GILK was deployed to confirm that inter-arrival times of zero were not real and ascertain if they were affecting the original data. It was used to instrument functions within the Linux TCP/IP stack as well as the Ethernet driver. The measurements taken confirmed that interarrival times were always positive and it was concluded that there was negligible difference between the power spectra generated with `tcpdump`. Figure 3 illustrates the comparison.

The significance of the power spectra and self-similarity are beyond the scope of this paper and the reader is referred to [9, 10] for more information.

5 Conclusion

GILK provides a useful instrumentation tool and provides an example implementation of runtime code splicing for a variable length architecture, which has not been done before. Experimental evidence shows that 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 machine state saving less expensive. 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. 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 instrumentation strategies and easier navigation through the kernel.

The source code for the tool has been placed under the GNU General Public License and is available for download, along with an extended version of this paper [11].

6 Acknowledgements

Uli Harder is supported by an EPSRC grant (QUAINT). David Pearce is supported by an EPSRC grant.

References

1. 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.
2. 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.
3. Ariel Tamches. *Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels.* PhD thesis, University of Wisconsin, 2001.
4. James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Software - Practice and Experience*, 24(2):197–218, February 1994.
5. James R. Larus and Eric Schnarr. EEL: machine-independent executable editing. *ACM SIGPLAN Notices*, 30(6):291–300, June 1995.
6. 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.
7. Amitabh Srivastava and Alan Eustace. Atom: A system for building customized program analysis tools. *ACM SIGPLAN Notices*, 29(6):196–205, June 1994.
8. Michael Beck et al. *Linux kernel internals.* Addison-Wesley, Reading, MA, USA, second edition, 1998.
9. C. Tang P.Bak and K. Wiesenfeld. Self organised criticality: an explanation of 1/f noise. *Physical Review Letters*, 59:381, 1987.
10. H. J. Jensen. Self-organised criticality, CUP, 1998.
11. Gilk: A dynamic instrumentation tool for the linux kernel, `http://www.doc.ic.ac.uk/~djp1/gilk.html`.