Survey of Dynamic Instrumentation of Operating Systems

Harald Röck

Department of Computer Sciences
University of Salzburg, Austria
hroeck@cs.uni-salzburg.at

July 13, 2007

1 Introduction

Operating systems and modern software systems get more complex and more difficult to understand every day. Instrumentation of a software system is *a technique for inserting extra code into an application to observer its behavior*[6]. It provides insights to a running system, and helps to understand even complex systems. The information provided by an instrumentation tool can be used do debug a running system, to improve the performance of a system by identifying bottlenecks, or simply help to understand the control flow of an application or system. In [6], the authors explain how they used their instrumentation tool to identify a stock ticker applet that was responsible for poor performance on a multi user system with more the 170 users. Just by killing the applet the system idle was increased by 15 percent.

Instrumentation can be divided in two major categories: static instrumentation and dynamic instrumentation. Static instrumentation refers to instrumentation techniques that provide information by inserting instrumentation code before the program runs. The additional code could be inserted by the developer in the source code of the program, by the compiler, or by linker. The instrumentation code is always present in the running program, and cannot be changed without stopping or even recompiling the program. The instrumentation statements, however, are usually guarded by a flag to dynamically enable or disable the instrumentation.

Dynamic instrumentation refers to techniques that insert instrumentation code into a running program. The instrumentation code is only present when needed, and can be changed without the need to stop or recompile the program. Dynamic instrumentation tools change a running system by either injecting *probes* or by dynamically *rewriting* the code. Probe based instrumentation tools work similar to a debug. They overwrite a single instruction in the program with a break instruction that transfers control to the instrumentation tool. The instrumentation tool examines the state of the instrumented program, outputs the requested information, and emulates the overwritten instruction before returning control to the instrumented program. Dynamic rewriting, on the other hand, takes a block of code and inserts instrumentation code while rewriting it. The resulting instrumented code is stored on a different location in memory and all references to the original code are updated to point to the new instrumented code. The original code is left unchanged at its old location.

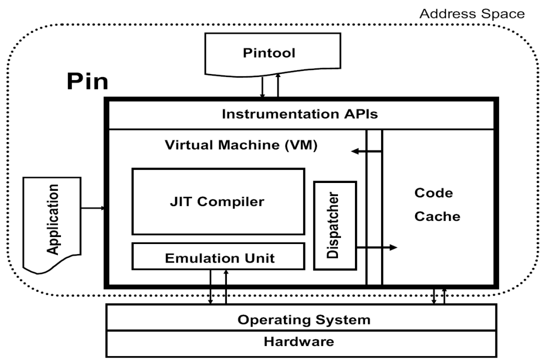

Figure 1: Pin's Software Architecture

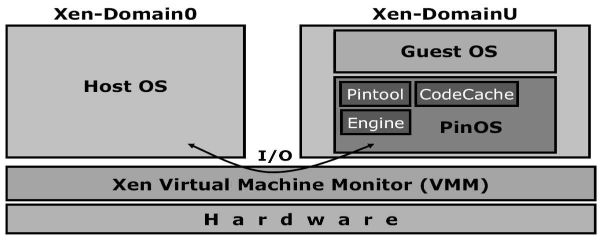

Figure 2: PinOS Architecture

The rewriting is done by a just in time (JIT) compiler, that attempts to optimize the new code using well known optimization algorithm.

Any instrumentation of a system introduces some additional computation and needs additional resources, which potentially distort the results. This is commonly confused with the uncertainty principle [4, 12]. However, the impact of observing a running process is defined as observer effect [11].

2 Pin

Pin [6] is an instrumentation system introduced by Intel. Is is designed for user applications, uses binary rewriting, and provides a rich API to implement instrumentation tools, called Pintools, in C/C++. The pin API and user model is based on ATOM [10] and it is architecture independent whenever possible. As a result it is possible to write portable instrumentation tools. A user can insert instrumentation code at arbitrary locations, and it is not required to manually inline instructions or to do state saving and restoring. By using a just-in-time (JIT) compiler to insert and optimize code, Pin achieves efficient instrumentation. The JIT compiler performs different optimizations to produce efficient code. It supports register reallocation, inlining, liveness analysis, and instruction scheduling.

Figure 1 depicts Pin's run-time software architecture. Pin consists of a virtual machine (VM), a code cache, and an instrumentation API that is used by the Pintools. The VM is further divided into a JIT compiler, an emulation unit and a dispatcher. The JIT compiler is responsible for binary rewriting of block of code. It compiles from one ISA directly to the same ISA without using any intermediate representation. The output of the JIT is stored in the code cache. The dispatcher launches the code produced by the JIT compiler in the code cache. It coordinates the interleaving of the JIT compiler and the actual application. The emulator is necessary to run code that cannot be executed directly. As an example of code that runs in the emulator are system calls, which require special handling from the VM. To obtain control of an application Pin uses the Unix Ptrace API, that is usually used by standard debuggers and tracing tools like strace. Using Ptrace, Pin is even able to attach to a currently running process.

2.1 PinOS

PinOS [1] is an extension to Pin that supports instrumentation across user and kernel code. It extends the Pin API to write Pintools that instrument kernel code in addition to traditional Pin, which was limited to user space code. PinOS is built upon Xen with Intel VT technology and uses virtualization techniques to achieve whole-system instrumentation. Figure 2 depicts the architecture of PinOS running with one guest OS. PinOS obtains system services from the host OS, but runs underneath the guest OS in the guest domain. Its internal architecture is the same as the original Pin.

3 DTrace

DTrace [2] was developed by Sun Microcsystems for dynamic instrumentation of production systems. It is a probe based tool for Solaris, which is able to instrument both user-level and kernel-level software in a ”unified and absolute safe fashion” [2]. The DTrace framework separates instrumentation *providers* from the core framework. When loaded into the kernel, a provider determines the potential instrumentation points and calls back into the core framework to create a probe. This step points out a potential for instrumentation to the DTrace framework, but does not actually instrument the system. A *consumer* can enable an advertised probe, and the DTrace framework calls the provider to activate the probe. Multiple consumers can use the same probe, without the provider being aware of it, since the DTrace framework handles the multiplexing.

A user implements its instrumentation code in a C-like high-level control language, called D. D programs are compiled into the ”D intermediate format” (DIF) instruction set that is emulated within the Solaris kernel. Since the DIF emulator assures safety by validating string and variable references when DIF code is loaded into the kernel, DTrace is considered absolutely safe. In addition, the virtual machine in the kernel checks for run-time errors, like division by zero, that cannot be detected statically.

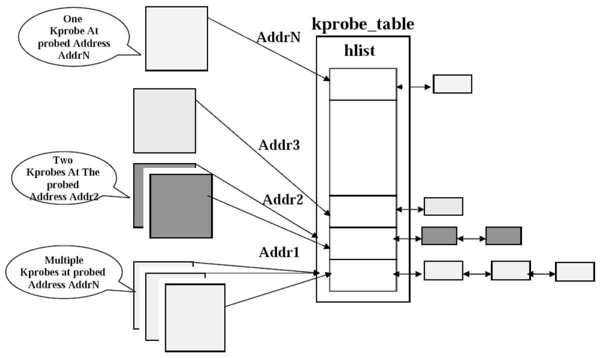
One Kprobe At probed Address AddrN
Two Kprobes At The probed Address Addr2
Multiple Kprobes at probed Address AddrN
AddrN
Addr3
Addr2
Addr1
kprobe_table
hlist

Figure 3: KProbes Internals

4 KProbes

KProbes [5] is the build-in dynamic instrumentation framework of the Linux kernel. The development of KProbes is based on the DProbes [7] patch. Kprobes provides an in-kernel interface for instrumenting the Linux kernel. A user has to write a loadable kernel module, that inserts probes when loaded. The user module has to ensure to remove the inserted installed probes when the module is unloaded. As the name suggest, Kprobes is probe based and overwrites instructions with a break instruction that result in a trap. Figure 3 depicts the internal data structures used by KProbes. It uses a global hash table that is indexed by code addresses. Each address can be associated with one or more Kprobe handler structures. If a trap is triggered, control is transfered to the Kprobes mechanism, which looks for probe handlers in the global hash table. Each probe handler consists of a pre-handler, a post-handler, and a fault-handler. The pre-handler is executed before the probed instruction, the post-handler executes after the probed instruction, and the fault-handler is used to handle any faults that occur during execution of the pre-, or post-handler, or when the probed instruction executes. The handlers can be used to dump the reg-

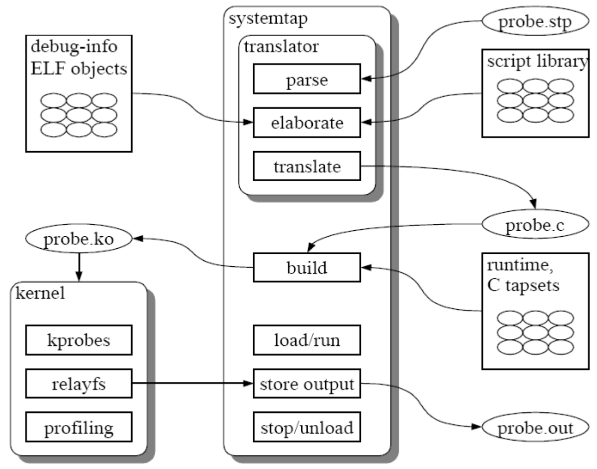

Figure 4: Systemtap Processing steps

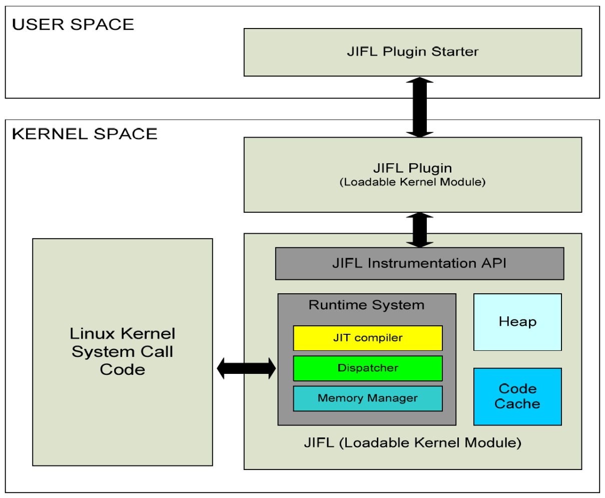

Figure 5: JIFL's Software Architecture

ister contents before, after, and when a fault occurs, respectively.

4.1 SystemTap

SystemTap [9, 3] on Linux corresponds to DTrace on Solaris. Similar to DTrace, SystemTap provides a high-level control language to instrument a running Linux system. It uses Kprobes as underlying mechanism to insert probe points into the kernel.

The processing steps of systemtap is outlined in Figure 4. The systemtap translator takes a user written probe script, the script library and the a image that contains the debug info of the kernel to instrument. The translator input is a C like script language that resembles "D" of Dtrace. A script provides an association of handler routines with probe points, which are abstract names to identify a particular place in the kernel or an event that could occur at any time. The systemtap translator parses the user script, elaborates it with the script library and produces a C source file that is compiled and linked with the runtime framework into a stand-alone loadable kernel module. Systemtap runs the probes by loading the kernel module. When a probe is running, systemtap provides different mechanisms to extract the profiling or logging information of a probe from the kernel. As default it uses relayfs to transport blocks of information from the kernel- to the user-space. A user space daemon collects the information and stores it in a temporary file, that can be examined and analyzed by the user.

5 JIFL

JIFL [8], short for JIT Instrumentation Framework for Linux, is the first framework that uses binary rewriting to instrument OS kernels. The design of JIFL is very close to Pin. Figure 5 depicts JIFL's software architecture. JIFL is a loadable kernel module that consists of a runtime system, a private heap, the code cache and the instrumentation API. The runtime system contains the JIT compiler that copies the machine code into the code cache, while instrumenting it with the user supplied code in a JIFL plugin.

The JIT compiler works on basic block level and supports different optimization techniques. When inserting instrumentation code it performs register and eflags liveness analysis to reduce the number of register savings and restoring operations. Additionally, the JIT attempts to inline the instrumentation code whenever possible.

The dispatcher locates the code blocks in the code cache and redirects execution to it. If the requested code block is not in the code cache the dispatcher starts the JIT compiler, that translate the respective code block and inserts it into the code cache. In addition, the JIFL system needs its own memory manager and heap while doing JIT compilation, because the Linux memory allocator is not reentrant, and JIFL could operate on behalf of a thread that is currently executing a memory allocation request.

A limitation of JIFL is that it is only possible to instrument code that is reached via a system call. The entry point to the JIFL runtime system is the system call table. Therefore, it is not possible to use this system to instrument kernel threads or interrupt handlers. Another implication of the system call table rewriting approach is performance: JIFL introduces a small constant overhead of about 2%. However, the JIFL outperforms the probe based Kprobes mechanism by order of magnitudes in both micro and macro benchmarks, since it uses binary rewriting instead of introducing traps.

6 Conclusion

Modern software systems and especially modern operating systems become more complex and it is increasingly difficult to understand their inner workings. Instrumentation is a well established technique to obtain insight into a running system. However, dynamic instrumentation of operating systems was until recently not very common. The available tools were not easy to use in production systems or introduced a high overhead. Dtrace [6] was ground braking in this field. It provided a good abstraction, an easy to use interface, tool support to instrument a whole system. Systemtap provides basically the same functionality as Dtrace, and runs on top of Kprobes. Both are probe based and overwrite a single instruction with a break instruction similar to a debugger. Recently, JIFL and PinOS were presented. These are tools that use dynamic binary rewriting to replace whole code sections with instrumented code. The resulting code is more efficient, because there is no break instruction necessary, and it is possible to run well known compiler optimization techniques, like inlining and register analysis, to speed up the execution of the instrumented code.

References

[1] BUNGALE, P. P., AND LUK, C.-K. Pinos: a programmable framework for whole-system dynamic instrumentation. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments* (New York, NY, USA, 2007), ACM Press, pp. 137–147.

[2] CANTRILL, B. M., SHAPIRO, M. W., AND LEVENTHAL, A. H. Dynamic instrumentation of production systems. In *ATEC'04: Proceedings of the USENIX Annual Technical Conference 2004 on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2004), USENIX Association, pp. 2–2.

[3] EIGLER, F., PRASAD, V., COHEN, W., NGUYEN, H., HUNT, M., KENISTON, J., AND CHEN, B. Architecture of systemtap: a Linux trace/probe tool. http://sourceware.org/systemtap/archpaper.pdf.

[4] HEISENBERG, W. Über den anschaulichen inhalt der quantentheoretischen kinematik und mechanik. *Zeitschrift für Physik 43* (1927), 172–198.

[5] LINUX TECHNOLOGY CENTER. Kprobes: Kernel probes. http://sourceware.org/systemtap/kprobes.

[6] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2005), ACM Press, pp. 190–200.

[7] MOORE, R. J. A universal dynamic trace for linux and other operating systems. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2001), USENIX Association, pp. 297–308.

[8] OLSZEWSKI, M., MIERLE, K., CZAJKOWSKI, A., AND BROWN, A. D. Jit instrumentation: a novel approach to dynamically instrument operating systems. In *EuroSys '07: Proceedings of the 2007 conference on EuroSys* (New York, NY, USA, 2007), ACM Press, pp. 3–16.

[9] RED HAT, IBM, INTEL, AND HITACHI. Systemtap. http://sourceware.org/systemtap/.

[10] SRIVASTAVA, A., AND EUSTACE, A. Atom: a system for building customized program analysis tools. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation* (New York, NY, USA, 1994), ACM Press, pp. 196–205.

[11] WIKIPEDIA. Observer effect. http://en.wikipedia.org/wiki/Observer_effect.

[12] WIKIPEDIA. Uncertainty principle. http://en.wikipedia.org/wiki/Uncertainty_principle.