iProbe: A Lightweight User-Level Dynamic Instrumentation Tool

Nipun Arora, Hui Zhang, Junghwan Rhee, Kenji Yoshihira, Guofei Jiang
NEC Laboratories America
{nipun,huizhang,rhee,kenji,gfj}@nec-labs.com

***Abstract*—We introduce a new hybrid instrumentation tool for dynamic application instrumentation called *iProbe*[1], which is flexible and has low overhead. iProbe takes a novel 2-stage design, and offloads much of the dynamic instrumentation complexity to an offline compilation stage. It leverages standard compiler flags to introduce "place-holders" for hooks in the program executable. Then it utilizes an efficient user-space "HotPatching" mechanism which modifies the functions to be traced and enables execution of instrumented code in a safe and secure manner. In its evaluation on a micro-benchmark and SPEC CPU2006 benchmark applications, the iProbe prototype achieved the instrumentation overhead an order of magnitude lower than existing state-of-the-art dynamic instrumentation tools like SystemTap and DynInst.**

***Index Terms*—monitoring, tracing, hotpatching, production systems, low-overhead**

I. Introduction

Over the years researchers have proposed many tools to assist in application performance analytics [1], [2], [3], [4], [5], [6], [7], [8]. While these techniques provide flexibility, and deep granularity in instrumenting applications, they often trade in considerable complexity in system design, implementation and overhead to profile the application. For example, binary instrumentation tools like Intel's PIN Instrumentation tool [1], DynInst [8] and GNU debugger [2] allow complete blackbox analysis and instrumentation but incur a heavy overhead. Production system tracers such as DTrace[3] and SystemTap[4] are optimized for inserting hooks in kernel function/system calls, and can monitor run-time application behavior over long time periods. However, they have limited instrumentation capabilities for user-space instrumentation, and incur a high overhead due to frequent kernel context-switches and complex trampoline mechanisms. Hence, software developers often utilize program print statements, write their own loggers, or use tools like log4j [9] or log4c [10] to track the execution of their applications. However, while they have low-overhead they are inflexible and can only be turned on/off at compile-time or before starting the execution, thereby greatly reducing their effectiveness.

The main idea in iProbe design is *decoupling the process of run-time instrumentation into offline and online stages*, which avoids several complexities faced by current state-of-the-art mechanisms [3], [4], [8], [1] such as instruction overwriting, complex trampoline mechanisms, and code segment memory allocation, kernel context switches etc. Most existing dynamic instrumentation mechanisms rely on a trampoline based design, and generally have to make several jumps to get to the instrumentation function as they not only do instrumentation but also simulate the instructions that have been overwritten. iProbe on the other hand, can be imagined as a framework which provides a seamless transition from a non-instrumented binary to an instrumented binary. We use a hybrid 2-step mechanism which offloads dynamic instrumentation complexities to an offline development stage, thereby giving us a simpler design with significantly better performance. Our first phase *ColdPatch* prepares "place-holders" in the binary which can be turned on and off using dynamic instrumentation in our *HotPatch* phase. iProbe has instrumentation overhead comparable to print/debug statements, while still providing users the flexibility to choose targets in the execution stage. iProbe showed an order of magnitude performance improvement in comparison to SystemTap[6] and DynInst[8] in terms of tracing overhead and scalability. Looking forward, we deem it a new paradigm in packaging of applications, wherein developers can insert instrumentation ready "place-holders" using our ColdPatch; an iProbe-ready application can then be deployed in the production environment and can easily be monitored when required.

II. Design

A. ColdPatching Phase

ColdPatching is a pre-processing phase which generates the place-holders for hooks to be replaced with the calls for instrumentation. The following are the key steps in coldpatch:

Firstly, iProbe uses compiler techniques to insert instrumentation calls at the beginning and end of each function call (see section IVand II-C for further details)[2]

Secondly, iProbe parses the binary and replaces all instrumentation calls generated using the compiler with a `NOP` instruction. This generates instructions in the binary which does no-operation, hence has a negligible overhead, and acts as an empty space for iProbe to be overwritten at run-time.

Thirdly, iProbe parses the binary and gathers meta-data regarding all the target instrumentation points into a *probe-list*. The probe-list is securely transferred to the run-time interface of iProbe and used to probe the instrumentation points.

[1]Tool Demo: http://www.youtube.com/watch?v=8fO9fvQZ9kQ, iProbe is a proprietary software and is not available publicly

[2]Apart from compiler techniques, placeholders can be added in a variety of ways including binary rewriting or user defined macros etc.

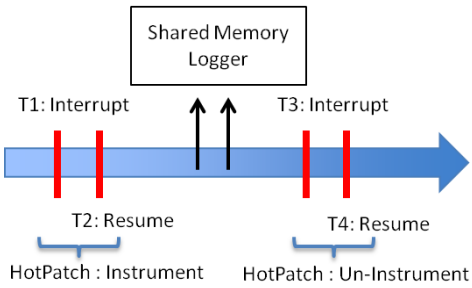

Fig. 1. HotPatching Work-flow.

The parameters of the instrumentation functions added by the compiler, are decided on the basis of the design of the compiler pass. Developers can define their own compiler passes(or choose from pre-defined passes) to customize the addition of placeholders. One of the advantages of this approach is that since target instrumentation is passed using a meta-data list, iProbe can strip away all debug and symbolic information in the binary making it more secure and light-weight, as debug information is not required at run-time.

B. HotPatching Phase

Once the application binary has been statically patched (i.e., ColdPatched), instrumentation can be applied at run-time. Compared to existing trampoline approaches, *iProbe does not overwrite any instructions in the original program, or allocate additional memory* when patching the binaries, and still ensures reliability. In order to have a *low overhead*, and *minimal intrusion* of the binary, iProbe avoids most of the complexities involved in HotPatching such as allocation of extra memory in the code segment or scanning code segments to find instrumentation targets in an offline stage. The process of HotPatching is as follows:

Firstly, iProbe loads the relevant instrumentation functions in a shared library to the code-segment of the target process. This along with allocation of `NOP`s in the ColdPatching phase allows iProbe to avoid allocation of memory for instrumentation in the code segment.

The probe-list generated in the ColdPatching phase is then given to our HotPatch program as a list of target probe points in the executable (iProbe can handle stripped binaries due to previous knowledge of the target instructions). As shown in Figure 1, our HotPatcher attaches itself to the target process and issues an interrupt (time T1). It then performs a safety check (see Section II-D), and subsequently replaces the `NOP` instructions in each target function, with a call to the instrumentation function. This is a key step which enables iProbe to avoid the complexity of traditional trampoline [11], [12] by not overwriting any logical instructions (non-`NOP`) in the original code. Since the place-holders (`NOP` instructions) are already available, iProbe can seamlessly patch these applications without changing the size or the run-time footprint of the process. Once the calls have been added iProbe releases the interrupt and let normal execution proceed (time T2). To disable tracing we reverse the process and convert probe points back to `NOP`s (time T3 and T4).

C. State Transition Flow

Figure 2 demonstrates an example of the operational flow of iProbe when instrumenting the entry and exit of function `func_foo`. The left most figure represents the instructions of a native binary. As an example, it shows three instructions (i.e., push, pop, inc) in the prolog and one instruction (i.e., pop) in the epilog of the function `func_foo`. The next figure shows the layout of this binary when it is compiled with the instrumentation option. As shown in the figure, two function calls, `foo_begin` and `foo_end` are automatically inserted by the compiler at the start and end of the function respectively. iProbe exploits these two newly introduced instructions as the place-holders for HotPatching. The ColdPatching process overwrites two call instructions with `NOP`s. At run-time, the instrumentation of `func_foo` is initiated by HotPatching those instructions with the call instructions to the instrumentation functions: `begin_instrument` and `end_instrument`.

D. Safety Checks for iProbe

In this section we briefly mention some of the safety and reliability issues iProbe deals with:

Firstly, when interrupting and overwriting instructions, iProbe ensures that the program counters of all threads belonging to the target applications do not point to the instruction being replaced. This is done so that we do not have an inconsistent state such that the instruction being read is partly a call instruction and partly NOP (which results in an illegal instruction crash). This check is similar to those from traditional Ptrace [7] driven debuggers [13], [11], [14] and uses `GETREGS()` to inspect registers.

Secondly, iProbe enforces use of *cdecl* [15] (as compared to *std* calls) type calls for instrumentation to ensure stack consistency when passing parameters. These calls ensure that the caller function performs stack cleanup operations. This is important, as iProbe needs to ensure that parameters passed to the instrumentation functions are safely removed after the function is executed.

Thirdly, iProbe also deals with ASLR[16] (address space layout randomization), a security measure which randomizes the load address of executables and shared libraries(ASLR can randomize offsets of probe points in each execution). To deal with this problem, iProbe assumes privileged access to the target system and finds the load addresses of each binary/shared library using the process-id mapping information. It then uses this information to find the base offsets of the binaries and generates correct instruction addresses.

III. Trampoline vs. Hybrid Approach

One of the key reasons for better performance of iProbe in run-time is avoiding multiple jumps enforced in the trampoline mechanism. As shown in Figure 3, to insert a hook for the

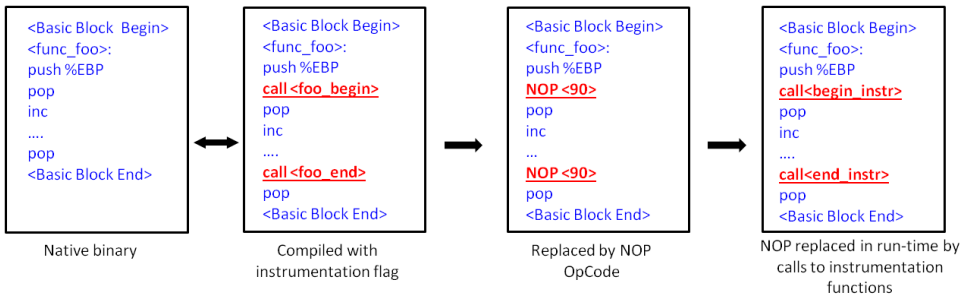

Fig. 2. Native Binary, the State Transition of ColdPatching and HotPatching.

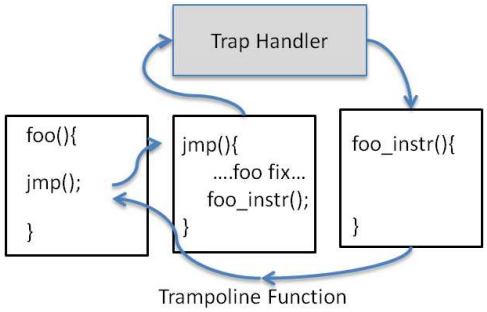

Fig. 3. Traditional Trampoline based Dynamic Instrumentation Mechanisms.

function `foo()`, dynamic instrumentation tools overwrite target probe point instructions with a jump to a small trampoline function (`jmp()`). Note that the overwritten code by `jmp` should be executed somewhere to ensure the correctness of the original program. The trampoline function executes the overwritten instructions (`foo fix`) before executing the actual code to be inserted. Then this trampoline function in turn makes the call to the instrumentation function (`foo_instr`). Each call instruction can potentially lead to branch mispredictions in the code cache and cause high overhead. Additionally tools like DTrace, and SystemTap [3], [4] have the logger in the kernel space, and cause a context switch in the trampoline using interrupt mechanisms. In comparison iProbe has a `NOP` instruction which can be easily overwritten without resulting in any illegal instructions, and since overwriting is not a problem trampoline function is not required. This makes the instrumentation process simple resulting in only a single call instruction at all times. Additionally, iProbe avoids overheads because of memory allocation required in trampoline approaches, and complex guarantees of safety and reliability, which iProbe offloads to an offline stage because of its hybrid design.

IV. Implementation

The design of iProbe is generic, and has been tested on gcc/g++ generated native binary executables in linux.

ColdPatch: We implemented this by compiling binaries using the `-finstrument-functions`[17] flag[3]. This compiler option places function calls to instrumentation functions (`_cyg_profile_func_enter/_exit`) after the entry and before the exit of every function. Subsequently, our ColdPatcher parses the binary to read through all the target binaries, and search and replace the instruction offsets containing the instrumentation calls with NOP instructions (instruction '90'). Symbolic and debug information is read from the target binary using commonly available `objdmp`[18] tools; This information combined with target instruction offsets are used to generate the probe list with the following information:

`<Instr Offset, Entry\Exit Point, Meta-Data>`

The meta-data here is the file, function name, line no. etc.

HotPatching: In the run-time phase, we first use the library interposition technique, `LD_PRELOAD`, to preload the instrumentation functions in the form of a shared library to the execution environment. The HotPatcher then uses a command line interface which interacts with the user and provides the user an option to input the target process and the probe list. Next, iProbe collects the base addresses of each shared library and the binary connected to the target process from `/proc/pid/maps`. The load address and offsets from the probe-list are then used to generate an index of all possible probing points. iProbe then prompts the user for a list of target functions and interrupts the process (we provide the user the meta-data information list of target functions and their respective file information.) We then use `ptrace` functionality to patch the target instructions with calls to our instrumentation functions, and release the process to execute as normal. The instrumentation from each function is registered and logged by an efficient lock-free shared memory logger.

V. Evaluation

A. What is the overhead of an iProbe enabled(ColdPatch) application without instrumentation?

We tested iProbe on 8 SPEC [19] benchmark applications shown in Figure 4. The first column shows the execution of a normal binary compiled without any instrumentation or

[3]Note that this can be done simply by appending this flag to the list of compiler flags (e.g., `CFLAG`, `CCFLAG`, `CXXFLAGS`) and most of cases it works without interfering with user code.

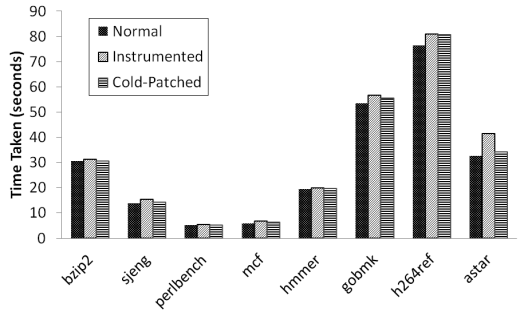

Fig. 4. Overhead of iProbe "ColdPatch Stage" on SPEC CPU 2006 Benchmarks.

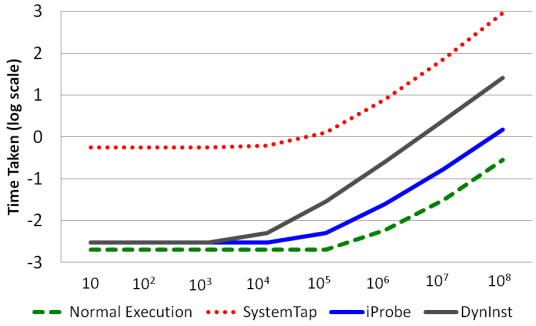

Fig. 5. Overhead and Scalability Comparison of iProbe HotPatching vs. SystemTap vs. DynInst using a Micro-benchmark.

debug flags. The next column shows the execution time of the corresponding binary compiled using the instrumentation flags[4]. Lastly, we show the overhead of a ColdPatched iProbe binary with `NOP` instead of the call instruction. The overhead for a ColdPatched binary was found to be less than five percent for all applications executed, and 0-2 percent for four of the benchmarks. The overhead here is basically because of the `NOP` instructions that are placed in the binary as place-holders for the HotPatching. In most production applications (e.g., apache, mysql) we have observed the overhead to be negligible (less than one percent), with no observable effect in terms of throughput. Additionally, increase in the size of the binary is directly proportional to the number of targets[5], and is usually not a concern in most production servers.

B. How does iProbe compare to existing tools in terms of overhead and scalability?

We compared the scalability and hotpatching overhead of iProbe with UTrace (user-space tracing in SystemTap) [6], and DynInst [8] on a x86_64, dual-core machine with Ubuntu 12.10 kernel using a micro-benchmark[6]. The micro-benchmark is a dummy application with multiple calls to an empty function `foo`. The instrumentation tools of each framework simply increases a global counter for each event triggered (entry and exit of foo). To test the scalability of our the tools, we have increased the number of calls made to `foo` exponentially (increase by multiples of 10).

We found that iProbe scales very well and is able to keep the overhead to less than five times for millions of events (10^8) generated in less than a second (normal execution) for entry as well as exit of the function. While iProbe executed in 1.5 seconds, the overhead observed in SystemTap is around 20 minutes for completion of a sub-second execution, while DynInst takes about 25 seconds. As explained in Section III, tools such as DynInst use a trampoline mechanism, hence have a minimum of 2 call instructions for each instrumentation. Additionally SystemTap uses a context switch to switch to the kernel space over and above the traditional trampoline mechanism, resulting in the higher overhead, and less scalability observed in our results.

VI. Conclusion

Flexibility and performance have been two conflicting goals for the design of dynamic instrumentation tools. iProbe offers a solution to this problem through a two-stage process that offloads much of the complexity involved in run-time instrumentation to an offline stage. We presented in the evaluation that iProbe is significantly faster than existing state-of-the-art tools, and scales well in large application software.

[4]calls inserted by the instrumentation flag do not execute any function and can be safely executed even without cold-patching

[5]In the example in this paper, two call instructions for every function

[6]For most large scale applications the selection of the functions to be probed and their frequency will have a heavy impact on the overhead. This micro-benchmark focuses only on the overhead generated by each of the frameworks, and since it's repeatable it can be easily used for a scalability analysis.

References

[1] C.-K. e. a. Luk, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI '05*, 2005.

[2] R. Stallman, R. Pesch, and S. Shebs, "Debugging with gdb: The gnu source-level debugger for gdb version 5.1. 1," 2002.

[3] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic instrumentation of production systems," in *USENIX'04*, 2004.

[4] V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen, "Locating system problems using dynamic instrumentation," in *Proceedings of the 2005 Ottawa Linux Symposium (OLS)*, 2005.

[5] M. Desnoyers and M. Dagenais, "The lttng tracer: A low impact performance and behavior monitor for gnu/linux," in *OLS (Ottawa Linux Symposium)*, 2006, pp. 209–224.

[6] R. McGrath, "Utrace," *Linux Foundation Collaboration Summit*, 2009.

[7] "Ptrace:linux process trace," http://linux.die.net/man/2/ptrace.

[8] B. Buck and J. K. Hollingsworth, "An api for runtime code patching," *Int. J. High Perform. Comput. Appl.*

[9] S. Gupta, *Logging in Java with the JDK 1.4 Logging API and Apache log4j*. Apress, 2003.

[10] "log4c: Logging for c library," http://log4c.sourceforge.net.

[11] "Livepatch," http://ukai.jp/Software/livepatch/.

[12] S. e. A. Bratus, "Katana: Towards patching as a runtime part of the compiler-linker-loader toolchain," *International Journal of Secure Software Engineering (IJSSE)*, vol. 1, no. 3, pp. 1–17, 2010.

[13] K. Yamato, T. Abe, and M. Corpration, "A runtime code modification method for application programs," in *Proceedings of the Ottawa Linux Symposium*, 2009.

[14] "Pannus: A hot patching tool," http://pannus.sourceforge.net/.

[15] "Calling conventions for different operating systems," http://agner.org/optimize/calling_conventions.pdf.

[16] "Pax aslr documentation," http://pax.grsecurity.net/docs/aslr.txt.

[17] "Gcc options for code generation," http://gcc.gnu.org/onlinedocs/gcc-4.4.6/gcc/Code-Gen-Options.html.

[18] "Object dump tool," http://sourceware.org/binutils/docs/binutils/objdump.html.

[19] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.