iProbe: A Lightweight User-Level Dynamic Instrumentation Framework

Nipun Arora*, Hui Zhang*, Kai Ma†, Junghwan Rhee*
Kenji Yoshihira*, Guofei Jiang*, Xiaorui Wang†
*NEC Laboratories America †Ohio State University
{nipun,huizhang,rhee,kenji,gfj}@nec-labs.com ma.495@osu.edu xwang@ece.osu.edu

***Abstract*—Application tracing in production systems requires dynamic and flexible instrumentation mechanisms with low-overhead. Tracing tools may be required to be started at anytime, and it can take potentially long time periods to collect enough information, but at the same time should not adversely affect service quality. Existing user-space code monitoring solutions are either inflexible developer-driven static instrumentation which require manual effort, or black-box dynamic instrumentation techniques which are flexible but have high overhead.**

To solve this problem, we introduce a new hybrid instrumentation technique for user-space code monitoring called *iProbe*, 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 executables. Then it utilizes an efficient user-space HotPatching mechanism which dynamically instruments the functions to be traced and enables execution of instrumented code in a safe and secure manner.

We implemented iProbe as a dynamic application profiling framework. In its evaluation on micro-benchmarks 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. We also built a hardware event profiling tool based on iProbe, and were able to obtain function-level hardware event breakdown on SPEC CPU2006 applications with controlled performance overhead (e.g., under 5%).

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

I. Introduction

Ideally, a production system tracing tool should have zero-overhead when it is not activated and should have a low overhead when it is activated. In other words, its performance should not adversely effect the usage of the traced application. At the same time, it should be flexible enough so as to meet versatile instrumentation needs at run-time for management tasks such as trouble-shooting or performance analysis.

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, which is unacceptable in production systems. Inherently, these tools have been developed for the development environment, hence are not meant for a production system tracer.

Production system tracers such as DTrace[3] and SystemTap[4] allow for low overhead kernel function tracing. These tools 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.

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. Those manually instrumented probe points can easily be deployed without additional libraries or kernel support, and have a low overhead to run without impacting the application performance noticeably. However, they are inflexible and can only be turned on/off at compile-time or before starting the execution. Besides, usually only a small subset of functions is chosen to avoid larger overheads.

In this paper, we introduce a dynamic instrumentation framework called *iProbe*. iProbe has instrumentation overheads comparable to print/debug statements, while still giving users the flexibility to choose targets in the execution stage. We evaluated iProbe on micro-benchmark and SPEC CPU 2006 benchmarks. iProbe showed an order of magnitude performance improvement in comparison to SystemTap[6] and DynInst[8] in terms of tracing overhead and scalability. Additionally, the instrumented applications incur negligible overhead when iProbe is not activated. We also present a new hardware event profiling tool called *FPerf* developed in the iProbe framework. FPerf leverages iProbe's flexibility and scalability to realize a fine-grained performance event profiling solution with overhead control. In the evaluation, FPerf was able to obtain function-level hardware event breakdown on SPEC CPU2006 applications while controlling performance overhead (e.g., under 5%)

The main idea in iProbe design is *decoupling the process of run-time instrumentation into offline and 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. Additionally, they have frequent context-switches as they use kernel traps to capture instrumentation points, and execute the instrumentation. The performance penalty imposed by these designs are unacceptable in a production environment.

Our design avoids any transitions to the kernel which generally causes higher overheads, and is completely in user space. iProbe can be imagined as a framework which provides a seamless transition from an instrumented binary to a non-instrumented binary. We use a hybrid 2-step mechanism which offloads dynamic instrumentation complexities to an offline development stage, thereby giving us a much better performance. The following are the 2 stages of iProbe:

- **ColdPatch:** We first prepare the target executable by introducing dummy instructions as "place-holders" for hooks during the development stage of the application. This can be done in 3 different ways: Primarily, we can leverage compiler based instrumentation to introduce our "place-holders". Secondly we can allow users to insert macros for calls to instrumentation functions which can be turned on and off at run-time. Lastly we can use static binary rewriter to insert place-holders in the binary without any recompilation. iProbe uses binary parsers to capture all place-holders in the development stage and generates a meta-data file with all possible probe points created in the binary.
- **HotPatch:** We then leverage these place-holders during the execution of the process to safely replace them with calls to our instrumentation functions during run-time. iProbe uses existing tools, ptrace [7], to modify the code segment of a running process, and does safety check to ensure correctness of the executing process. Using this approach in a systematic manner we reduce the overhead of iProbe while at the same time maintaining a relatively simple design.

We propose a new paradigm in development and packaging of applications, wherein developers can insert probe points in an application by using compiler flag options, and applying our ColdPatch. An iProbe-ready application can then be packaged along with the meta-data information and deployed in the production environment. iProbe has negligible effect on the application's performance when instrumentation is not activated, and low overhead when instrumentation is activated. We believe this is an useful feature as it requires minimal developer effort, and allows for low overhead production-stage tracing which can be switched on and off as required. This is desirable in long-running services for both debugging and profiling usages.

The rest of the paper is organized as following. Section II discusses the design of iProbe framework, explaining our ColdPatching, and HotPatching techniques; we also discuss how safety checks are enforced by iProbe to ensure correctness, and some extended options in iProbe for further flexibility. Section III compares traditional trampoline based approaches with our hybrid approach and discusses why we perform, and scale better. Section IV explains the implementation of iProbe, and describes *FPerf* a tool developed using iProbe framework. In section V we evaluate the iProbe prototype. Section VI discusses the related work, and Section VII concludes this paper.

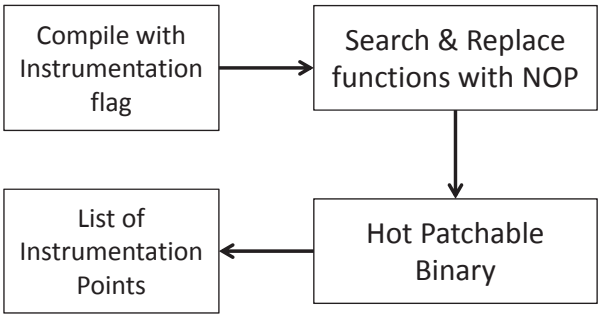


Fig. 1. The Process of ColdPatching.

II. Design

In this section we present the design of iProbe. Additionally, we then explain some safety checks imposed by iProbe that ensure the correctness of our instrumentation scheme. Finally, we discuss extended iProbe modes, static binary rewriting and user written macros, which serve as alternatives to the default compiler-based scheme to insert instrumentation in the pre-processing stage of iProbe.

The first phase of our instrumentation is an offline pre-processing stage to make the binaries ready for runtime instrumentation. We call this phase *ColdPatching*. The second phase is the an online *HotPatching* stage which instruments the monitored program dynamically at runtime without shutting down and restarting the program. Next, we present the details of each phase.

A. ColdPatching Phase

ColdPatching is a pre-processing phase which generates the place-holders for hooks to be replaced with the calls for instrumentation. This operation is performed offline before any execution by statically patching the binary file. This phase is composed of three internal steps that are demonstrated in Figure 1.

- Firstly, iProbe uses compiler techniques to insert instrumentation calls at the beginning and end of each function call. The instrumentation parameters, are decided on the basis of the design of the compiler pass. The current implementation by default passes callsite information and the base stack pointer as they can be used to inspect and form execution traces. Calls to the these instrumentation functions must be *cdecl* calls so that stack correctness can be maintained, this is discussed in further detail in Section II-C.

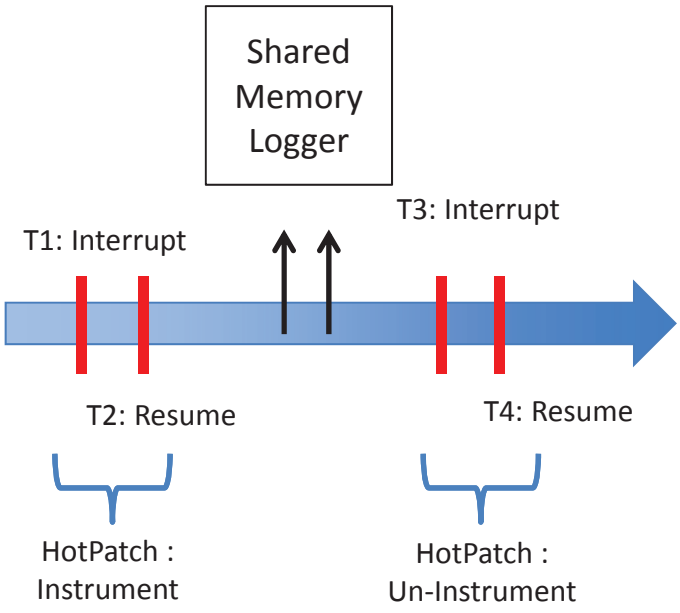


Fig. 3. HotPatching Workflow.

- Secondly, iProbe parses the executable and replaces all instrumentation calls with a `NOP` instruction which is a no-operation or null 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*. Optionally, iProbe can strip away all debug and symbolic information in the binary making it more secure and light-weight. The probe-list is securely transferred to the run-time interface of iProbe and used to probe the instrumentation points. Hence iProbe does not have to rely on debug information at run-time to HotPatch the binaries.

B. HotPatching Phase

Once the application binary has been statically patched (i.e., ColdPatched), instrumentation can be applied at runtime. 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 given to our run-time environment as a list of target probe points in the executable. iProbe can handle stripped binaries due to previous knowledge of the target instructions in the ColdPatching.
- As shown in Figure 3, in our instrumentation stage, our HotPatcher attaches itself to the target process and issues an interrupt (time T1). It then performs a reliability check (see Section II-C), and subsequently replaces the `NOP` instructions in each of the target functions, with a call to our 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 runtime footprint of the process. Once the calls have been added iProbe releases the interrupt and let normal execution proceed (time T2).
- At the un-instrumentation stage the same process is repeated, with the exception that the target functions are again replaced with a `NOP` instruction. The period between time T2 and time T3 is our monitoring period, wherein all events are logged to a user-space shared memory logger.

State Transition Flow: Figure 2 demonstrates the operational flow of iProbe in the example to instrument the entry and exit of the `func_foo` function. 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 runtime, the instrumentation of `func_foo` is initiated by HotPatching those instructions with the call instructions to the instrumentation functions: `begin_instrument` and `end_instrument`. This is illustrated in the right most figure in Figure 2.

Logging Functions and Monitoring Dispatchers : The calls from the target function to the instrumentation function are generally defined in the coldpatch stage by the compiler. However, iProbe also provides monitoring dispatchers which are common instrumentation functions that are shared by target functions. Our default instrumentation passes the call site information, and the function address of the target function as parameters to the dispatchers. Each monitoring event can be differentiated by these dispatchers using a quick hashing mechanism representing the source of each dispatch. This

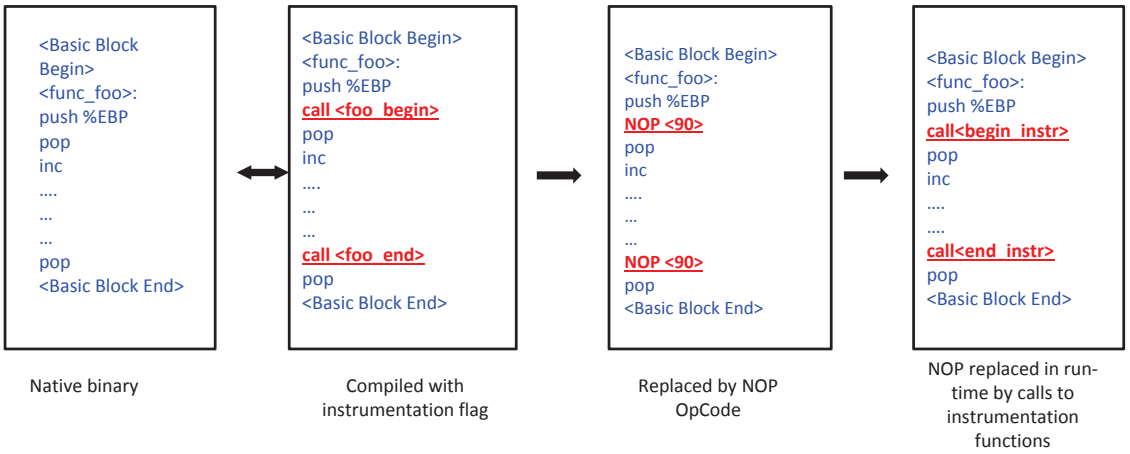


Fig. 2. Native Binary, the State Transition of ColdPatching and HotPatching.

allows iProbe to uniquely define instrumentation for each function at run-time, and identify its call sites.

C. Safety Checks for iProbe

Safety and reliability of the instrumentation technique is a big concern for most run-time instrumentation techniques. One of the key advantages of iProbe is that because of its hybrid design reliability and correctness issues are handled in a better way inherently. In this section we discuss how our HotPatch can achieve such properties in details.

HotPatch check against Illegal instructions: Unlike previous techniques iProbe relies on compiler correctness to ensure safety and reliability in its binary mode. To ensure correctness in our ColdPatching phase, we convert call instructions to instrumentation functions with `NOP` instruction. This does not in any way effect the correctness of the binary, except that instrumentation calls are not made. To ensure run-time correctness, iProbe uses a safety check when it interrupts the application while HotPatching. Our safety check pass ensures that the program counters of all threads belonging to the target applications do not point to the region of code that is being overwritten (i.e. `NOP` instructions are not overwritten while they are being executed. This check is similar to those from traditional Ptrace[7] driven debuggers etc [13], [11], [14]. Here we use the Ptrace `GETREGS()` call to inspect the program counter, and if it is currently pointing to the target `NOP` instructions, we allow the execution to move forward before applying the HotPatch. Unlike existing trampoline oriented mechanisms iProbe has a small `NOP` code segments equal to the length of a single call instruction that it overwrites with instrumentation calls, this means that the check can be performed in a fast and efficient manner. It is also important to have this check for all threads which share the code-segment, hence the checking must be able to access the process memory map information, and interrupt all the relevant threads.

Safe parameter passing to maintain stack consistency: An important aspect for instrumentation is the information passed along to the instrumentation function via the parameter values. Since the instrumentation calls are defined by the compiler driven instrumentation, the mechanism in which the parameters passed are decided based on the calling convention used by the compiler.

Calling conventions can be broadly classified in two types: caller clean-up based, and callee clean-up based. In the former the caller is responsible to pop the parameters passed to function, and hence all parameter related stack operations are performed before and after the call instruction inside the code segment of the caller. In the later however, the callee is responsible to pop the parameters passed to it. Since parameters are generally passed using the stack it is important to remove them properly to mantain stack consistency.

To ensure this iProbe enforces that all calls that are made by the static compiler instrumentation must be *cdecl* calls where the caller performs the cleanup as compared to *std* calls, where the callee performs it[15]. As the stack cleanup is automatically performed, it maintains stack consistency, and there is a negligible impact in performance due to the redundant stack operations. Alternatively for *std* call convention, push instructions could also be converted to `NOP`s and HotPatched at run-time, we do not do so as a design choice.

Address Space Layout Randomization: Another issue that iProbe addresses is ASLR (address space layout randomization), a security measure used in most environments which randomizes the loading address of executables and shared libraries. However, since iProbe assumes the full access to the target system, the load addresses are easily available. HotPatcher uses the process id of the target to find all load addresses of each binary/shared library and uses them as base offsets to generate correct instruction pointer addresses.

D. Extended iProbe Mode

As iProbe ColdPatching requires compiler assistance, it is unable to operate on pre-packaged binary applications. Additionally, compiler flags generally have limited instrumentation flexibility as they generally operate on a programming language abstraction(eg. function calls, loops etc.). To provide

further flexibility, iProbe provides a couple of extended options for ColdPatching of the application

1) Static Binary Rewriting Mode: In this mode we use a static binary rewriter to insert instrumentation in a pre-packaged binary. Once all functions are instrumented, we use a ColdPatching script to capture all call sites to the instrumentation functions and convert them to `NOP` instruction. While this mode allows us to directly operate on binaries, a downside is that our current static binary instrumentation technique also uses mini-trampoline mechanisms. As explained in Section III static binary rewriters use trampoline based mechanisms which induces minimum two jumps. In the ColdPatch phase, we convert calls to the instrumentation function to `NOP`s, however the jmp operations to the trampoline function, and simulation of the overwritten instructions still remain. This approach has a small overhead even when instrumentation is turned off. However, in comparison to pure dynamic instrumentation approach it reduces the time spent in HotPatching. This is especially important if the number of instrumentation targets is high, and the target binary is large, as it will increase the time taken in analyzing the binaries. Additionally, if compiler options cannot be changed for certain sections of the program (plugins/3rd party binaries), iProbe can still be applied using this extended feature.

Our current implementation uses the dyninst [8] and cobi [16] to do static instrumentation. This allows us to provide the user a configuration file and template which can be used to specify the level of instrumentation (e.g., all entry and exit points for instrumentation), or names of specific target functions, and the instrumentation to be applied to them. Subsequently in ColdPatch we generate our meta-data list, and use it to HotPatch and apply instrumentation at run-time.

2) Developer Driven Macros: Compiler assisted instrumentation may not provide complete flexibility (usually works on abstractions, such as enter/exit of functions), hence for further flexibility, iProbe provides the user with a header file with calls to macros which can be used to add probe points in the binary. A call to this macro can be placed as required by the developer. The symbol name of the macro is then used in the ColdPatch stage to capture these macros as probe points, and convert them to `NOP`s. Since the macros are predefined, they can be safely inserted and interpreted by ColdPatcher. The HotPatching mechanism is very much the same, using the probe list generated by ColdPatch.

III. Trampoline vs. Hybrid Approach

In this section we compare the advantages of our approach compared to traditional trampoline based dynamic instrumentation mechanisms. We show the steps followed in trampoline mechanisms, and why our approach has a significant improvement in terms of overhead. The basic process of dynamic instrumentation based on trampoline can be divided into 4 steps

- **Inspection for Probe Points:** This step inspects and generates a binary patch for the custom instrumentation to be inserted to the target binaries, and find the target probe points which are the code addresses to be modified.
- **Memory Allocation for Patching:** Appropriate memory space is allocated for adding the patch and the trampoline code to the target binary.
- **Loading and Activation of a Patch:** At run-time the patch is loaded into the target binary, and overwrites the probe point with a jump instruction to a trampoline function and subsequently to the instrumentation function.
- **Safety and Reliability Check:** To avoid illegal instructions, it is necessary to check for safety and reliability at the HotPatch stage, and that the logic and correctness of the previous binary remains.

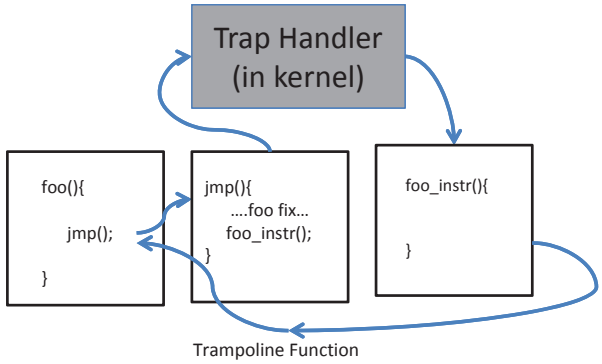


Fig. 4. Traditional Trampoline based Dynamic Instrumentation Mechanisms.

One of the key reasons for better performance of iProbe as compared to traditional trampoline based designs is the avoidance of multiple jumps enforced in the trampoline mechanism. For instance, Figure 4 shows the traditional trampoline mechanism used in existing dynamic instrumentation techniques. To insert a hook for the 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.

In addition pure binary instrumentation mechanisms need to provide complex guarantees of safety and reliability and hence may lead to further overhead. Since the patch and trampoline

functions overwrite instructions at run-time correctness check must be made at HotPatch time so that an instruction overwrite does not result in an illegal instruction, and that the instructions being patched are not currently being executed. While this does not enforce a run-time overhead it does enforce a considerable overhead at the HotPatch stage.

Again iProbe avoids this overhead by offloading this process to the compiler stage, and allocating memory ahead of time.

Another important advantage of our hybrid approach as compared to the trampoline approach is that pure dynamic instrumentation techniques are sometimes unable to capture functions from the raw binary. This can often be because some compiler optimizations may inherently hide function calls boundaries in the binary. A common example of this is *inline functions* where functions are inlined to avoid the creation of a stack frame and concrete calls to these functions. This may be done explicitly by the user by defining the function as *inline* or implicitly by the compiler. Since our instrumentation uses compiler assisted binary tracing, we are able to use the users definition of functions in the source code to capture entry and exit of functions despite such optimizations.

IV. Implementation

The design of iProbe is generic and platform agnostic, and works on native binary executables. We built a prototype on Linux which is a commonly used platform for service environments. In particular, we used a compiler technique based gcc/g++ compiler to implement the hook place holders on standard Linux 32 bit and 64 bit architectures. In this section we first show the implementation of the iProbe framework, and then discuss the implementation of FPerf a tool built using iProbe.

A. iProbe Framework

As we presented in the previous section, the instrumentation procedure consists of two stages.

ColdPatch: In the first phase the place holders for hooks are created in the target binary. We implemented this by compiling binaries using the `-finstrument-functions` flag. 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.

In details this compiler option places function calls to instrumentation functions (`_cyg_profile_func_enter` and `_cyg_profile_func_exit`) after the entry and before the exit of every function. This includes inline functions (see second state in Figure 2). Subsequently, our ColdPatcher uses a binary parser 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` 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>`

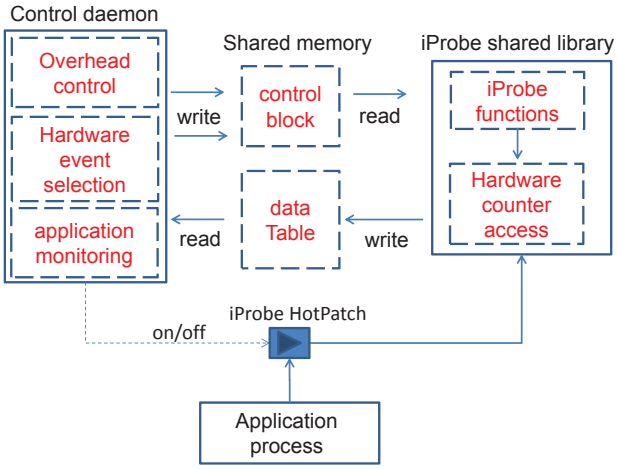


Fig. 5. Overview of *FPerf*: Hardware Event Profiler based on iProbe.

The first field is the instruction offset from the base address, and the second classifies if the target is an entry or an exit point of the function. The meta-data here specifies the file, function name, line number 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 a hash of all possible probing points. iProbe then use the meta-data information to provide users a list of target functions and their respective file information. It takes as input the list of targets and interrupts the target process. 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 a shared memory logger. To avoid any locking overhead, we have a race free mechanism which utilizes thread local storage to keep all logs, and a buffered logging mechanism.

B. FPerf: An iProbe Application for Hardware Event Profiling

We used iProbe to build *FPerf*, an automatic function level hardware event profiler. FPerf uses iProbe to provide an automated way to gather hardware performance information at application function granularity.

Hardware counters provide low-overhead access to a wealth of detailed performance information related to CPU's functional units, caches and main memory etc. Using iProbe's all function profiling, we capture the hardware performance counters at the entry and exit of each function. To control the perturbation on applications and the run-time system, *FPerf* also implements a control mechanism to constraint the

function profiling overhead within a budget configured by users.

Figure 5 summarizes *FPerf* implementation. It includes a control daemon and an iProbe shared library with customized instrumentation functions. The iProbe instrumentation functions access hardware performance counters (using PAPI[17] in the implementation) at the entry and exit of a selected target function to get the number of hardware events occurring during the function call. We define this process as taking one sample. Each selected function has a budget quota. After taking one sample, the instrumentation functions decrease the quota for that application function by one. When its quota reaches zero, iProbe does not take sample anymore for that function.

The daemon process controls run-time iProbe profiling through shared memory communication. There are two shared data structures for this purpose: a shared control block where the daemon process passes to the iProbe instrumentation functions the profiling quota information, and a shared data table where the iProbe instrumentation functions record the hardware event information for individual function calls. When iProbe is enabled, i.e., the binary is HotPatched, daemon periodically collects execution data. We limit the total number of samples we want to collect in each time interval to restrict the overhead. This limitation is important because in software execution, the function call happens very frequently. For example, even with test data size input, the SPEC benchmarks generate 50MB-2GB trace files if we log the records for each function call. Functions that are frequently called will get more samples. Each selected function cannot take more samples than its assigned quota. The only exception happens when one function has never been called before; we assign a minimum one sample quota for each selection function. And we pick a function with quota that has not been used up, and decrease the quota of it by one. The above overhead control algorithm is a simplified Leaky Bucket algorithm [18] originally for traffic shaping in networks. Other overhead control algorithms are also under consideration.

The control daemon also enables/disables the iProbe HotPatching based on user-defined application monitoring rules. Essentially, this is an external control role on when and what to trace a target application with iProbe. A full discussion of the hardware event selection scheme and monitoring rule design is beyond the scope of this paper.

V. Evaluation

A. Overhead of ColdPatch

The SPEC INT CPU benchmarks 2006 [19] is a widely used benchmark in academia, industry and research as relevant representation of real world applications. We tested iProbe on 8 benchmark applications shown in Figure 6. The first column shows the execution of a normal binary compiled without any instrumentation or debug flags. The next column shows the execution time of the corresponding binary compiled using the instrumentation flags (Note here the instrumentation functions are dummy functions). Lastly, we show the overhead of a ColdPatched iProbe binary with `NOP` instead of the call instruction. Each benchmark application was executed ten times using SPEC benchmark tools. 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 non-compute intensive 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. Further reduction of the overhead can be achieved by reducing the scope of the functions which are prepared for function tracing by iProbe; for example only using place holders in selected components that need to be further inspected. Negligible overhead of ColdPatching process of iProbe shows that applications can be prepared for instrumentation (HotPatching) without adversly effecting the usage of the application.

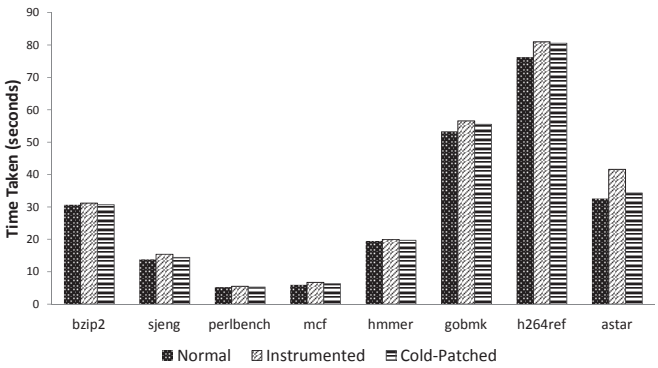


Fig. 6. Overhead of iProbe "ColdPatch Stage" on SPEC CPU 2006 Benchmarks.

B. Overhead of HotPatching and Scalability Analysis

We compared iProbe with UTrace (User Space Tracing in SystemTap) [6], and DynInst [8] on a x86_64, dual-core machine with Ubuntu 12.10 kernel. To test the scalability of these tools, we designed a micro-benchmark and tested the overhead for an increasing amount of events instrumented. We instrumented a dummy application with multiple calls to an empty function `foo`, the instrumentation function in the cases simply increases a global counter for each event triggered (entry and exit of foo). Tools were written using all three frameworks to instrument the start and end of the target function and call the instrumentation function.

Figure 7 shows our results when applying iProbe and SystemTap on this micro-benchmark. 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 subsecond execution, while DynInst takes

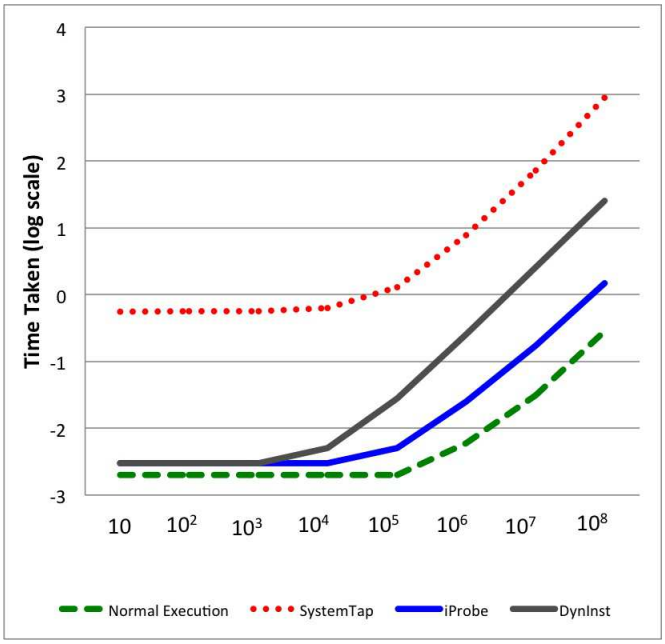


Fig. 7. Overhead and Scalability Comparison of iProbe HotPatching vs. SystemTap vs. DynInst using a Micro-benchmark.

TABLE I
EXPERIMENT PLATFORM.

CPU	Intel Core^{TM} i5-2500 CPU 3.3GHz
OS	Ubuntu 11.10
Kernel	3.0.0-12
Hardware event access utility	PAPI 5.1.0
Applications	SPEC CPU2006

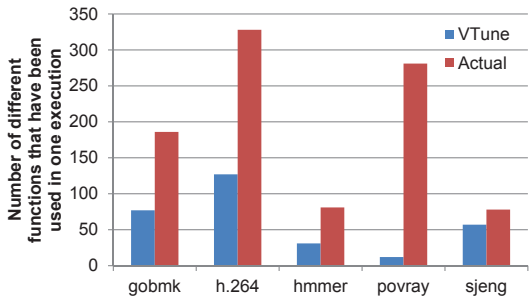


Fig. 8. The number of different functions that have been profiled in one execution.

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 high overhead, and less scalability observed in our results.

C. Case Study: Hardware Event Profiling

1) Methodology: In this section, we present preliminary results on FPerf. The purpose of this evaluation is for the illustration of iProbe as a framework for lightweight dynamic application profiling. Towards it, we will discuss the results in the context of two FPerf features in hardware event profiling:

- **Instrumentation Automation**: FPerf automates hardware event profiling on massive functions in modern software. This gives a wide and clear view of application performance behaviors.
- **Profiling Automation**: FPerf automates the profiling overhead control. This offers a desired monitoring feature for SLA-sensitive production systems.

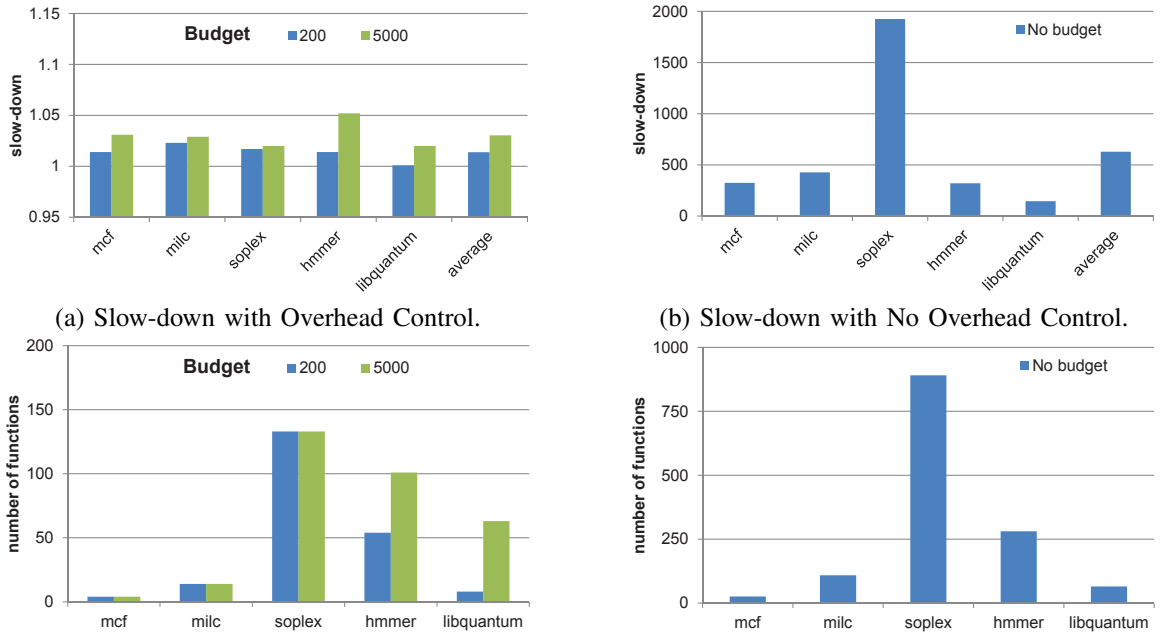
While there are many other important aspects on FPerf to be evaluated such as hardware event information accuracy and different overhead control algorithms, we focus on the above two issues related to iProbe in this paper.

Our testbed setup is described in Table I. The server uses an Intel Core^{TM} i5 CPU running at 3.3GHz, and runs Ubuntu 11.10 Linux with 3.0.0-12 kernel. FPerf uses PAPI 5.1.0 for hardware performance counter reading, and the traced applications are SPEC CPU2006 benchmarks.

2) Instrumentation Automation: Existing profilers featuring hardware events periodically (based on time or events) sample the system-wide hardware statistics and stitch the hardware information to running applications (e.g. Intel VTune [20]). Such sampling based profilers work well to identify and optimize hot code, but with the possibility of missing interesting application functions yet not very hot. In sharp contrast, *FPerf* is based on iProbe framework, it inserts probe functions when entering and exiting each target function. Therefore, *FPerf* can catch all the function calls in application execution. In Figure 8, we use VTune and *FPerf* (without budget quota) to trace SPEC workloads with test data set. VTune uses all default settings. We find that VTune misses certain functions. For example, on *453.povray* VTune only captures 12 different functions in one execution. In contrast, *FPerf* does not misses any function because it records data at enter/exit of each function. Actually, there are 280 different functions have been used in this execution. having the capability to profile all functions or any subset in the program is desirable. For example, [21] reported that in deployment environment, non-hot functions (i.e., functions with low call frequency) might cause performance bugs as well.

FPerf leverages iProbe's all-function instrumentation and functions-selection utility to achieve instrumentation automation.

3) Profiling Automation: We tested the measured performance overhead and the number of captured functions of FPerf with different overhead budget. As shown in Figure 9, the Y axis of Figure 9 (a) and (b) is slow-down, which is defined as the execution time with tracing divided by the execution time without tracing. The Y axis of Figure 9 (c) and (d) is the number of profiled functions. The "budget" legend is the



(a) Slow-down with Overhead Control.

(b) Slow-down with No Overhead Control.

(c) Number of Profiled Functions with Limited Budget.

(d) Number of Profiled Functions with No Budgeting.

Fig. 9. Overhead Control and Number of Captured Functions Comparison.

total number of samples we assign *FPerf* to take. With no budgeting, *FPerf* records hardware counter values at every enter/exit points of each function. From Figure 9 (b) and (d), no budgeting can capture all the functions but with large 100x-1000x slow-downs. In contrast, FPerf showed its ability to control the performance overhead under 5% in Figure 9 (a). Of course, FPerf had the possibility to miss functions, as when the budget is too tight, we only sample a limited number of function enter/exit points.

FPerf leverages iProbe's scalability property (predictable low overhead) to achieve the automation on realizing a low and controllable profiling overhead.

VI. Related Work

Source Code or Compiler Instrumentation Mechanisms: Source code instrumentation is one of the most widely available mechanisms for monitoring. In essence, users can insert debug statements with runtime flags to dump and inspect program status with varying verbosity levels. The log4j [9] and log4c frameworks are commonly used libraries to perform program tracing in many open source projects in the source code level. Additionally compilers have several inbuilt profilers which can be used along with tools such as gprof and jprof to gather statistics about program execution. While source code techniques allow very light weight instrumentation, by design they are static and can only be changed at the start of application execution. iProbe on the other hand offers run-time instrumentation that allows dynamic decisions on tracing with comparable overhead.

Run-time Instrumentation Mechanisms: There are several kernel level tracing tools such as DTrace, LTTng, SystemTap [3], [5], [4] developed by researchers over the years. iProbe differs from these approaches mainly in two ways: Firstly, all of these approaches use a technique similar to software interrupt to switch to kernel space and generate a log event by overwriting the target instructions. They then execute the instrumentation code, and either generate a trampoline mechanism or re-execute the overwritten target instructions and then jump back to the subsequent instructions. As shown in Figure.10 this introduces context-switches between user-space and the kernel, causing needless overhead. iProbe avoids this overhead by having a completely user-space based design. Secondly, all these approaches require to perform complex checks for correctness which can cause unnecessary overhead at both hotpatching, and when running an instrumented binary.

Fay [22] is a platform-dependent approach which uses the empty spaces at the start of the functions available in Windows binaries for instrumentation. To ensure the capture of the entry and exit of functions, Fay calls the target function within its instrumentation thereby introducing an extra stack frame for each target instrumentation. This operation is similar to a mini-trampoline and hence incurs an overhead. Fay logs function execution in the kernel space and hence also has a context-switch overhead. iProbe avoids such overhead by introducing markers at the beginning and end of each function using a

Another well known tool is DynInst[8]. This tool provides a rich dynamic instrumentation capability and has pure back box solution towards instrumentation of any application. However, as shown in Figure.10 it is also based on traditional trampoline mechanisms, and induces a high overhead becauses of unnecessary jump instructions. Additionally it can have higher overhead because of complex security checks. Other similar

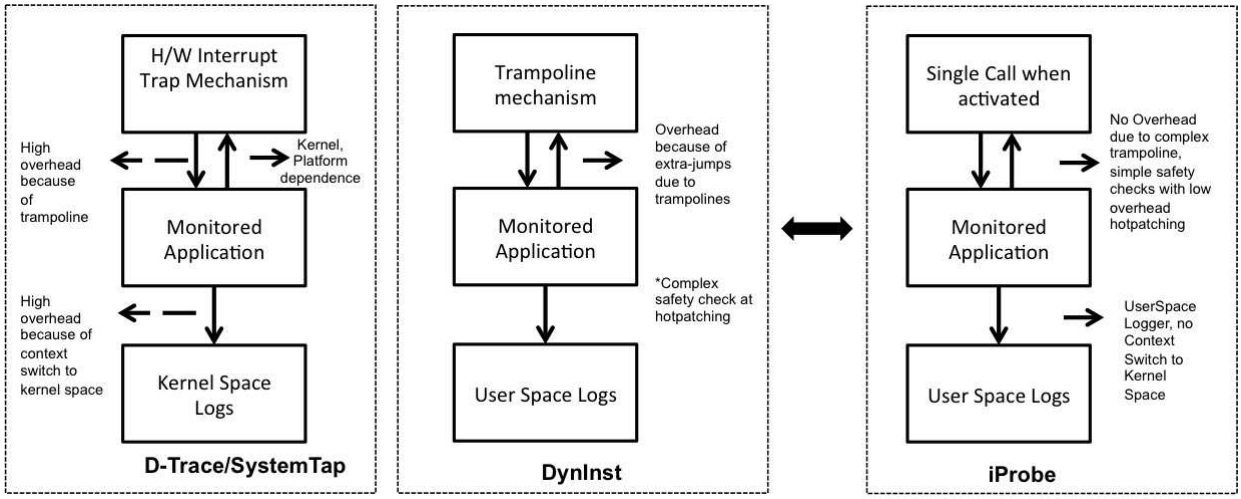


Fig. 10. Advantages of iProbe over existing monitoring frameworks DTrace/SystemTap and DynInst

trampoline based tools like kaho and pannus[14], [13] have also been proposed, but they focus more towards patching binaries to add *fixes* to correct a bug.

Debuggers: Instrumentation is a commonly used technique in debugging. Many debuggers such as gdb [2] and Eclipse have breakpoints and watchpoints which can stop the execution of programs and inspect program conditions. These features are based on various techniques including `ptrace` and hardware debugging support (single step mode and debug registers). While they provide such powerful instrumentation capabilities, there are in general not adequate for beyond the debugging purposes due to overwhelming overhead.

Dynamic Translation Tools: Software engineering communities have been using dynamic translation tools such as Pin [1] and Valgrind [23] to inspect program characteristics. These tools dynamically translate program code before execution and allow users to insert custom instrumentation code flexibly. They are capable to instrument non-debug binaries and provide versatile tools such as memory checkers and program profilers. However, similar to debuggers, they are generally considered as debugging tools and their overhead is significantly higher than runtime tracers.

VII. Conclusion

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

References

[1] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065034

[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)*. Citeseer, 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. [Online]. Available: 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.*, vol. 14, no. 4, pp. 317–329, Nov. 2000. [Online]. Available: http://dx.doi.org/10.1177/109434200001400404

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

[10] log4c: Logging for c library. [Online]. Available: http://log4c.sourceforge.net

[11] Livepatch: http://ukai.jp/software/livepatch/. [Online]. Available: http://ukai.jp/Software/livepatch/

[12] S. Bratus, J. Oakley, A. Ramaswamy, S. Smith, and M. Locasto, "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/. [Online]. Available: http://pannus.sourceforge.net/

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

[16] J. Mussler, D. Lorenz, and F. Wolf, "Reducing the overhead of direct application instrumentation using prior static analysis," in *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, ser. Euro-Par'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 65–76. [Online]. Available: http://dl.acm.org/citation.cfm?id=2033345.2033353

[17] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *In Proceedings of the Department of Defense HPCMP Users Group Conference*, 1999, pp. 7–10.
[18] A. S. Tanenbaum, *Computer Networks, Fourth Edition*. Prentice Hall PTR, 2003.
[19] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006. [Online]. Available: http://doi.acm.org/10.1145/1186736.1186737
[20] Intel, "Vtune amplifier," http://www.intel.com/software/products/vtune, 2011.
[21] M. Jovic, A. Adamoli, and M. Hauswirth, "Catch me if you can: performance bug detection in the wild," in *OOPSLA*, 2011.
[22] U. Erlingsson, M. Peinado, S. Peter, and M. Budiu, "Fay: extensible distributed tracing from kernels to clusters," in *SOSP '11*, 2011.
[23] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *PLDI '07*, 2007.