Autonomous analysis of interactive systems with self-propelled instrumentation *†

Alexander V. Mirgorodskiy and Barton P. Miller

Computer Sciences Dept., Univ. of Wisconsin, 1210 W. Dayton St., Madison, WI, 53706, USA

ABSTRACT

Finding the causes of intermittent bugs and performance problems in modern systems is a challenging task. Conventional profilers focus on improving aggregate performance metrics in an application and disregard many problems that are highly visible to users but are deemed statistically insignificant. Finding intermittent bugs is also hard—breakpoint debuggers change the timing of events, often masking the problem. To address these limitations, we propose a novel approach called *self-propelled instrumentation*—using an autonomous agent to perform self-directed exploration of the system. We inject the agent into a running application, and the agent starts propagating through the code, carried by the application's flow of control. As it propagates, it inserts instrumentation dynamically to collect and analyze detailed execution information. The key feature of this approach lies in its ability to meet three requirements: high level of detail, low overhead, and autonomy (here, little reliance on human help). Existing techniques for tracing and profiling violate at least one of the requirements. As a proof of concept, we implemented a tool called *spTracer* that uses self-propelled instrumentation to obtain function-level traces from applications and the kernel. The tool enabled us to locate and fix three problems in unfamiliar code in the Linux environment.

Keywords: tracing, performance analysis, dynamic instrumentation

1. INTRODUCTION

Finding the root causes of bugs and performance problems in modern systems is a challenging task. The focus of conventional profilers has been on improving aggregate performance metrics in an application, not the latency of individual operations. However, latency-related problems are common in end-user desktop environments as well as in large production systems. For example, streaming video, telephony, and most GUI applications are particularly sensitive to short delays in the execution. Although such delays are highly visible to the user, they are often ignored by traditional profilers as statistically insignificant. Finding intermittent bugs is also hard, especially in concurrent environments—debuggers change the timing of events, often masking the problem.

We propose to identify intermittent bugs and performance problems by autonomously following the flow of control in applications and operating system kernels. While systems such as *DIOTA*,[22] *Dynamo*,[4,6] and *PIN*[13] offer an effective way to follow the control flow, our approach has not been previously explored because of the significant warm-up overhead of such mechanisms. These techniques often demonstrate reasonable performance for the steady-state case of long-running applications. However, they introduce severe perturbation for some time after activation. In contrast, our mechanism has low warm-up overhead, which allows us to apply it to studies of latency-sensitive systems.

For an analysis technique to be practical in real-world environments, it needs to satisfy three main requirements. First, it should be **autonomous**, able to perform investigation without heavy reliance on human help. In

† This work is supported in part by Department of Energy Grants DE-FG02-93ER25176 and DE-FG02-01ER25510, and Office of Naval Research grant N00014-01-1-0708. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

Authors' E-mail: {mirg,bart}@cs.wisc.edu

production environments, users can often see the symptoms of a problem, but they may not understand where to search for its root cause. Second, it should be **detailed** to not miss relevant facts. Aggregated or coarse-grained performance data is often insufficient for determining the cause of a problem. Third, the approach should introduce **little perturbation** to the system. Changing timing of events in the system may hide intermittent problems. Furthermore, slowing down a production system that services other users may be unacceptable.

Currently, no existing analysis technique satisfies all the three requirements. Performance profilers[16, 25, 26] are autonomous and often have acceptable overhead, yet they fail the detailed test as they collect only aggregate metrics about the execution. Finding short and intermittent problems is often impossible with such mechanisms. Breakpoint debuggers require user guidance, failing the autonomy and the low perturbation requirement.

Tracing techniques fail either the autonomy test or the low perturbation test. Some tracing tools allow insertion of probes at user-specified locations in the code.[8, 18] In the hands of an experienced analyst, this approach can provide an arbitrary level of detail with controlled overhead. However, it is not autonomous and requires significant user expertise to know where to insert the trace probes. Furthermore, the lack of autonomy in such user-directed approach often implies multiple re-runs of the test to narrow down the cause of the problem. Analyzing problems that are hard to reproduce may not be possible with this approach.

Some tools also allow insertion of trace calls at all possible locations in the system (e.g., all function entries[8, 33]). This approach is detailed and autonomous, but it fails the low-perturbation requirement. If all binaries in the system are pre-instrumented with trace probes, the users will suffer from significant overhead even when no tracing is being performed. An alternative approach is to insert all the probes at run time, when the user decides to start tracing. This approach will have adverse latency effects on performance: run-time insertion of trace probes at all possible locations is a long operation.

Self-propelled instrumentation does satisfy the three requirements and has several additional practical features. The techniques that enable these features are described below.

- **Autonomy**. To perform self-propelled instrumentation, we inject an agent code into the system. Once injected, the agent propagates autonomously with the system's flow of control, collecting information and making further propagation decisions on its own.
- **Little perturbation**. The agent propagates via dynamic instrumentation performed within the same address space. Such code insertion is done with local memory operations, without expensive system calls or context switches.
- **High level of detail**. The agent is programmed to intercept the flow of control at the granularity of individual functions. Past experience with standard profilers suggests that function-level data often provide the right level of detail for performance analysis. Intercepting the flow of control at a lower level (individual basic blocks or even instructions) may also be possible, albeit at the price of higher overhead.
- **Rapid activation**. The agent can start data collection on demand, without costly preparation. This feature is important for analysis of short-lived and intermittent problems.
- **Ability to analyze black-box systems**. Self-propelled instrumentation can cut through non-transparent layers of code, including the kernel boundary. This capability is crucial for introspection of complex multi-component systems.

As a proof of concept, we implemented an autonomous tracing tool called *spTracer* that uses self-propelled instrumentation to propagate through the code. The tool lets the user collect detailed function-level traces of an application and the kernel as follows. First, the user specifies two events as tracing start and stop points (e.g., start when a particular key combination is pressed, stop one second later). When the start event happens, we inject the tracer into a currently-executing application and it starts propagating through the code with the application's flow of control. When the application performs a system call or is preempted by a hardware interrupt, the tracer will seamlessly walk into the kernel and continue propagating through the kernel code. The propagation procedure continues until the second user-specified event happens. In the end, we obtain and visualize a complete log of functions invoked between the two events.

To demonstrate the usefulness of the approach, we analyzed three problems in the x86 Linux environment. First, we traced the cause of choppy DVD playback in the *MPlayer* movie player application to a problem in the Linux kernel. Second, we found the cause of a two-second freeze on access to the menu bar of Microsoft Word running under the Wine MS-Windows emulator. Third, we were able to identify why some Linux laptops cannot wake up after entering the standby mode. Self-propelled instrumentation proved to be a powerful mechanism for obtaining detailed and accurate execution information.

Finally, we measured the overhead of self-propelled instrumentation. Instrumenting *MPlayer* in the course of DVD playback resulted in less than 35% slowdown. For short periods of time, this level of slowdown is often tolerable even for performance analysis of latency-sensitive multimedia systems. If not, one can reduce the overhead further by instructing self-propelled instrumentation to avoid tracing through certain parts of the code. For example, it can stop following deeply nested function calls or repetitive calls to a function in a tight loop. The flexible nature of self-propelled instrumentation makes such policies easy to implement.

2. RELATED WORK

Conventional approaches to performance analysis and debugging are often unsuitable for time-sensitive systems. A common technique for performance analysis is full-system profiling made possible by tools like *DCPI*[3] and *OProfile*.[12] As mentioned above, the focus on aggregate system performance often makes such tools ignore short one-time problems. To find such problems, a more detailed analysis technique is required.

Standard debuggers are often unsuitable for finding bugs in time-sensitive systems, because of the introduced overhead. Pausing a multithreaded application and stepping through its code may change the timing of events and hide subtle race conditions. Pausing highly-available systems that use watchdog timers[30] may not be possible at all: if a watched application does not respond for a certain time interval, it is killed and restarted.

We partition further discussion of related work into three sections. First, we describe techniques that have been used for performance analysis and debugging of time-sensitive systems. Second, we describe existing mechanisms for collecting detailed execution traces regardless of their applications. Finally, we discuss existing approaches for automated analysis of trace data for anomaly detection.

2.1. Performance analysis and debugging of interactive systems

To address the inability of standard profilers to find short performance problems in interactive systems, the *TIPME* project[15] uses constant monitoring via tracing. As the system runs, it logs certain events (context switches, process creation, events passing through the X server, and a few others) in a circular buffer in main memory. When a performance problem happens, the user can hit a key combination to save the trace to disk for later analysis. The approach proved useful in finding several problems in the OS process and disk schedulers.

The *TIPME* framework records events of a few pre-defined types and is not easily extendable. Adding another event type to *TIPME* appears to require source code modification and recompilation of the kernel and possibly the X server. A fixed set of high-level events is unlikely to be sufficient for locating the cause of an arbitrary performance problem. Past experience with standard profilers shows that some problems call for more detailed function-level performance data.

For detailed analysis, developers of software for embedded systems often use simulators. With recent advances in simulation technology,[23,32] similar techniques can be applied to general-purpose systems.[2] The approach is able to collect function-level or even instruction-level performance data with no perturbation to the system's execution. Furthermore, it provides deterministic replay capabilities, useful for finding intermittent bugs.

Despite these advantages, full-system simulators have two drawbacks. First, they introduce the slowdown of 30-75 times relative to native execution.[23] This slowdown eliminates the possibility of online diagnostics and complicates collection of data for long-running programs. Second, to analyze a real-world problem with such techniques, one needs to reproduce the problem in a simulated environment. For hardware-related problems or intermittent bugs that are highly sensitive to changes in system configuration, it may be difficult to achieve.

2.2. Existing mechanisms for trace collection

There are several mechanisms that can trace execution of applications and kernels. Trace statements can be inserted statically at functions' entries and exits by a source-to-source translator,[21] a compiler,[19] or a binary rewriter.[34] The common drawback of static mechanisms lies in their inability to remove instrumentation when tracing is not required. The overhead study in Section 5.4 shows that executing null instrumentation on every function call and return may still result in more than 20% slowdown. Users of static instrumentation frameworks would have to endure this slowdown for the entire running time of the system.

In contrast, dynamic techniques can activate and deactivate tracing at run time. One of the most well-known dynamic tracing tools is *strace*.[1] *Strace* can launch an application and print the names and arguments of system calls that the application executes. This functionality proved useful for quick insight into program's behavior. However, a trace of system calls often does not capture enough details of the execution. It does not allow tracing regular function calls in the application. Similarly, it provides no insight into calls made in the kernel mode.

This limitation is partially addressed by a tool called *itrace*.[31] *Itrace* provides an interface similar to *strace*, but records all function calls in an application (it does not support tracing the kernel code). To obtain this information, *itrace* single-steps through instructions in an application and generates a trace record whenever a call or a return instruction is encountered. The approach is highly intrusive as it incurs a context switch after every instruction in a traced process. Although it may be tolerable for debugging some time-insensitive applications, this approach is not suitable for performance analysis.

Another approach for obtaining function-level traces is implemented in the *LTrace* tool.[9] *LTrace* traces calls to functions in shared libraries by inserting breakpoints at entries of all exported functions. The Solaris *Truss* tool[35] uses the same technique, but is able to instrument all functions present in the symbol table, not only those exported by shared libraries. With this mechanism, most of the applications code executes natively, without single-stepping. However, the overhead of a breakpoint trap on every function call is still prohibitively expensive for performance analysis of many applications.

A more efficient tracing approach is chosen by the *TraceTool* project.[33] It uses dynamic instrumentation capabilities of Dyninst API[7] to insert trace calls at all function entries and exits in an application. Similarly, the *DTRACE* framework supports instrumentation of both user and kernel functions.[8] The downside of en-masse instrumentation mechanisms is significant latency of inserting all trace statements. To avoid introducing such delays at run time, one has to pre-instrument all functions in advance. However, this approach has the same limitation as static instrumentation—it introduces noticeable overhead even when tracing is not being performed.

2.3. Collecting traces by following the flow of control

To reduce the cost of inserting trace statements, dynamic instrumentation can be used to inject them incrementally, as the flow of control in an application reaches new code. There are several projects that can follow the flow of control. The techniques most similar to self-propelled instrumentation are *Dynamo*,[4,6] *DIOTA*,[22] and *PIN*.[13] The primary goal of *Dynamo* is dynamic program optimization. *DIOTA* and *PIN* provide general-purpose instrumentation facilities. The systems use similar mechanisms for following the flow of control.

The projects build a cache (clone) of the code on the fly and execute the cached fragments instead of the original code. When a cached fragment attempts to jump to an uncached location, the framework takes control, creates a new fragment for the uncached location, copying and adjusting the instructions, and executes it, giving up control until the next jump out of the cache. To obtain an execution trace with this technique, *DIOTA* and *PIN* insert calls to tracing routines into new fragments as they are being generated. *DIOTA* supports tracing entries and exits of individual basic blocks. *PIN* also supports tracing of function entries and exits.

Although *DIOTA*, *PIN*, and *Dynamo* offer capabilities similar to those of self-propelled instrumentation, they may be unsuitable for analysis of latency-sensitive systems. Building the code cache is a slow operation. Each instruction at an uncached location must be examined and moved to the cache, adjusting position-dependent instructions accordingly. Until the cache is reasonably populated, the traced application will suffer from substantial overhead. Even after running for some time, the application may still experience slowdown when the flow of control reaches previously-unseen code. Analyzing latency-sensitive systems and finding the root causes of intermittent delays using a mechanism that introduces its own intermittent delays may not be possible.

2.4. Existing mechanisms for trace analysis

After collecting a detailed trace of events, we need to analyze it and search for anomalous behavior. If the trace is complex, its visual examination may not be feasible. Two recent projects for automated analysis of traces in large multi-component Internet services are *PinPoint*[10] and *Magpie*.[5] Although both systems collect relatively coarse-grained information, their analysis methods may also apply to finding anomalies in function-level traces.

PinPoint follows client requests through the system, collects names of components involved in processing them, and detects whether each request succeeded or failed. Per-request traces are later clustered to identify components that are highly correlated with failures. *Magpie* also follows requests through the system, but records certain high-level events (context switches, remote procedure calls, system calls, and network communications) rather than component names. Strings of events are later grouped into clusters, based on their similarity. Strings that are not close to any existing cluster are considered anomalous. Later, *Magpie* builds a probabilistic state machine and identifies the likely cause of the anomaly—an event that has suspiciously low probability.

3. OVERVIEW OF THE APPROACH

We have devised a novel approach for performance analysis and debugging of complex systems that we called *self-propelled instrumentation*. The corner stone of our approach is an autonomous agent that performs self-directed exploration of the system. In a typical usage scenario, the agent attaches to a running application and stays dormant, waiting for a user-specified activation event. When the event happens, the agent starts propagating through the application's code, carried by the flow of execution. The agent takes control at a point of interest, executes user-specified *payload code* (e.g., collects performance data), decides where it wants to receive control next, and lets the application continue. Below, we outline main elements of activation, propagation and payload execution. The detailed implementation of these steps is discussed in Section 4.

Activation. The agent stays dormant inside an application until a user-specified event happens. Currently, the agent can wake up on a function call (when a specified function in the application is executed, similar to a traditional breakpoint), on a timer event (at a certain time), and on a key press (when a certain key combination is pressed). In the future, we plan to extend the supported set of events to include common user-interface actions.

Propagation. To propagate, the agent uses *in-situ instrumentation*, a kind of dynamic instrumentation performed within the same address space. The agent examines the current location in the code where the application executes, finds the next point of interest ahead of the flow of control (we call it an *interception point*), modifies the code to insert a call to itself at that point, and allows the application to continue. When the application hits the instrumented point, the agent will receive control and propel itself to the next point.

The characteristic feature of our approach is low and predictable overhead. First, unlike *DIOTA* and *PIN*, we instrument the code in-place, without generating a cached copy where possible. This technique allows us to start executing the code at full speed, obviating the need for the costly cache-buildup period. To bring the instrumentation costs further down, we perform code analysis in advance, before the application starts running. The key benefits of this approach are rapid agent activation and elimination of intermittent delays when the flow of control reaches previously-unseen code in the middle of execution.

Second, our technique may be able to achieve lower overhead than *DIOTA*, *PIN*, and *Dynamo* even in the steady-state scenario. After the code cache is populated, these systems have to continue intercepting all indirect jumps to prevent the control from returning to the original uncached code. This fact accounts for the major share of residual overhead in *DIOTA*.[22] In contrast, our in-situ technique can support indirect jumps without interceptions, because we execute original code.

As our studies in Section 5 show, latency-related problems in the kernel can manifest themselves in user applications. To investigate such problems, we can propagate through both the application and the kernel. When a traced application performs a system call or is preempted by a hardware interrupt, the agent walks into the kernel and starts propagating through the kernel code. Our mechanism uses local memory operations for instrumentation and allows us to use the same techniques on both sides of the kernel boundary.

Payload execution. For extensibility, the agent executes user-specified payload code at each interception point. The payload can passively record and analyze execution information. This approach can be used for

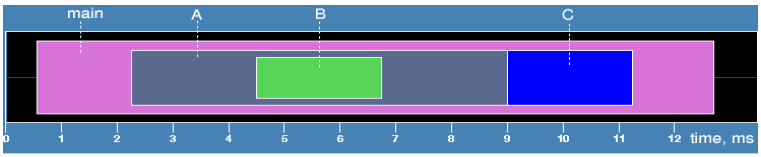

Figure 1. A sample trace of functions

performance analysis and automated bug finding. The payload can also react to observed information, potentially altering the execution. Two common applications of such an active approach are security analysis (e.g., monitor a process and prevent it from executing malicious actions) and dynamic optimization.

Our current implementation uses the passive monitoring approach to record the time stamp of reaching each interception point along with the point identifier. We intercept the flow of control at functions' entries and exits, collecting a function-level trace of execution. Once the trace is collected, the agent saves it to disk and we examine it visually for anomalies. In the future, we plan to automate the analysis process to let the agent find bugs and performance problems with little human help, obviating the need for collecting and storing the trace.

To visualize function-level traces, we use the *Nupshot* tool,[17] as shown in Figure 1. Each function in the trace is represented by a rectangle. The width of the rectangle is the duration of the function's execution in milliseconds. Nested rectangles represent nested function calls. In Figure 1, Function `main` calls Function `A`, `A` calls `B`, `B` returns to `A`, `A` returns to `main` and then `main` immediately calls `C`.

4. IMPLEMENTATION

Using techniques described in the previous section, we implemented an autonomous tracing tool, called *spTracer*. The tool propagates through the code with self-propelled instrumentation and collects traces of function calls in a chosen application and the kernel made between two user-specified events. To discuss its implementation, we start with propagation mechanisms and then discuss the questions of activation in the user and kernel domains.

4.1. Propagation

spTracer records all function entries and returns. To obtain such information with self-propelled instrumentation, we treat call sites in the code as interception points. That is, *spTracer* intercepts all calls made by a traced application to receive control before each function is entered and after each function returns. At each entry, *spTracer* makes the next propagation step—intercepts all call sites in the function being called. As the application executes, *spTracer* propagates through its call graph, moving from a function into its callees, and into their callees.

Suppose that *spTracer* received control before entering Function `A`, and Function `A` contains a call to Function `B`. To intercept the call to `B`, *spTracer* replaces the "`call B`" instruction with a jump to a run time-generated patch area. The patch area contains calls to tracing routines and the relocated call to `B`: "`call tracerEntry(B); call B; call tracerExit(B)`". When the `tracerEntry` routine is reached, it emits a trace record for `B`'s entry and propagates the instrumentation into `B`. (It uses pre-computed code analysis results to locate all call sites in `B`, generates patch areas for each site, and replaces the call instructions with jumps to the corresponding patch areas). When the `tracerExit` routine is reached, it simply emits a trace record for `B`'s exit.

In a previous implementation of our tracer, we generated no per-site patches and simply replaced the "`call B`" instruction with "`call tracerEntry`". When `tracerEntry` was reached, it used the return address to look up the original destination of the call (Function `B`) in a hash table and jumped to `B`, returning control to the application. To intercept the flow of control when `B` exits, `tracerEntry` modified `B`'s return address on the stack to point to `tracerExit`. Although this scheme is more memory-efficient (no need for per-site patches), it had substantially higher run-time overhead because of two factors. First, performing a hash lookup on every function call is expensive. Second, modifying `B`'s return address on the stack resulted in mis-predicting the destination of the corresponding return instruction by the CPU. This penalty can be significant on modern processors.[22]

4.1.1. Instrumenting indirect calls

Intercepting regular call instructions is straightforward. On platforms with fixed-size instructions, intercepting indirect calls (e.g., calls through function pointers and C++ virtual functions) can be equally simple: we could still overwrite an indirect call with an unconditional branch to a patch area. However, our target platform, x86, has a variable-length instruction set. On x86, an indirect call instruction can be as small as two bytes, so we cannot replace it with the long five-byte jump.

To address this problem, we relocate basic blocks that contain short indirect call instructions as follows. First, we identify the basic block that contains the indirect call instruction. Second, we copy the block contents to a heap location, adjusting position-dependent instructions appropriately. When copying the indirect call, we also insert calls to `tracerEntry` and `tracerExit` around it. Finally, we write a jump to the relocated block at the beginning of the original block, such that further execution flows through the relocated block.

Block relocation is similar to building the code copy as performed by *Dynamo*,[4] *DIOTA*,[22] and *PIN*.[13] Unlike these systems, we do not relocate the entire code of an application. Only blocks that contain short indirect calls are moved to the heap. Furthermore, we perform most of the analysis and code generation off-line, before the application starts running. These factors allow us to lower the run-time overhead of generating relocated code.

Another similar technique is relocation of a single function as performed by Dyninst.[7] There are two advantages to doing relocation at the basic block level. First, relocating a block is simpler and more efficient. It can be done with a single pass over the code of the block, while function relocation needs post-processing to adjust the destinations of forward branches. Second, function relocation is not always possible. For example, functions that contain jump tables are not relocated by Dyninst because of the involved complexity of adjusting indirect jumps in the code.

Note that block relocation is not always possible either. The assumption of the technique is that it is possible to overwrite the entry point of a block with a jump to the relocated version. Blocks longer than four bytes can be relocated with a single long jump. Blocks longer than one byte can often be relocated with a short jump to a springboard location,[36] which contains the long jump to the relocated version. Finding space for the springboard is usually possible by displacing a long (10 bytes or longer) block nearby. Finally, relocating one-byte blocks is not possible with our technique. Fortunately, we do not need to instrument such blocks as they cannot contain call instructions. For other purposes, we used trap-based instrumentation and function relocation in the past.

4.1.2. Code analysis

To perform instrumentation as described above, the tracer needs to parse the binary code of an application and the kernel. In particular, it must be able to find all call sites in any function being instrumented and determine their destinations (i.e., it needs to know the call graph of the application). When instrumenting indirect calls, it relocates the corresponding basic blocks of a function to the heap. To support this functionality, it also needs to obtain the control flow graph for any function with indirect calls.

Our prototype obtains both the call graph and the control flow graphs before the application starts running. We leverage the code analysis infrastructure of Dyninst API[7] to retrieve and save these data structures to disk. To obtain the graphs for the Linux kernel, we use Dyninst to analyze the on-disk `vmlinux` ELF image of the kernel. (Currently, we do not analyze loadable kernel modules, although it can be done similarly). When the application starts, our tracer fetches the data from disk and begins using them when the user enables tracing. In the future, we envision a system-wide repository that will keep results of code analysis for installed applications.

4.2. Activating user-level tracing

We trace an application and the kernel at the same time. Both tracing components use the same propagation mechanism, but separate activation methods. The user-side agent is implemented as a shared library that is pre-loaded into applications. Pre-loading can be done when an application starts (via the `LD_PRELOAD` mechanism) or at run time (via the *Hijack* mechanism[38]). In both cases, the library stays dormant until the user asks the system to initiate tracing. Our case studies motivated us to implement three primitive activation methods.

The first method activates the tracer when a particular function in the application is executed. Naturally, it assumes that the user is familiar with internals of the application to specify the name of such function. When

the agent is injected into the application, it instruments the chosen function and lets the application continue. When the instrumented function is reached, the agent receives control and starts propagating.

The second method initiates tracing when a user-specified time interval elapses since the start of an application. This mechanism is useful for analyzing problems that happen at a fixed moment and do not depend on input from the user. To receive control on timer expiration, the tracer library uses the `alarm` system call to arrange for a `SIGALRM` signal to be delivered to the application in the specified number of seconds. The tracer library catches the signal, determines where the application was executing before the signal, instruments that function and the agent starts propagating from that point as usual.

The third method activates the tracer when the user hits a particular key combination. To be notified of keyboard events, we implemented a Linux kernel module that intercepts keyboard interrupts (IRQ1). When the application starts, our library issues an `ioctl` system call to register the application with the kernel module and enable this snooping ability. When the module receives a keyboard interrupt, it checks if the key combination is pressed and sends a `SIGPROF` signal to the registered application to initiate tracing. The library will then catch the signal and process it similarly to `SIGALRM` events.

4.2.1. Stack walking

The activation methods discussed above have a common shortcoming of tracing only a subgraph of the application's call graph. Indeed, they start tracing at a particular function and propagate the instrumentation into its callees and further down the call graph. However, the callers of the initial function will not be instrumented and the tracing may stop after the initial function returns. If tracing starts in a function which contains no calls, we will not see any trace records.

To address this problem, we augmented the starting mechanisms with a stack walking capability. When tracing starts, we walk the stack and instrument not only the current function, but also all functions on the stack up to `main`. This technique allows us to continue tracing after the current function returns and obtain complete traces. However, stack walking techniques are not fully robust. Optimizing compilers often emit functions with no stack frames or with stack frames that do not use the frame pointer. Walking the stack past these functions or even detecting such cases is unreliable and may crash the application.

To obviate the need for stack walking, we plan to implement a new scheme. When tracing starts, we will instrument not only call sites in the current function, but also all its return instructions. When the return-point instrumentation is reached, the `%sp` register points to the return address. Therefore, the tracer will be able to find the caller of the current function and propagate into it. In turn, it will also instrument the caller's return points to advance to its caller when necessary. Notice that the return-point instrumentation is needed only to walk out of functions that were on the stack when we started tracing. It can be removed when reached.

4.3. Activating kernel-level tracing

If a traced application issues a system call or is preempted by a hardware interrupt, the tracer starts recording kernel functions until the application resumes user-level execution. The kernel tracer uses the same propagation mechanisms and shares the code base with the user-level counterpart. For controlling kernel tracing, we implemented a Linux kernel module */dev/ktracedrv* with the following `ioctl` interface.

To activate kernel tracing, the application issues the `KTRACE_START ioctl` command. Typically, our framework issues this command together with starting user-level tracing. The command makes the driver pre-instrument two kernel entry points: `do_IRQ` for following hardware interrupts and `system_call` for following system calls. From that moment, if one of these events happens, we will collect all incurred function calls into the module's trace buffer. Similarly, to stop kernel tracing, the application needs to issue the `KTRACE_STOP ioctl` command. When *ktracedrv* sees the command, it pulls out all inserted instrumentation, restoring the kernel image to the original state. The collected data are later returned to the user space via another `ioctl`.

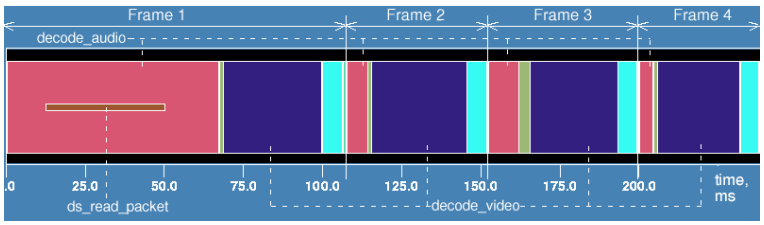

Figure 2. A user-level trace of *MPlayer* (irrelevant details hidden)

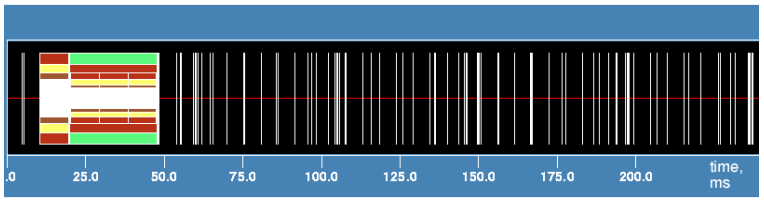

Figure 3. Kernel activity while *MPlayer* is running

5. EXPERIMENTAL RESULTS

To evaluate our new approach, we applied it to analysis of two performance problem (choppy DVD playback in *MPlayer* and non-responsive GUI in *Microsoft Word* running under the *Wine* emulator) and a system crash. The problems required different techniques for trace collection, so we describe them separately. At the end of the section, we compare the overhead of self-propelled tracing to that of existing approaches.

5.1. Analysis of a multimedia application

MPlayer is a popular multimedia player for Linux and several other platforms.[27] It supports a wide variety of video and audio formats and works well even on low-end consumer systems. However, in our experience, its quality of DVD playback is not always adequate: the player often drops frames, resulting in user-visible freezes in the playback even on modern systems. As the symptoms are relatively common, the *MPlayer* community has accumulated a knowledge base of potential fixes. Yet, we decided to proceed autonomously, to model real world conditions where many bugs are too specific to be found with community help. In the end, the ease of tracing with self-propelled instrumentation allowed us not only to fix the problem, but also to identify its root cause.

To investigate the problem, we used *MPlayer* to display a short DVD trailer on a low-end Pentium II 450 MHz desktop computer running Linux. The system satisfies typical hardware requirements for DVD playback.[14] In fact, we verified that the hardware capabilities of the machine were adequate for the task: the trailer displayed accurately under two commercial DVD players in Microsoft Windows. At the same time, playback under Linux was choppy and *MPlayer* reported frames being dropped.

To find the cause of the problem, we subject it to self-propelled tracing as follows. When the player starts, we pre-load the tracer library into it using the `LD_PRELOAD` mechanism. When the library initializes, it requests an alarm signal to be delivered to it 10 seconds later. The signal arrives when *MPlayer* is in the middle of playback. The handler in the tracer library catches it and initiates instrumentation. It collects the trace for 200 milliseconds (several frames of the trailer), pulls out the instrumentation and lets the test run to completion. When *MPlayer* terminates, we save the trace to disk to visualize with the *Nupshot*[17] tool later.

Nupshot proved powerful in navigating through collected traces interactively, zooming in on relevant fragments. Yet, a printed snapshot of the data may be hard to understand because of significant volume of irrelevant details. To address this problem, Figure 2 displays a real trace from *MPlayer* showing functions called directly from `main` and a few other relevant details. We see a periodic sequence of events—four fragments of similar structure correspond to processing four sequential frames. Note that the timing of events in the first frame is clearly different from the rest of the frames. Frames 2, 3 and 4 spend on average 7 milliseconds in `decode_audio` and more than 27 milliseconds in `decode_video`. While Frame 1 spends a similar amount of time (31 milliseconds) in `decode_video`, it spends significantly more (66 milliseconds) in `decode_audio`.

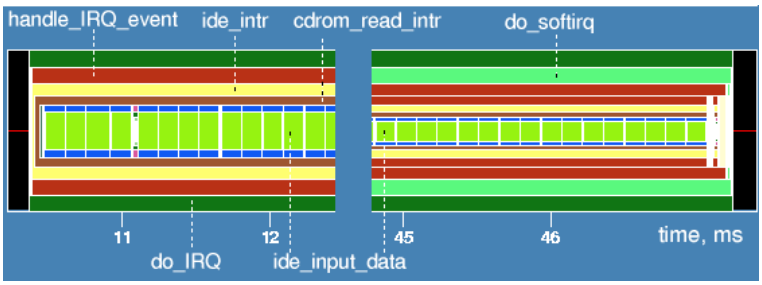

Figure 4. Kernel activity under magnification

The trace reveals that more than 50% of the time spent in `decode_audio` is attributed to a single call to the `ds_read_packet` function. This finding is surprising for two reasons. First, the trace shows that the time was spent in `ds_read_packet` itself, not in its callees. Second, the time was not spent executing in a loop. Although the binary code of `ds_read_packet` does contain one loop, it calls `memcpy` on every iteration and the trace did not contain repetitive calls to `memcpy`.

The long delay in a short straight-line code sequence suggests that *MPlayer* might be preempted in the middle of the function. After collecting additional instances of the trace, we saw similar long delays in other parts of the code, both expected (e.g., the `read` system call) and surprising (e.g., in calls to the `malloc` memory allocation routine). To verify the preemption hypothesis, we turn to results of kernel tracing. Again, to hide irrelevant details, we capture only preemption events by starting kernel instrumentation from the `do_IRQ` kernel entry point. Figure 3 shows that the system is mostly idle, except periodic timer interrupts triggered every 10 milliseconds. However, there is a burst of kernel activity right at the time when `ds_read_packet` was executing. The data suggest that the time attributed to `ds_read_packet` was spent in the kernel, handling hardware interrupts.

To find the cause of preemption, we examine the kernel trace at a closer range. Figure 4 shows that the time is spent in `cdrom_read_intr`, called from `ide_intr`. This fact suggests that the system was handling an IDE interrupt, that signaled that the DVD drive had data available to be read. To handle the interrupt, `cdrom_read_intr` called `atapi_input_bytes` in a loop, which immediately called `ide_input_data`. By looking at the source code for `ide_input_data`, we found that it spent the time reading the data from the drive 4 bytes at a time with `insl` (input long from port) instructions.

The fact that fetching the data from the drive buffers is the ultimate bottleneck suggests a fix for the problem. By enabling DMA (Direct Memory Access) we may allow the DVD drive to put the data in main memory without CPU intervention. The Linux IDE driver allows the superuser to change its settings at run time with the *hdparm* utility. As soon as we enabled DMA transfers for the DVD drive, *MPlayer* stopped dropping frames and the problem went away. New kernel traces reveal that DMA greatly offloads the CPU and allows the system to spend much less time handling IDE interrupts. The time to handle a single IDE interrupt (one invocation of `ide_intr`) has shrunk from 9.0 milliseconds to 0.6 milliseconds.

After searching the Internet, we confirmed that problems with choppy DVD playback on Linux are often solved by enabling DMA. DMA is not used by default as certain old IDE chipsets are reported to cause data corruption in these conditions.[28] To summarize, self-propelled tracing was able to correctly identify the cause of a short and intermittent performance problem. Its ability to cross the kernel boundary proved to be a powerful mechanism that let us trace the problem into the kernel.

5.2. Analysis of a GUI application

Wine[37] is an application that allows users to run Microsoft Windows applications in the X Windows environment. While it works surprisingly well in most cases, *Wine* is not always able to match the native execution speed for some Windows applications. Often, GUI applications under *Wine* become noticeably less responsive. For example, *Wine* sometimes takes several seconds to display a drop-down menu in Microsoft Word. This behavior is a minor nuisance as it happens only on the first access to the menu bar after the application starts. Nevertheless, it lets us demonstrate the general approach to pinpointing such problems with self-propelled instrumentation.

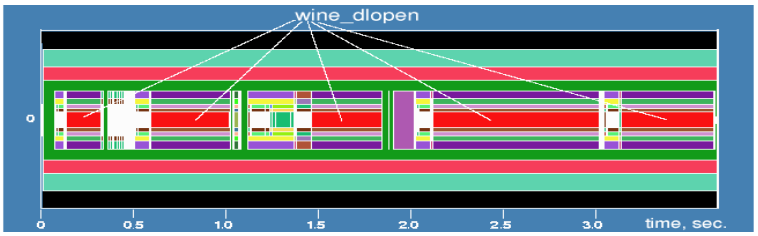

Figure 5. Trace of functions on opening a drop-down menu (low-level details hidden)

In this study, we initiate tracing when the user hits a particular key combination for the first time after launching *Wine* (we use "Alt-F" to open the drop-down File menu). We stop tracing when the user hits any key again. Although the key press activation method worked for other applications, we encountered a serious problem with *Wine*—our stack walking routine was unable to walk the stacks of *Winword* running under the emulator. A closer look at the binary code of the application revealed that *Wine* (a Linux binary) directly executes the *Winword* code (a Windows binary). Portions of the *Winword* code and supporting libraries were optimized not to use the standard frame pointer, making stack walks virtually impossible.

In the future, the technique described in Section 4.2.1 will eliminate our reliance on stack walks. As an interim workaround, we use the following mechanism to trace *Wine*. We activate the self-propelled agent on application startup to follow the flow of control in *Wine*, but emit no trace records. When the "Alt-F" combination is pressed, all functions currently on the call stack are already instrumented. Therefore, the agent is able to start tracing without walking the stack.

A function-level trace of *Wine* obtained with this technique is displayed in Figure 5. The data show that *Wine* spent most time in five calls to `wine_dlopen`. After examining its source code, we found that this function loads dynamic libraries at run time. Apparently, on the first access to the menu bar, Microsoft Word needs to load five dynamic libraries from disk. The operation is expensive and it produces a user-visible freeze in the middle of displaying the menu. Further accesses to the menu bar will trigger no calls to `wine_dlopen`, as all required libraries will have been loaded by that time.

To identify the libraries that *Winword* tries to load, we modified the source code for `wine_dlopen` to print the name of a library being loaded. To our surprise, the majority of libraries loaded on a menu click were multimedia-related, including winmm.dll, a standard component of Windows Multimedia API, and wineoss.drv.so, the Wine OSS (Open Sound System) driver. Apparently, when the user opens a drop-down menu in *Winword*, the system tries to play an audible click. To play a sound, *Wine* needs to load multimedia libraries. After discovering the cause of the problem, we found that similar problems have been reported in the native Windows environment with another application (Explorer.exe).[24]

If we could pre-load these five libraries when *Wine* starts the application, the problem might go away. The users will take a relatively insignificant performance hit at startup, instead of taking the same hit at run time, when it is highly visible. To test this idea, we modified *Wine*'s source to perform the pre-loading and the problem was eliminated, reducing the time it takes to display the menu from several seconds to a fraction of a second.

Although *spTracer* was able to identify the cause of the performance problem, the study revealed a limitation of our current prototype. Our code analysis tool was unable to discover parts of code in the *Winword* executable and native Windows DLL libraries. As a result, our tracer did not follow some call chains, producing blind spots in the trace. A close examination of the trace revealed that the call chain leading to `wine_dlopen` passed through such a blind spot. Fortunately, functions after the blind spot on that chain were already instrumented at that time, because Wine executed them when loading several shared objects at startup. This fact allowed *spTracer* to receive control again, notice the calls to `wine_dlopen`, and identify the cause of the performance problem.

Discovering all code with offline analysis is not always possible.[29] Furthermore, offline tools cannot analyze dynamically-generated code. We plan to address these limitations with a hybrid approach that will combine our offline code analysis with on-the-fly capabilities similar to those of *Dynamo*,[4] *DIOTA*,[22] and *PIN*.[13] By maintaining a repository of code analysis results for known applications, we will retain the low-overhead feature of offline code analysis. Only when an unknown code fragment is discovered, our tracer will parse it on the

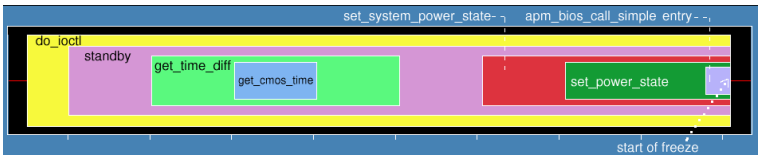

Figure 6. A trace of APM events right before the freeze in Kernel 2.4.26

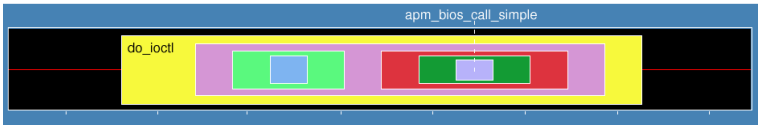

Figure 7. A normal sequence of standby APM events in Kernel 2.4.20

fly and add the results to the persistent repository. A similar approach has proved viable for dynamic binary translation in *FX!32*, where the repository was used for keeping translated code.[11]

5.3. Debugging a laptop suspend problem

Locating the cause of a problem in an unfamiliar piece of kernel code may be a challenging task. Here, we describe how self-propelled instrumentation helped us investigate a power-management problem in an IBM Thinkpad T41 laptop computer running Linux. When left idle for a certain amount of time, the laptop at question would go into a power-saving standby mode. However, it could not wake up to resume normal execution after standby—the screen remained blank, it did not respond to keyboard or mouse events, and did not accept network connections. The freezes happened with two recent Linux kernels that we tried (2.4.26 and 2.6.6). However, the original kernel that was pre-installed with the RedHat 9 distribution (2.4.20) did not exhibit this behavior.

To locate the cause of the problem, we used self-propelled instrumentation to trace kernel events that happen when the system transitions into the standby state. To initiate tracing, our tool sends a `KTRACE_START ioctl` request to */dev/ktracedrv* and then executes the *"apm -S"* command to initiate standby. The system starts generating the trace, runs for less than one second, and then locks up. Fetching the trace from memory of the frozen system may not be possible. Therefore, we resorted to printing trace records to a serial connection as they were being generated. (The T41 laptop does not have a serial connector, but its infrared port can emulate a serial port and connect to a similarly-configured machine on the other end).

Although the approach proved to work well for short traces, sending the complete record of all kernel function calls over a narrow 115-Kbit serial connection introduced unbearably high overhead. The system was unable to reach the standby state in 20 minutes, while without tracing it takes less than 1 second. To reduce the amount of trace data that needs to be sent over the serial connection, we restricted the scope of tracing to the *APM* power-management code in the Linux kernel. Instead of instrumenting the standard entries for hardware interrupts and system calls, we injected initial instrumentation at entry points of the *APM* device driver, */dev/apm_bios*, and did not trace functions that do not belong to the address range of the driver.

Figure 6 shows the trace of APM-related events immediately preceding the freeze. Apparently, the system locks up in the `apm_bios_call_simple` function—the call to `apm_bios_call_simple` never returns. By looking at the source code for `apm_bios_call_simple`, we confirmed that it invokes APM routines in the BIOS firmware. The chain of calls indicates that the driver was servicing the `STANDBY ioctl` request from a user-level program. Compare it to Figure 7, that shows the record of APM events in the 2.4.20 kernel, which does not freeze. We see an almost-identical trace of calls, but `apm_bios_call_simple` returns normally.

While the system executes APM BIOS calls, the interrupts are typically disabled. This fact may explain why the system does not respond to keyboard or mouse wake up events. To confirm that interrupts are not being serviced after the freeze, we adjusted our tracing approach to inject instrumentation into the `do_IRQ` hardware interrupt entry point after the flow of control reaches the `standby` function. The new traces showed no interrupt activity after the call to `apm_bios_call_simple`.

	Warm-up	Steady-state	Inactive
Pre-instr	13%	29%	29%
DIOTA	95%	58%	58%
Self-prop	14%	34%	0%

(a) Compiled with gcc-3.2.2

	Warm-up	Steady-state	Inactive
Pre-instr	10%	10%	10%
DIOTA	89%	23%	23%
Self-prop	11%	15%	0%

(b) Compiled with gcc-2.95.3

Table 1. Slowdown in *MPlayer*

To verify whether the interrupts are not being serviced because they are disabled, we made the system generate periodic Non-Maskable Interrupts (NMI) by booting the kernel with the `nmi_watchdog=2` flag.[30] NMI interrupts are not affected by `cli` instructions and should be delivered even if other interrupts are disabled. To our surprise, even NMI interrupts are no longer generated when `apm_bios_call_simple` is entered. This behavior cannot be caused by disabling the maskable interrupts, and it made us suspect that the interrupt controller stopped functioning properly after the `STANDBY` call in `apm_bios_call_simple`.

Recent x86 systems are typically equipped with two different interrupt controllers. SMP machines require the use of APIC (Advanced Programmable Interrupt Controller), while uniprocessors can use either the new APIC or the original PIC (Programmable Interrupt Controller). APIC provides several new useful features, while hardware and software support for PIC appears to be better tested. To verify whether our problem is an APIC-stability issue, we recompiled the 2.4.26 kernel to use the standard PIC. With the new kernel, the system was able to enter the `STANDBY` mode and wake up on user activity properly.

The same problem might be found with traditional methods. One could step through execution of *"apm -S"* with an application debugger to identify the problem `ioctl` call and then step through `do_ioctl` in the *APM* driver with a kernel debugger. Further hypotheses could be tested by instrumenting the kernel source code and sending traces through the serial connection. However, the flexibility and dynamic nature of self-propelled instrumentation allowed us to find this problem without modifying the kernel.

5.4. Overhead study

To estimate the run-time overhead of self-propelled tracing, we measure the slowdown that it introduced to *MPlayer* in the course of DVD playback on Thinkpad T41. To measure the slowdown, we modified the *MPlayer* source code to count the number of cycles spent processing each video frame. Note that to show frames at a regular interval (1/25th of a second for our video clip), *MPlayer* inserts a calculated delay after decoding a frame, but before showing it on the screen. The adjustable nature of the delay may mask the overhead that we are measuring. Therefore, we exclude the time spent sleeping from the total time of processing a frame.

We compare self-propelled instrumentation to two other approaches: pre-instrumentation and *DIOTA* (Version 0.9). To perform pre-instrumentation, we use the self-propelled framework, but insert trace statements into all functions at startup, before starting measurements, and disable propagation of instrumentation. With pre-instrumentation and self-propelled instrumentation, we call empty trace routines on entry and exit from every function in the application. Since *DIOTA* does not currently support function-level tracing, we measure the lower bound of its overhead—the overhead of following the control flow without executing any trace calls.

To model trace collection for a short interval during execution, we measure the overhead at three stages: warm-up (the first frame after enabling tracing), steady-state (after being enabled for a while), and inactive (after disabling or before enabling tracing). The overhead for all approaches proved to depend on the compiler used for building *MPlayer*. In particular, the overhead varied greatly depending on how many functions a compiler decided to inline. Table 1(a) and Table 1(b) present the highest (for *MPlayer* compiled with gcc-3.2.2) and the lowest (for *MPlayer* compiled with gcc-2.95.3) overhead results respectively. Further discussion applies to both runs, as all instrumentation methods show similar relative performance.

The pre-instrumentation approach demonstrates reasonable performance at the warm-up and steady-state stages. (Notice that warm-up overhead of pre-instrumentation and self-propelled instrumentation is often lower than steady-state. Even without tracing, the first frame takes longer to fetch and decode than the rest of the

frames, which lowers the warm-up overhead of all instrumentation methods). However, pre-instrumentation cannot be rapidly enabled on the fly—even the same-address space instrumentation of the self-propelled framework requires approximately 45 milliseconds to instrument more than 11000 functions in *MPlayer* and supporting libraries. This time exceeds the 40-millisecond interval allotted to displaying a video frame. For larger applications, the delay will be even more pronounced. An alternative approach is to pre-instrument every function on startup, but it will force the user to endure the steady-state overhead for the entire running time of the system.

The current version of *DIOTA* does not support on-demand activation either, although it could be implemented. Therefore, its non-zero overhead at the inactive stage is less problematic. More importantly, *DIOTA* has significant warm-up overhead, resulting in dropped frames on startup even in the lower overhead run. We believe that building the code cache on the fly may be unacceptable for analysis of latency-sensitive applications. It may introduce intermittent delays upon run-time activation of the tracer and also whenever the flow of control reaches previously-unseen code. In contrast, the warm-up overhead of self-propelled instrumentation is significantly lower than that of *DIOTA*, since we perform code analysis in advance and do not need to populate the code cache. We also impose no overhead at the inactive stage, as tracing can be rapidly activated on demand.

To measure the upper bound of steady-state overhead of self-propelled instrumentation, we traced a microbenchmark that was invoking an empty function in a tight loop. With self-propelled instrumentation, the benchmark ran 500% slower. Such pathological cases are rare in real applications, but they need to be addressed. We plan to design an adaptive scheme that would detect short frequently-executed functions and avoid tracing them to lower the run-time overhead. The efficacy of such a scheme is yet to be studied.

6. CONCLUSION

We presented a novel approach for detailed performance analysis and system introspection. The key features of the approach are: ability to analyze black-box systems with little human help, high level of detail, and low overhead. Our enabling technology is the ability to perform analysis by following the flow of control. To follow the control flow efficiently, we use in-situ instrumentation and perform code analysis in advance. The approach proved feasible and useful for performance analysis and debugging of real-world interactive systems. Its ability to cross the kernel boundary is a foundation for future full-system analysis.

ACKNOWLEDGMENTS

We wish to thank Victor Zandy, Eli Collins, Philip Roth, and the anonymous reviewers for helpful feedback on revisions of the paper. The Dyninst expertise of Laune Harris was invaluable in debugging our code analysis tool.

REFERENCES

1. W. Akkerman, *strace home page*, http://www.liacs.nl/∼wichert/strace/
2. L. Albertsson, "Temporal Debugging and Profiling of Multimedia Applications", *Multimedia Computing and Networking*, San Jose, CA, Jan. 2002.
3. J.M. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.A. Leung, R.L. Sites, M.T. Vandevoorde, C.A. Waldspurger, and W.E. Weihl, "Continuous Profiling: Where Have All the Cycles Gone?", *ACM Transactions on Computer Systems*, **15**, 4, Nov. 1997, pp. 357–390.
4. V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A Transparent Dynamic Optimization System", *Conference on Programming Language Design and Implementation*, Vancouver, British Columbia, June 2000.
5. P. Barham, R. Isaacs, R. Mortier, D. Narayanan, "Magpie: real-time modelling and performance-aware systems", *9th Workshop on Hot Topics in Operating Systems*, Lihue, Hawaii, May 2003.
6. D. Bruening, E. Duesterwald, and S. Amarasinghe, "Design and Implementation of a Dynamic Optimization Framework for Windows", *4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, Austin, TX, Dec. 2001.
7. B. Buck and J.K. Hollingsworth, "An API for runtime code patching", *Journal of High Performance Computing Applications*, **14**, 4, pp. 317–329, Winter 2000.
8. B.M. Cantrill, M.W. Shapiro, and A.H. Leventhal, "Dynamic Instrumentation of Production Systems", *USENIX Annual Technical Conference*, Boston, June 2004
9. J. Cespedes, *LTrace home page*, http://www.cespedes.org/software/ltrace/

10. M. Chen, E. Kiciman, E. Fratkin, E. Brewer, and A. Fox, "Pinpoint: Problem Determination in Large, Dynamic, Internet Services", *International Conference on Dependable Systems and Networks*, Washington D.C., June 2002.
11. A. Chernoff and R. Hookway, "DIGITAL FX!32 Running 32-Bit x86 Applications on Alpha NT", *USENIX Windows NT Workshop*, Seattle, Washington, Aug. 1997.
12. W.E. Cohen, "Multiple Architecture Characterization of the Linux Build Process with OProfile", http://people.redhat.com/wcohen/wwc2003/
13. R. Cohn and R. Muth, "Pin User Guide", http://rogue.colorado.edu/Pin/documentation.php
14. CyberLink Corporation, "CyberLink PowerDVD 5 System Requirements", http://www.gocyberlink.com/english/products/powerdvd/system_requirements.jsp
15. Y. Endo, M. Seltzer, "Improving Interactive Systems Using TIPME", *ACM SIGMETRICS Performance Evaluation Review*, **28**, 1, June 2000, pp. 240–251.
16. S. Graham, P. Kessler, and M. McKusick, "gprof: A Call Graph Execution Profiler", *SIGPLAN '82 Symposium on Compiler Construction*, Boston, June 1982, pp. 120–126.
17. V. Herrarte, E. Lusk, "Studying parallel program behaviour with upshot", *Technical Report ANL91/15*, Argonne National Lab, 1991.
18. J.K. Hollingsworth, M. Altinel, "Dyner User's Guide", http://www.dyninst.org/docs/dynerGuide.v40.pdf
19. J. Houston, "Kernel Trace Mechanism for KDB", http://www.ussg.iu.edu/hypermail/linux/kernel/0201.3/0888.html
20. "KerninstAPI Programmer's Guide", http://www.paradyn.org/kerninst/release-2.0/kapiProgGuide.html
21. K.A. Lindlan, J. Cuny, A.D. Malony, S. Shende, B. Mohr, R. Rivenburgh, C. Rasmussen, "Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates", *SC 2000: High Performance Networking and Computing Conference*, Dallas, TX, Nov. 2000.
22. J. Maebe, M. Ronsse, K. De Bosschere, "DIOTA: Dynamic Instrumentation, Optimization and Transformation of Applications", *WBT-2002: Workshop on Binary Translation*, Charlottesville, Virginia, September 2002.
23. P.S. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenstrom, and B. Werner, "SimICS/sun4m: A Virtual Workstation", *USENIX Annual Technical Conference*, New Orleans, June 1998.
24. Microsoft Knowledge Base, "Performance Issues When Loading Winmm.dll", http://support.microsoft.com/default.aspx?scid=kb;en-us;266327
25. B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall, "The Paradyn Parallel Performance Measurement Tool", *IEEE Computer*, **28**, 11, Nov. 1995, pp. 37-46.
26. A.V. Mirgorodskiy and B.P. Miller, "CrossWalk: A Tool for Performance Profiling Across the User-Kernel Boundary", *International Conference on Parallel Computing*, Dresden, Germany, Sept. 2003.
27. *MPlayer home page*, http://www.mplayerhq.hu/
28. "Playing DVD and videos", http://www.linuxquestions.org/questions/answers/28
29. M. Prasad and T. Chiueh, "A Binary Rewriting Defense against Stack-based Buffer Overflow Attacks", *USENIX 2003 Annual Technical Conference*, San Antonio, TX, June 2003.
30. "Red Hat Cluster Suite Configuring and Managing a Cluster", http://www.redhat.com/docs/manuals/enterprise/RHEL-3-Manual/cluster-suite/ap-hwinfo.html
31. W. Robertson, *itrace home page*, http://www.cs.ucsb.edu/∼wkr/projects/itrace/
32. M. Rosenblum, E. Bugnion, S. Devine, and S.A. Herrod, "Using the SimOS machine simulator to study complex computer systems", *ACM Transactions on Modeling and Computer Simulation*, **7**, 1, Jan. 1997, pp. 78–103.
33. B. Schendel, "TraceTool: A Simple Dyninst Tracing Tool", http://www.paradyn.org/tracetool.html
34. A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools", *Conference on Programming Language Design and Implementation*, Orlando, Florida, June 1994.
35. Sun Microsystems, "Truss: trace system calls and signals", *Solaris 9 Reference Manual Collection*, http://docs.sun.com
36. A. Tamches and B.P. Miller, "Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels", *3rd Symposium on Operating Systems Design and Implementation*, New Orleans, Feb. 1999.
37. *Wine home page*, http://www.winehq.org
38. V. Zandy, "Force a Process to Load a Shared Library", http://www.cs.wisc.edu/∼zandy/p/hijack.c