

Dynamic Program Analysis of Microsoft Windows Applications

Alex Skaletsky, Tevi Devor, Nadav Chachmon, Robert Cohn, Kim Hazelwood[†], Vladimir Vladimirov, Moshe Bach
Intel Corporation [†]University of Virginia

Abstract—Software instrumentation is a powerful and flexible technique for analyzing the dynamic behavior of programs. By inserting extra code in an application, it is possible to study the performance and correctness of programs and systems. Pin is a software system that performs run-time binary instrumentation of unmodified applications. Pin provides an API for writing custom instrumentation, enabling its use in a wide variety of performance analysis tasks such as workload characterization, program tracing, cache modeling, and simulation. Most of the prior work on instrumentation systems has focused on executing Unix applications, despite the ubiquity and importance of Windows applications. This paper identifies the Windows-specific obstacles for implementing a process-level instrumentation system, describes a comprehensive, robust solution, and discusses some of the alternatives. The challenges lie in managing the kernel/application transitions, injecting the runtime agent into the process, and isolating the instrumentation from the application. We examine Pin’s overhead on typical Windows applications being instrumented with simple tools up to commercial program analysis products. The biggest factor affecting performance is the type of analysis performed by the tool. While the proprietary nature of Windows makes measurement and analysis difficult, Pin opens the door to understanding program behavior.

I. INTRODUCTION

Analyzing the dynamic behavior, performance, and correctness of software and systems is invaluable to software developers and hardware designers. Software instrumentation [8], [9], [14], [25] achieves this goal by inserting extra code in an application to observe its run-time behavior. Instrumentation tools are especially useful for analyzing the behavior of programs running on Windows because the source code is typically not available and the kernel interface cannot be adapted to support observability. Whole system simulators [4] are operating system independent and can provide observability for Windows execution, but it is more difficult to attribute the analysis to processes, procedures, and shared libraries because the instrumentation system runs beneath the operating system. The ability to instrument unmodified binaries at the process level facilitates analyzing the performance of proprietary Windows applications in realistic scenarios.

Pin is a software system that performs run-time binary instrumentation of Linux and Windows applications. Pin provides an API for writing custom instrumentation, enabling its use in a wide variety of performance analysis tasks such as workload characterization, program tracing, cache modeling, and simulation [11], [15], [18], [19], [26]. Pin is the underlying infrastructure for commercial products like the Intel[®] Parallel Studio suite of performance analysis tools. Intel makes Pin available free of charge at <http://www.pintool.org>. With over 30,000 downloads and 500 citations in academic papers, Pin has a large community developing instrumentation tools for research, education, and product development.

Pin instruments execution by running unmodified applications in a process-level virtual machine. Pin intercepts execution at the beginning of a process and injects a runtime agent that is similar to a dynamic binary translator. Instrumentation is inserted while the program is running, and consists of calls to functions written in C++.

Pin can be used to observe low level events like memory references, instruction execution, and control flow. Higher level abstractions such as procedure invocations, shared library loading, thread creation, and system call execution can also be observed.

This paper identifies the Windows-specific obstacles for implementing a process-level instrumentation system, describes a comprehensive solution, discusses alternatives, and evaluates the performance of the system itself. It highlights the issues specific to providing programmable instrumentation including observability, control, and isolation.

First and foremost, Pin is a system for instrumentation and must provide a high degree of *observability*. Instrumentation must be notified that new threads are created or shared libraries are loaded before the first instruction executes. It should receive a similar notification when the process, thread, or library ends. The instrumentation system must find the proper time to deliver such notifications, potentially waiting for additional system state to be initialized.

Second, Pin must *control* execution of the process. Control is maintained by copying the application code into a software code cache and rewriting branch instructions so control stays in the copied code. As the program executes from the code cache, Pin can instrument the application. The challenges in maintaining control occur in getting initial control of the process (called *injection*) and maintaining control during transitions between the application and kernel. To provide the maximum observability, Pin must inject itself into a new process as early as possible. Pin depends on some basic Windows services so it is preferable from an implementation perspective to delay injection until various services are initialized. However, late injection provides less observability. Compared to other systems, Pin minimizes its dependence on Windows system services that need initialization and gets control immediately after they are initialized. As a process-level virtual machine, Pin loses control when the kernel executes and must regain control when the kernel resumes executing the application. However, the kernel interface is not designed to have an independent agent interposed between the kernel and application and the challenge is to design a robust method for managing the transitions.

Third, the instrumentation must be *isolated* from the application execution. Instrumentation executes in the same process as the application it is observing. If Pin overwrites memory or registers used by the application, then the application could behave differently or incorrectly. Simple isolation mechanisms place restrictions on the code an instrumentation tool developer can insert or incur a high overhead by saving and restoring all the state before switching to instrumentation. Pin allows the instrumentation tool writer to use a conventional C++ programming environment with minimal restrictions, and uses optimization to make it efficient.

Achieving each of these goals in a robust manner is non-trivial, but the final system is invaluable for analyzing the performance and correctness of modern programs. Prior literature has described various internal design decisions within the Pin engine, however, all such work has focused exclusively on the Linux implementation. By presenting the internal design and applications of a program analysis system for Windows, this paper fills a current void in the literature.

```

1 #include <stdio.h>
2 #include "pin.H"
3 FILE * trace;
4
5 VOID Address(VOID * addr) {
6     fprintf(trace, "%p\n", addr);
7 }
8 VOID Instruction(INS ins, VOID *v) {
9     if (INS_IsMemoryRead(ins)) {
10         INS_InsertCall(
11             ins, IPOINT_BEFORE, AFUNPTR(Address),
12             IARG_MEMORYREAD_EA, IARG_END);
13     }
14     if (INS_IsMemoryWrite(ins)) {
15         INS_InsertCall(
16             ins, IPOINT_BEFORE, AFUNPTR(Address),
17             IARG_MEMORYWRITE_EA, IARG_END);
18     }
19 }
20 VOID Fini(INT32 code, VOID *v) { fclose(trace); }
21
22 int main(int argc, char *argv[]) {
23     PIN_Init(argc, argv);
24     trace = fopen("pinatrace.out", "w");
25     INS_AddInstrumentFunction(Instruction, 0);
26     PIN_AddFiniFunction(Fini, 0);
27     PIN_StartProgram();
28     return 0;
29 }

```

Listing 1: Tool for printing addresses of all program memory reads/writes.

The paper begins with an introduction to Pin’s instrumentation and software architecture. Section III then describes how to inject the Pin runtime agent into an application process. Sections IV, V, and VI describe the problems and solutions for executing system calls, handling exceptions, and executing multithreaded applications, respectively. A description Pin’s runtime support appears in Section VII. Finally, Section VIII describes some program analysis tools that use Pin, and Section IX presents a performance analysis.

II. PIN OVERVIEW

In this section, we provide an overview of Pin. We first provide a brief introduction to writing instrumentation tools and then follow with high-level description of the software architecture. For more specific details, the interested reader can refer to prior work [14] or <http://www.pintool.org/learn.html>.

Programmable Instrumentation. Pin provides user-programmable instrumentation; a developer writes a pintool in C++ using the Pin API. The pintool describes where to insert instrumentation and what it should do. A pintool consists of instrumentation, analysis, and callback routines [24]. *Instrumentation* routines walk over the instructions of an application and insert calls to *analysis* routines. *Analysis* routines are called when the program executes an instrumented instruction, collecting data about the instruction or analyzing its behavior. *Callbacks* are invoked when an event occurs, for example when the program is about to exit.

Listing 1 shows a simple tool that prints out the memory addresses of all data that is read or written by a program. The instrumentation routine named `Instruction()` is called by Pin the first time an instruction is executed, so that the routine may specify how it should be instrumented. If the instruction reads or writes memory, this example inserts a call to the analysis routine named `Address()`, and directs Pin to pass it the virtual address of the memory reference. `Address()` is called immediately before an instrumented instruc-

```

1 typedef void (*malloc_funptr_t)(size_t size);
2 malloc_funptr_t app_malloc;
3
4 void * malloc_wrap(size_t size) {
5     void * ptr = app_malloc(size);
6     printf("Malloc %d return %p", size, ptr);
7     return ptr;
8 }
9
10 void Image(IMG img, void *v) {
11     RTN mallocRtn = RTN_FindByName(img, "malloc");
12     if (RTN_Valid(mallocRtn)) {
13         app_malloc = (malloc_funptr_t)RTN_ReplaceProbed(
14             mallocRtn, AFUNPTR(malloc_wrap));
15     }
16 }

```

Listing 2: Wrapping malloc using probe-based instrumentation.

tion executes, and prints the virtual address of the memory reference to a file. `Fini()` is a callback routine that is invoked when the program exits. Instrumentation and callback routines are registered on lines 25 and 26 in the `main()` function of the pintool.

Listing 1 demonstrates only a small part of the Pin API. While the example uses an instrumentation routine that can only see a single instruction at a time, Pin allows instrumentation routines to see blocks of instructions or whole binaries. The only argument passed to `Address()` is the virtual address; possible arguments include register contents (e.g. value of R9), the program counter (PC), procedure argument values, constants, etc. This example only uses the program end callback, but Pin provides a wide variety of callbacks to notify a tool about shared library loading, thread creation, system calls, Unix signals, and Microsoft Windows exceptions.

Pin uses a just-in-time (JIT) compiler to insert instrumentation into a running application. The JIT compiler recompiles and instruments small chunks of binary instructions immediately prior to executing them. Pin stores the modified instructions in a software code cache where they are executed in lieu of the original application instructions. The code cache allows code regions to be generated once and then reused for the remainder of program execution, amortizing the costs of compilation. Compilation introduces overhead that is highly dependent on the application and workload; SPEC integer benchmarks have an average slowdown of 50% [14].

Pin has a high-performance option called *probe mode* where the base overhead is near zero because the program binary is modified in place, rather than translated and cached. Probe mode is a subset of the Pin API; a limited set of callbacks are available and instrumentation is limited to interposing wrapper routines for global functions. Listing 2 shows a fragment of a probe-based tool that wraps calls to `malloc()` so that it can print the argument and return values. `Image()` is an instrumentation routine that is invoked every time a program binary or shared library is loaded. It searches the binary for a function called `malloc()` and replaces it with a call to `malloc_wrap()`. When the program calls `malloc()`, `malloc_wrap()` is called instead, which calls the application `malloc()` and then prints the argument and return value. To avoid infinite recursion the call to `malloc()` from `malloc_wrap()` should not be redirected. `RTN_ReplaceProbed()` returns a function pointer that invokes the replaced routine without redirection, and `malloc_wrap()` must invoke it instead of the original `malloc()` entry point.

In probe mode, the program binary is modified in memory. Pin overwrites the entry point of procedures with jumps (called probes)

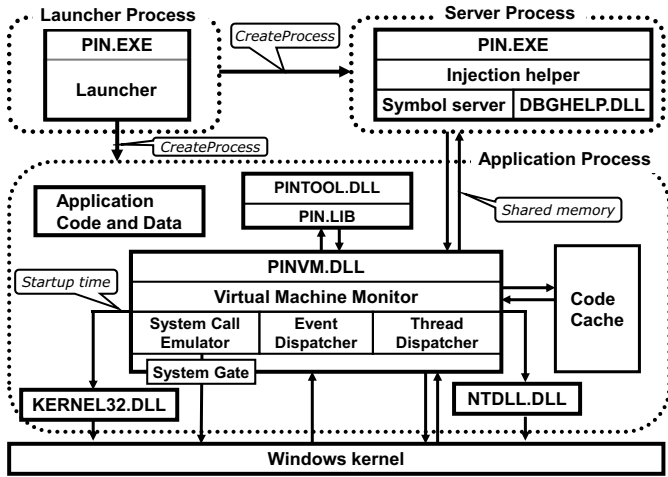


Fig. 1: The Windows Pin Architecture. The launcher, server, and application run as separate processes. The application process contains the Pin VMM and the code cache of instrumented instructions.

to dynamically-generated instrumentation [12]. This code can invoke analysis routines or a replacement routine. When the replacement routine needs to invoke the original function, it calls a copy of the entry point (without the probe) and continues executing the original program. By avoiding JIT compilation, probe mode achieves better performance. However, it provides less observability and is only suitable for instrumentation where procedure granularity is sufficient.

Software Architecture Overview. Figure 1 illustrates the Pin software architecture for Windows. At the highest level, the Pin Windows framework consists of the following processes:

- 1) The *launcher* creates an application process, injects Pin components, and waits for the process to terminate.
- 2) The *server* provides services that cannot be performed by the instrumented process, such as managing symbol table information or injecting Pin. The server communicates with the instrumented process via shared memory.
- 3) The instrumented guest *application*.

An instrumented application process contains the Pin DLL (`pinvm.dll`), the user-defined `pintool` DLL, and the application executable and DLLs. `pinvm.dll` consists of a virtual machine monitor (VMM), a code cache, and an instrumentation API invoked by the `pintool`. The Pin VMM is the engine that controls and instruments the application. It consists of a just-in-time (JIT) compiler, a dispatcher, and OS interaction units. Pin executes in user level ring 3, and can therefore only capture user-level code. After Pin gains control of the guest application, the VMM coordinates execution of the application. The JIT compiler generates and instruments application code, which is then launched by the dispatcher. The compiled code is stored in the code cache. The `pintool` is a plug-in that contains the instrumentation and analysis routines, and is linked with the Pin library that allows it to communicate with the Pin VMM.

III. INJECTION

Injection is the procedure for loading the Pin VMM into the address space of an application. Injection allows Pin to gain control of execution and then instrument and execute the application. It is desirable to perform injection as early as possible so that tools can observe the execution of all guest application instructions, but it is not possible to load `pinvm.dll` until the loader has initialized. The

challenge is to get control reliably at the earliest point where it is safe to load a DLL.

Pin creates the application process in suspended state and attaches to it using Win32 debugger APIs. The debugger waits for `kernel32.dll` to finish initializing and then detaches from the debugged process. Pin saves the application’s context and changes the instruction pointer to a small boot routine that is copied to target process memory. The boot routine loads `pinvm.dll` and calls its main procedure. `pinvm.dll` initializes itself and loads the tool DLL. Using this method, tools miss the opportunity to instrument the initialization of the `ntdll.dll`, `kernel32.dll`, and `kernelbase.dll` system libraries, but see every executed instruction of the program binary and DLLs. We do not believe it is possible to load `pinvm.dll` earlier.

Compared to other systems, our method provides earlier and more reliable injection. `DynamoRIO` [7] uses a Windows registry key that causes `user32.dll` to load DLLs listed in the key. However, `user32.dll` is loaded later than Pin’s injection point and therefore misses more of the program execution. `DynamoRIO`’s method fails for programs that do not load `user32.dll`, therefore their backup method is to hook the entry point of the application; But this is also late and we have found that some applications overwrite the entry point at runtime.

When an instrumented application creates a child process, it is desirable to instrument the child as well. On 64-bit Windows, processes can be either 32-bit or 64-bit, and Pin can instrument child processes for all combinations of 32/64-bit parent/child. Injecting into a child process is slightly different than the initial injection into the parent because the application creates the child process, not Pin. Creating a child usually requires a sequence of three system calls: `NtCreateProcess`, `NtCreateThread`, and `NtResumeThread`. Immediately before `NtResumeThread`, Pin alters the child process context so the Pin boot routine is executed instead of the application code. `NtResumeThread` can be called in situations other than creating a child process, and Pin must be able to recognize when it is being used to start a process. Altering the child’s context during `NtResumeThread` is more transparent than altering it during `NtCreateThread` (as is done by `DynamoRIO` [7]). Furthermore, we encountered applications that create child processes in a suspended state and query the context before resuming. Such applications would notice an unexpected context in the child and would fail if we altered the context in `NtCreateThread`.

IV. SYSTEM CALLS

System calls are instructions that transition from executing an application in user mode to executing the kernel in system mode. They are used to request system services like file I/O and process creation. As a process-level VM, Pin has full control of everything that executes in user mode, but loses control in kernel mode. Pin must manage the execution of system calls to ensure that it regains control when the system resumes execution in user mode.

There are three steps to managing system calls. First, Pin must detect when the application is about to execute a system call, and instead redirect control to the VMM. Second, Pin must be able to execute the system call on behalf of the application. Third, Pin must be able to regain control after an interrupted system call. When a system call is interrupted, the kernel may cause the application to continue execution elsewhere. Pin must intercept execution after an interrupted system call so it can direct control back to its code cache. At all points of executing a system call, Pin must be able to construct a precise state reflecting the value of application registers without the effects of instrumentation.

TABLE I: Pin monitored system calls

System Call Name	Pin's Action
NtAllocateVirtualMemory NtFreeVirtualMemory NtProtectVirtualMemory	Update internal database that keeps protection attributes for application code pages, which is essential for handling self-modified code in the application.
NtMapViewOfSection NtUnmapViewOfSection	Update internal database of modules mapped into the address space of the process. Notify pintool about module load/unload.
NtCreateProcess/Ex NtCreateUserProcess NtCreateThread/Ex NtResumeThread	Intercept creation of a new child process. Notify pintool and inject Pin into the child process, if requested.
NtSuspendThread	Suspend the target thread at a safe point (see Section VI-D).
NtGetContextThread NtSetContextThread	Retrieve/modify original context associated with the safe suspension point in the target thread (see Section VI-D).
NtContinue	Change the target context to keep control in Pin. Restore original context on return from the system call.
NtCallbackReturn	Recover original context of the system call that was interrupted by the callback.
NtTerminateThread	Notify pintool about the thread exit and destroy thread-local objects in Pin.
NtTerminateProcess	Notify pintool about the process exit and shutdown Pin VMM.

A. System Call Interception

Pin needs to intercept system call instructions at a point where it can capture the arguments and system call number, and then transfer control to the VMM. Pin translates and executes all user-mode instructions from its software code cache, making it logical to intercept system calls at the instruction that transfers control from user mode to the kernel. Windows uses `sysenter` or `int 2e` on 32-bit systems and `syscall` for 64-bit processes. 32-bit (WOW64) processes on 64-bit Windows use `jmp far` instruction to enter kernel through the WOW64 layer. This instruction is located at a well-known address that is stored in the Thread Environment Block (TEB) of each thread, so it can be easily recognized. When Pin translates a trace, it detects system calls by checking type and address for each instruction in the trace. Ideally, the trace is translated only once, so the cost of the system call detection is low relative to the overhead for leaving the code cache and entering the Pin VMM each time the system call is executed.

As the system call interface is not published by Microsoft, we considered intercepting at the published Win32 API. However, intercepting Win32 is problematic [7] since the mapping of system operations to the Win32 API is not one-to-one. Some system calls (e.g. `NtContinue`, `NtCallbackReturn`) are not explicitly represented in the Win32 API, while other Win32 API functions aggregate several system calls in a single function. For example, the `LoadLibrary` API can map multiple modules in a single function call, but Pin needs to handle each module separately.

B. System Call Execution

After interception, Pin must execute the system call for the application. Some system calls must be monitored because they may affect Pin's internal state. For example, when the application creates or terminates a thread, Pin needs to initialize or destroy its own thread-local objects. The rest of the system calls have unknown arguments and side effects and must be invoked in a way that preserves the original semantics. Table I shows the most prevalent Pin-monitored system calls and the corresponding procedures performed by Pin before and after system call execution. We adhere to the descriptions of the API functions described in the Windows Native API Manual [16].

When a system call is executed, the system call number is in register `rax`. Pin uses this number to determine whether special handling is necessary. Windows system call numbers are unpublished and potentially change with each system build and therefore cannot be built into Pin. Instead, Pin automatically extracts the system

call number from the `ntdll.dll` system library at runtime. Every Pin monitored system call has a wrapper function in the `ntdll.dll` system library that loads the system call number into a register and invokes the system call. At initialization time, Pin uses the debugger APIs to single step a dummy process through the wrapper functions for the Pin monitored system calls. When it reaches the system call instruction, it records the system call number and location of arguments relative to the top of stack. Since there are fewer than 30 Pin-monitored system calls, the performance impact of dynamic discovery is insignificant.

We considered several alternatives to dynamic discovery. `DynamoRIO` uses statically prepared tables for supported versions of Windows [7]. However, Pin must support unreleased versions and it would be cumbersome to have to manually update the tables for each one. We also investigated extracting system call numbers from `ntdll.dll` by using static analysis of the instructions but concluded that it would be less robust because it makes more assumptions about code patterns of the system call wrappers.

The remaining, unmonitored system calls just need to be executed. We make the following conservative assumptions to be less dependent on the undocumented APIs and changes in future Windows versions:

- Any register, except the instruction pointer (IP), may be used as an input, output, or scratch register of the system call.
- System calls may block (suspend) the calling thread for an indefinite time interval.
- The system may interrupt system call execution by asynchronous calls to application procedures. In reality, only a few system calls are interruptible, but the list is not published.
- Any system call may cause an exception.

Figure 2 shows the flow for executing unmonitored system calls. The JIT compiler emits a jump to the `System Call Emulator` in the VMM at the point where the system call appears in the original trace. The jump instruction transfers control to the VMM along with the type of the original system call instruction and the original context. The VMM uses the type information to avoid additional instruction decoding that would be necessary because the system calling convention depends on the system call instruction (e.g. `int 2e` vs. `sysenter`). The `System Call Emulator` checks the system call number (in the context register) and identifies the appropriate handling procedure. If the system call is a Pin-monitored and non-interruptible system call, the emulator executes it and performs all necessary actions before and after the execution. Otherwise, if the system call is unknown or interruptible, it is forwarded to the

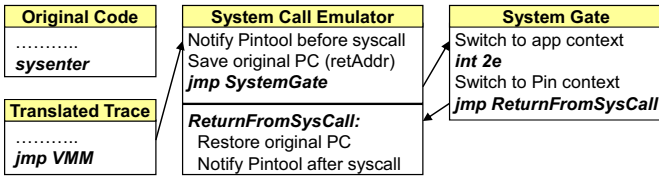


Fig. 2: System call execution flow.

System Gate – a VMM-generated procedure that executes system calls in their original context (except for the PC) and keeps control in Pin upon return. Executing all interruptible system calls in the System Gate guarantees that all asynchronous callbacks are invoked on the application’s stack. This is important, because doing otherwise may cause recursive VMM entries and Pin stack overflow when the system delivers nested asynchronous callbacks. Due to the transparency requirements, the System Gate procedure must fill and spill original context around system calls without touching the application stack. This implies use of absolute addressing when accessing the thread-local state and as a result, unique encoding of the system gate procedure for each thread. For this reason, Pin implements the system gate as a thread-private procedure that is generated from a template the first time the thread enters the VMM.

Our measurements (see Table IV) show that leaving the code cache and entering the Pin VMM for each system call is the major contributor to the overhead of the system call emulation. As an alternative, we considered executing system call instructions in Pin’s code cache, just like other application instructions. This method would be difficult because the JIT compiler steals a register and uses it to address thread-local storage. We wanted all registers to hold their original value during the system call. In addition, a system call can block indefinitely, making it difficult to reuse memory in the code cache containing blocked system call instructions.

32-bit applications can use either the `int 2e` or `sysenter` instruction to invoke a system call, but the system gate always uses `int 2e`. The difference is that `int 2e` always resumes executing at the next instruction, while `sysenter` resumes executing at a fixed address in `ntdll.dll`. Regaining control after `sysenter` requires placing a trampoline at the resume point in `ntdll.dll` or replacing the application return address [7]. As long as Windows supports `int 2e` system calls, our approach seems to be easier and less likely to be detected by the application.

As mentioned earlier, Pin executes system calls in their original context. Pin saves the original return address of the system call in a thread-local variable while executing the system call so it can resume the application at the proper place. However, the `NtContinue` system call does not return to the caller; it switches to the context provided as a system call argument. To regain control on return from `NtContinue`, Pin saves the target context’s instruction pointer and replaces it with the address of a system gate dedicated to intercepting successful `NtContinue` returns. This allows Pin to recognize successful `NtContinue` calls and choose where to resume executing the application.

C. System Call Interruptions

Some Windows system calls may cause the calling thread to enter an alertable state where the kernel is allowed to deliver pending asynchronous events to the application. These system calls are called *interruptible* or *alertable*. Pin must intercept the kernel delivery of asynchronous events to maintain control of the application.

Windows has two mechanisms for the delivery of asynchronous events to the user mode: asynchronous procedure calls (APCs) and *callbacks*. The system dispatches APCs and callbacks through entry points in the `ntdll.dll` system library. To regain control when the kernel delivers an asynchronous event, Pin overwrites the entry points with trampolines that jump to the corresponding event dispatchers in Pin.

Since Pin does not use alertable system calls and prohibits pintools from using alertable functions (see Section VII), only system calls in the application can be interrupted. As noted in Section IV-B, all interruptible system calls execute in the system gate. The system gate loads the complete application context before executing a system call, which simplifies handling interrupted system calls.

Handling APCs. When Pin intercepts an APC, it first checks whether it came from a new application thread. If the thread that called the APC is not found in the list of Pin-controlled application threads, Pin registers the new thread, initializes its thread-local state and starts executing the translated `KiUserApcDispatcher` code. (See Section VI for more details on threads.) If the thread is already registered with Pin, then the APC is being delivered while Pin’s system gate is executing a system call on behalf of the application. When the system delivers an APC, it passes the context of the interrupted system call as an argument on the stack. Pin must recover the native application context. This is straightforward because the system call emulator saves the original instruction pointer before all system calls and Pin’s system gate guarantees the registers contain original values. After the original interruption context is recovered, Pin transfers control to the translated `KiUserApcDispatcher` routine in the code cache. A program can return from an APC with the `NtContinue` system call (see Section IV-B). Nested APCs are not a problem – Pin’s system gate executes all interruptible system calls on the application stack and does not hold any lock because it uses only thread-local data.

Handling Callbacks. A callback differs from an APC in two respects: where the interrupted context is saved and how execution is resumed. When delivering a callback, the system saves the interrupted context in an internal data structure that is not accessible by Pin. The callback returns to the system via the `NtCallbackReturn` system call (or `int 2b` equivalent). Callbacks can be nested so the kernel pushes an interrupted context on its internal data structure when a callback is delivered and pops it after the callback returns.

Our approach to handling callbacks is similar to that of `DynamoRIO` [7]. Pin maintains a stack of callback entries, each of which contains the original return address of the corresponding interrupted system call. When Pin intercepts a callback at the entry to the callback dispatcher, it creates a new callback entry with the return address of the system call being interrupted. The system call emulator saves the return address before system call execution, as illustrated by Figure 2. Once the callback entry is created and pushed onto the dedicated stack, Pin transfers control to the translated `KiUserCallbackDispatcher` code in the code cache.

When the system call emulator intercepts `NtCallbackReturn`, it reads the return address of the interrupted system call from the top callback entry and saves it as the return address of the `NtCallbackReturn` system call being currently handled (as illustrated in Figure 3). The emulator then pops the callback entry from the stack and transfers control to Pin’s system gate where `NtCallbackReturn` is executed. When this system call returns (into Pin’s system gate), Pin restores the original PC of the thread by reading the address that was saved before `NtCallbackReturn` (see Figure 2). The restored PC is the return address of the system call that was interrupted by the callback.

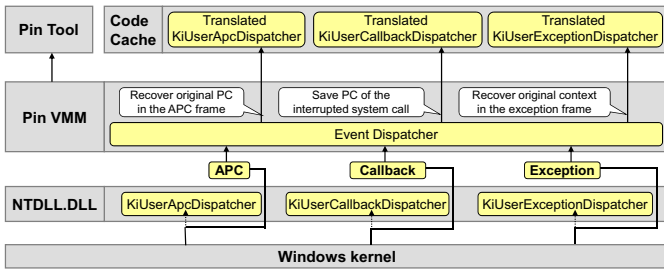


Fig. 3: Mechanism for handling APCs, callbacks, and exceptions in Windows.

V. EXCEPTIONS

Unlike asynchronous events (APCs, callbacks) that are queued and delivered at the next alertable system call, exceptions are synchronous events. The system delivers exceptions immediately when they are detected by hardware or generated by software. Exceptions do not necessarily cause abnormal termination of the process even if they are caused by an illegal instruction or an access violation. Windows provides a mechanism for handling exceptions, called Structured Exception Handling (SEH). Applications can use this mechanism to catch exceptions, analyze information associated with exceptions, and specify appropriate exception handlers. From Pin’s perspective, an exception is a control transfer from the kernel to user mode that must be intercepted to execute the SEH dispatcher under Pin’s control and provide it with the same information that would accompany the exception in the native application.

The system dispatches exceptions through the `ntdll.dll` routine `KiUserExceptionDispatcher`. The dispatcher receives two data structures on the stack: the context at the point where the thread was interrupted, and the exception record that specifies the exception type (code) and additional exception-specific information. Pin intercepts exceptions via a trampoline inserted at the entry to the original `KiUserExceptionDispatcher` routine.

Pin’s exception dispatcher checks the exception origin to distinguish application exceptions from internal Pin exceptions. Pin exceptions can occur when the JIT compiler tries to fetch application instructions from inaccessible memory. This can potentially occur if the JIT compiler fetches beyond the current instruction and actual program execution never reaches that address. In this case, Pin ends the translation block and dismisses the exception. If the program does try to execute at an inaccessible address, Pin generates a synthetic exception with the original context at the faulting point.

Hardware exceptions caused by the application appear in the Pin-controlled process as exceptions in the code cache. Pin does not insert instructions that may fault into the code cache, so only original application instructions may cause exceptions in the code cache. The JIT compiler ensures that translated instructions raise the same exceptions under the same conditions as the corresponding original instructions. This means that most of the exception’s attributes, passed to the exception dispatcher (in the exception record), have their original values. The exception context is the only data structure for which recovery of original values is nontrivial. Pin’s register allocator may assign application registers to different physical registers or spill them in order to free up registers for use by the instrumentation code. Some dynamic binary translation systems [3] keep the context-translation data for each potentially faulting instruction in the trace to enable support for precise exceptions in highly-optimized traces. We did not employ this technique because it would require a significant amount of additional persistent data to be stored for each trace.

Instead, we recompute the context-translation information when an exception occurs.

Hardware exception handling starts with finding the trace that contains the faulting instruction. The trace descriptor in the code cache directory contains the starting address and register binding at the trace entrance. Pin keeps this data to enable trace linking ([14]) and reuses it for exception context recovery. Once the trace is found, the JIT compiler retranslates the trace and generates a list of program counters and register bindings for each point where an exception could occur. The list is then scanned to find the register binding at the address where the current exception occurred. The original register state is restored by inverting the register bindings before transferring control to the translated `KiUserExceptionDispatcher` routine in the code cache.

Retranslating the trace in the JIT compiler costs approximately 20,000 cycles per original instruction. With an average trace length of 10 instructions, the overhead is 200,000 cycles, compared to 5,000 cycles to service an exception without Pin. Most applications do not have any exceptions, but we observed one application that experienced high overhead in Pin because it used memory faults to trigger memory allocation. The faults typically occurred on a small number of instruction addresses. To avoid the cost of retranslation for frequently occurring faults, Pin uses a small cache of register bindings indexed by exception address.

Besides hardware exceptions, the application can initiate software exceptions by calling the `RaiseException` Win32 API function. Pin generates these exceptions while executing the `NtRaiseException` system call on behalf of the application. All software exception arguments, including the register state in the exception context, have their native values because they are set up in user mode under Pin’s control. To distinguish between software and hardware exceptions, Pin checks the exception address. For software exceptions, the address points to the original application code and not to a code cache address. After a software exception is detected, Pin skips the context recovery and enters the code cache to execute the translated exception dispatcher.

If a pintool wishes to monitor exceptions in the application, it can do so by registering a `CONTEXT_CHANGE_CALLBACK` procedure. This callback provides the exception context and other exception attributes. The tool can inspect them and even handle the exception if it decides that the exception should not be delivered to the application. The same callback can be used for monitoring asynchronous events (see Section IV-C) in the application. Pin does not allow pintools to generate and to handle private exceptions and asynchronous events. Enabling this feature is future work.

VI. MULTITHREADING SUPPORT

Multithreading is pervasive in Windows applications, making its support critical in Pin. Some of Pin’s design decisions necessary for supporting Linux-based multithreaded applications were presented in prior work [10]. In this section, we focus on Windows-specific challenges in supporting multiple threads and describe solutions implemented in Pin.

A. Thread Creation Interception

To provide observability, Pin must be able to instrument every user-mode instruction from thread creation to the last user-mode instruction. Other than the pre-injection phase in the main thread, discussed in Section III, Pin achieves this goal. Monitoring the `NtCreateThread` system call will not catch threads that are remotely created by another process. Instead, Pin leverages the fact

that all new threads, including remotely created threads, begin with an APC. Pin watches APCs to see if they are dispatched in a thread that is not yet controlled by Pin (described in Section IV-C). When a new thread is observed, Pin redirects execution to the code cache, and adds the new thread to the Pin-controlled thread database.

To isolate an application thread from Pin, each thread has a shadow stack that is used by Pin and the pintool. It is important to preserve the application stack because some applications incorrectly access data beyond the top of the stack. There is a small amount of time just after the thread start that Pin runs on the application stack. Fortunately, this lack of isolation does not seem to be a problem because it occurs before any application data could be written to the stack.

B. Thread-Local Data

A thread representation in Pin is called a VMM thread object. This object is an aggregate that contains or points to all of Pin’s thread-local data, including the Pin stack, register spill area, state of active system calls, state of system libraries, and thread-local data for pintools. To enable fast access to thread-local data while running in the Pin VMM, we allocate one slot in the application’s thread-local storage (TLS) and use it as a pointer to the VMM thread object. One of the key thread-local elements is the spill area, which keeps values of spilled virtual registers. JIT-compiled traces use virtual registers that are bound to physical registers or memory slots in the spilling area. Traces in the thread-shared code cache that access spilled registers need fast access to the spill area. Instead of loading the spill area pointer from a TLS slot and then performing the fill or spill, we instead dedicate one general-purpose register to permanently point directly to the spill area. Stealing a general-purpose register allows the same implementation to be used for multiple architectures and operating systems. Loading the spill area pointer on demand is more complicated and is likely to be slower than dedicating a register.

C. Thread Synchronization

Aside from thread-local data, Pin has various global, thread-shared objects, including trace descriptors and page protection information. Access to the global objects in Pin is serialized by a single VMM lock. As a consequence, most execution paths through the Pin VMM have serialized code sections. Since these sections are relatively short and a typical program spends the majority of the time inside code cache traces and not inside the VMM [10], the VMM lock contention is low. Our initial implementation used a “sleep mutex” to guard global Pin objects. However, later on, we found that this synchronization mechanism does not work well in some heavily-multithreaded applications, like Windows Media Player, causing inversion of thread priorities in the Windows scheduler. In our current implementation, the VMM lock is a wrapper for the Windows critical section object, which is customized to avoid use of shared resources in ntdll.dll. The serialization of Pin activities does not interfere with the thread synchronization in the application because Pin does not use thread-shared objects in system libraries and consequently never tries to acquire process-wide locks that could be concurrently acquired by the application, e.g. the loader lock or process heap lock.

D. Cross-Thread Suspension and Context Manipulation

An application can use the `NtSuspendThread` system call to suspend a thread in the same or a different process. Handling this system call in Pin is especially challenging. Since the application has suspended the thread, there is no guarantee when the thread will resume execution. If the suspended thread holds a Pin lock, then Pin could deadlock. If the suspended thread continues by

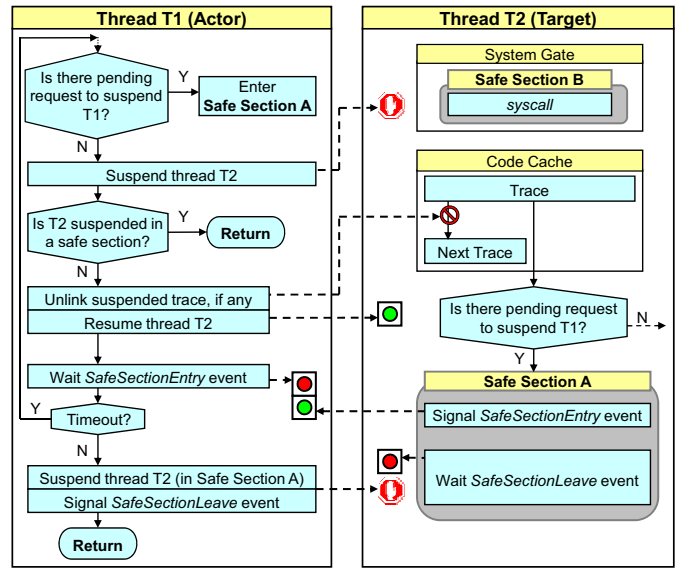


Fig. 4: Algorithm for suspending threads at a safe point.

executing in the code cache, then Pin must know not to reuse the code cache space of the resume point. The context of a suspended thread can be read and modified by the `NtGetContextThread` and `NtSetContextThread` system calls, but might have spilled registers to memory at the suspension point.

To resolve the above issues, Pin delays suspension until a thread reaches a safe point [6], which must be reachable from any point in the thread’s execution path within a limited time interval. This dictated the choice of two safe points in Pin, shown in Figure 4. Safe point A is in the VMM dispatch procedure, which is called on each trip through the VMM. If there is a pending request to suspend the thread, this procedure releases the VMM lock, signals that a safe state has been reached (`SafeSectionEntry` event) and waits for the thread to be suspended (`SafeSectionLeave` event). The thread can be suspended at any point between these two signals. The `SafeSectionLeave` event allows the thread to leave the safe point when another thread issues the `NtResumeThread` system call. While the thread is suspended in the safe point its original context is stored in memory and can be retrieved and modified by other threads. Safe point B is located in the system gate, which executes system calls on behalf of the application. The system gate does not hold locks while executing system calls and knows how to `get/set` original context of the suspended system call. Executing system calls in a safe section is an important feature, because it allows safe suspension of threads that are blocked in the kernel indefinitely.

A critical feature of the thread suspension algorithm is ensuring that the thread reaches a safe point within a limited time interval. Pin forces control to exit the code cache by patching all exits from the current instruction trace to jump back to the VMM [10]. An alternative would have been to insert explicit checks in the code that we generate, but that would have made the common case slow to support something that rarely happens.

The thread suspension mechanism has another application. Pin reuses this mechanism to emulate the `NtTerminateThread` system call, which is invoked when a thread attempts to terminate another thread. Pin suspends the target thread at a safe point before executing the `NtTerminateThread` system call, ensuring that the terminated thread does not hold a Pin lock.

VII. RUNTIME SUPPORT FOR THE PIN VMM AND PINTOOLS

To enhance productivity, Pin provides a conventional programming environment for Pin VMM and pintool developers. This includes using the C++ language including streams and Standard Template Library (STL), C run time (e.g. `printf()`, `malloc()`), and some Windows APIs (e.g. `GetProcAddress()`). However, the Pin VMM and pintool are DLLs in the application process, and calls into the same runtime libraries used by the application can cause unwanted interactions between Pin and the application. For example, when the application calls `malloc()` to allocate memory, we want to be able to instrument its entire execution down to the system call level. The pintool may need to call `malloc()`, but its execution should not be instrumented. Furthermore, a pintool call to `malloc()` could occur as part of the instrumentation of an application call to `malloc()`, and it would appear that the pintool call has re-entered `malloc` before the application call has finished. System libraries are designed to support concurrent execution by using locks and thread-local storage, but these methods are not effective because the application and instrumentation occur in the same thread. As a consequence, global data may be corrupted and synchronization operations can deadlock.

Pin employs several methods to avoid problems caused by concurrent calls to system library routines. Pin and the pintool link with a static C run time (CRT) library. Since the application cannot access code or data in the CRT static library, application-Pin interactions cannot occur through its static data. However, the application and Pin CRT both call the low-level `kernel32.dll` library, and it is not possible to duplicate it via static linking. To avoid conflicts for `kernel32.dll`, Pin provides independent implementations of some CRT functions such as `malloc()` and `free()`, and for some Win32 APIs such as `InitializeCriticalSection()`, `DeleteCriticalSection()`, and thread-local storage APIs. Another consequence of the instrumentation and application executing in the same thread is that they share the same Thread Environment Block (TEB). The TEB is a system data structure (allocated by the kernel in user space) that keeps the state of the image loader and various system DLLs [22]. When switching between the application and Pin, some TEB fields must be swapped, such as the last error number and stack boundaries.

Pintool developers must adhere to several rules in their use of the runtime. As described in Section IV-C, an APC or callback can be delivered when an alertable system call is executed. If Pin executes an alertable system call, then an APC or callback could be delivered at a point in the application where it was not previously possible, breaking the application. We ensure that the Pin CRT does not make alertable system calls. If a tool needs to call a Windows API directly, it must be done before the application starts or the tool author must ensure that the API does not make an alertable system call. Most tools only need to call the CRT, which is always safe, and relatively few Windows APIs make alertable system calls.

To the best of our knowledge, Pin permits a more standard C++ programming environment for instrumentation tools than other instrumentation systems. Pin guarantees safe use of the CRT library, even when the CRT calls system services. Pin preserves correct TEB values when switching between Pin and the application code. Pin also allows usage of Win32 APIs (exported by `kernel32.dll`) in the initialization procedure of Pin and pintools.

VIII. EXAMPLE INSTRUMENTATION TOOLS

To demonstrate the power and versatility of instrumentation, we describe Pin-based tools to analyze program performance. We first

TABLE II: Description of applications used in evaluation

Benchmark	Description / Workload
SPECINT 32	CPU 2000 Integer (icc for win32) / Ref inputs
SPECFP 32	CPU 2000 Floating Pt (icc for win32) / Ref inputs
SPECINT 64	CPU 2000 Integer (icc for win64) / Ref inputs
SPECFP 64	CPU 2000 Floating Pt (icc for win64) / Ref inputs
Illustrator 9.1	Adobe publishing / Exercise features via Visual Test
MS Excel XP	Spreadsheet / Exercise features via Visual Test
CINEBENCH	Image rendering / Render a scene
POVRAY	Image rendering / Render patio-radio_640_480_AA

TABLE III: Description of the instrumentation pintool configurations

Tool	Mode	Description
Native		Run without Pin; Used as baseline
Probe	Probe	No instrumentation; Probe-mode baseline
LocksAndWaits	Probe	Locking analysis (Intel Parallel Amplifier)
JIT	JIT	No instrumentation; JIT-mode baseline
BBCount	JIT	Increment a counter once per basic block
BBCount_mt	JIT	Increment a thread-private counter once per basic block
MemTrace	JIT	Record all memory refs in thread-private buffer; Baseline for memory analysis tools
MemError	JIT	Memory error analysis (Parallel Inspector)
DataRace	JIT	Data race analysis (Parallel Inspector)

give an overview of the tool’s purpose and then explain how it uses Pin to collect the data necessary for the analysis.

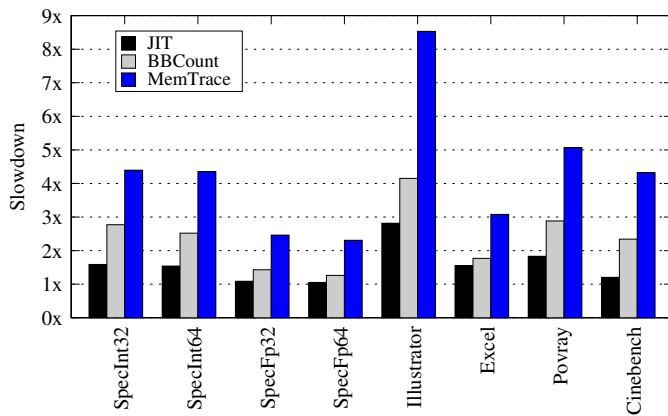
A. Intel® Parallel Amplifier

Intel® Parallel Amplifier [1] performs three types of analysis to help programmers improve program performance: *Hotspots*, *Concurrency*, and *Locks and Waits*. *Hotspots* attributes time to source lines and call stacks, identifying program regions that would benefit from tuning and parallelism. *Concurrency* measures the CPU utilization, giving whole program and per-function summaries. *Locks and Waits* measures the time multithreaded programs spend waiting on locks, attributing time to synchronization objects and source lines. Identifying locks that are responsible for wait time and the associated source lines helps a programmer to improve the CPU utilization of a parallel program.

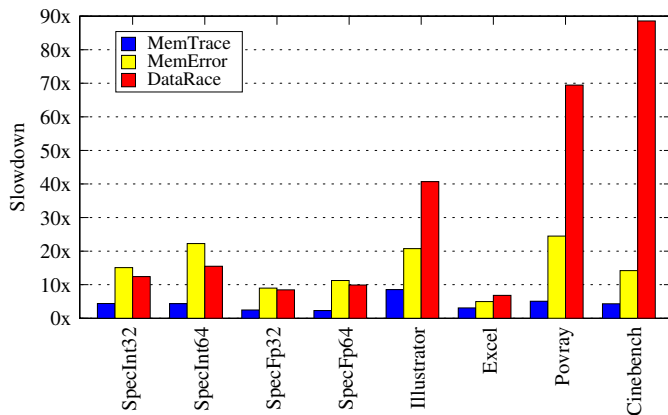
The data for *Hotspots* and *Concurrency* analysis comes from sampling. Intel® Parallel Amplifier uses Pin to instrument the application to collect data for the *Locks and Waits* analysis. Capturing accurate timing data requires low overhead instrumentation. The Locks and Waits analysis uses Pin probes to replace calls to synchronization APIs with wrapper functions, as demonstrated in Listing 2. The wrapper functions call the original synchronization function to record the wait time, synchronization object, and call stack.

B. CMP\$im

Memory system behavior is critical to parallel program performance. Computational bandwidth increases faster than memory bandwidth, especially for multi-core systems. Programmers must utilize as much of the bandwidth as possible for programs to scale to large processor counts. Hardware-based monitors can report summary statistics such as memory references and cache misses. However, they are not well suited for collecting more detailed information such as the degree of cache line sharing or the cache miss frequency resulting from false sharing. CMP\$im [13] uses Pin to collect the memory addresses of multithreaded and multiprocess programs and then uses a memory system software model to analyze program behavior. CMP\$im reports miss rates, cache line reuse, cache line



(a) Lightweight analysis



(b) Heavyweight analysis

Fig. 5: Slowdowns for various tools normalized to native execution

sharing, and coherence traffic. CMP\$im’s memory system model configuration is versatile and can be used to predict application performance on future systems. While CMP\$im is not yet publicly available, the Pin distribution includes the source for a simple cache model, `dcache.cpp`.

IX. PERFORMANCE

The performance of an application instrumented with Pin depends on the properties of the application and the tool. This section presents various combinations of applications and tools that characterize different points in the spectrum. We start with an overall performance characterization, and then examine factors that contribute to overhead, with special attention to Windows specific aspects. The evaluation uses a mix of SPEC and interactive GUI applications as well as a variety of instrumentation. Applications used in the evaluation are shown in Table II and tool configurations are shown in Table III.

A. Overall Overhead

The overhead that a user of Pin sees is most dependent on the tool. In this section we measure the performance of a variety of tools that covers the spectrum of actual use of Pin from simple analysis to heavyweight tools with complex runtime analysis. Figure 5 shows the slowdown for various tools normalized to native execution. The lightweight and heavyweight tools are divided into two figures so that each can have an appropriate scale for the slowdown.

The JIT configuration in Figure 5(a) executes the application under control of the JIT, with no instrumentation. It represents a lower

bound for lightweight tools that need the observability of JIT mode execution. We note that the JIT configuration for SPECINT is higher than SPEC FP because integer programs have more calls/returns and low trip count code [14]. Adobe Illustrator has the highest overhead – it is a GUI application that has a large amount of low trip count code [20]. Microsoft Excel is also a GUI application, but the workload allows a large amount of idle time, which hides overhead introduced by instrumentation. BBCount introduces one counter increment per basic block. It is a lower bound for tools that only observe control flow. Its overhead grows as the average number of instructions per branch shrinks. MemTrace instruments every memory reference and its overhead is an indication of the percentage of instructions that are memory references. It is a lower bound for instrumentation tools that observe memory reference patterns. Although SPEC win64 programs have fewer memory references than win32 [26], it does not appear to affect performance.

In Figure 5(b) we show slowdown for heavyweight tools. We include MemTrace again because MemError and DataRace primarily analyze memory references. The MemError tool is slower for 64-bit SPEC programs than 32 because it must look up every program memory reference in a data structure, and the lookup function is slower for 64-bit addresses. Otherwise, the MemError slowdown tracks the MemTrace slowdown. The DataRace slowdown is low for all programs except Adobe Illustrator, POV RAY, and CINEBENCH because they are the only multithreaded programs and the DataRace tool does minimal work until the second thread is created. The Adobe Illustrator workload has more idle time than POV RAY and CINEBENCH, which explains its low slowdown despite the high MemTrace slowdown.

It is important to note that the slowdown for executing the program with Pin and no instrumentation (JIT) is small compared to the tools that do something useful (MemError and DataRace) and the time for the JIT configuration is not a good predictor for heavyweight tool time [17]. Nevertheless, developers of commercial pintools are reducing the overhead of their tools and we expect that in the future the base overhead of Pin will become more critical.

B. Overall Time Breakdown

To understand the sources of overhead better, we measured the time spent in the VMM, JIT compiler, and code cache for various applications and tools. *Code cache* is the time spent executing the application or its instrumentation, *JIT* is the time spent translating the application and inserting instrumentation, and *VMM* includes time processing the operating system events like system calls, exceptions, and APCs, invoking the JIT compiler, and management of the code cache. The focus of this paper is Windows support, so Figure 6 reports the breakdown of total time for Excel and Illustrator, which were the only applications spending a significant amount of time in the VMM. Even for these applications, the VMM time is relatively small, never exceeding 6.3%. The JIT compiler overhead is larger, and is a function of the code footprint and complexity of the instrumentation.

C. Kernel Interaction Overhead

Our application benchmarks show that Pin’s Windows support has a small impact on performance. To characterize operating system related effects better, we wrote microbenchmarks to evaluate the overhead introduced by Pin handling application-kernel interaction events: system calls, exceptions, APCs and callbacks. To measure system call overhead, we use a loop that executes `NtDelayExecution()` system calls with sleep time of 0. Exceptions are measured by repeatedly reading from an invalid memory

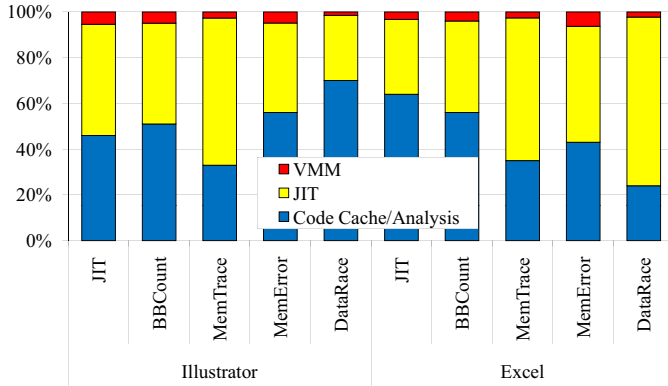


Fig. 6: Breakdown of time spent in code cache, JIT compiler, and VMM.

TABLE IV: Slowdown relative to native execution for system interactions

	Pin	DynamoRIO
System Calls	12.01	5.50
Exceptions	10.45	11.81
APCs	3.11	3.13
Callbacks	1.80	1.79

location within a `__try/except` block. To measure APCs, one process uses `QueueUserAPC()` to send APCs to another process in a loop. Callbacks are measured by using `PostMessage()` to send messages from one process to another.

Table IV shows the relative slowdown for each microbenchmark for Pin and DynamoRIO on 32-bit Windows relative to native execution. Compared to DynamoRIO, Pin is faster for exceptions, slower for system calls and the same for APC and Callbacks. System call overhead has a higher slowdown than the other events. We believe this is because there is a high fixed cost for leaving the code cache and entering the Pin VMM (~ 3000 cycles), and the native cycle count for system calls is lower (~ 500) compared to the other events. Most system calls do not need to be handled by the VMM so we could reduce the overhead by providing a fast path that checks the system call number.

We added a register binding cache (see Section V) to accelerate exception handling. To measure its effectiveness, we reran the Exceptions microbenchmark with the register bindings cache disabled. Exception overhead increased to 60x native execution. We are only aware of a single application where exceptions are frequent. We could not include complete data in this paper, but can report that the register binding cache grows to 159 entries, and has a 99.9% hit rate.

In Table V we measure how often the system events occur to gauge their relative importance. System calls are far more frequent than other events. We estimate that system call related overhead accounts for 3.3% and 2.8% of the total run time for the JIT configuration for Illustrator and Excel in Figure 5(a). The net overhead is insignificant for the other events.

TABLE V: Counts for application-kernel interaction events

	Illustrator	Excel	CINEBENCH	POVRAY
System Calls	1,659,298	658,683	101,700	75,313
Exceptions	1	0	0	0
APCs	6	6	24	24
Callbacks	73,062	68,767	961	7,682

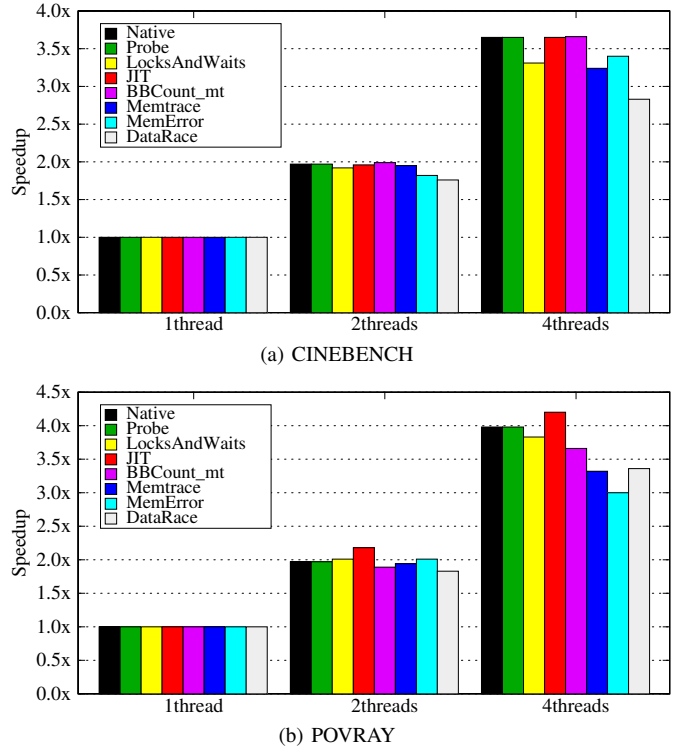


Fig. 7: Speedup for instrumented parallel applications

D. Scalable Workload Performance

In this section, we examine the impact of Pin and instrumentation on the scalability of an application. To measure scalability in execution time we use two parallel workloads, MAXON CINEBENCH and POVRAY. Figure 7(a) shows the scaling in CPU time for a variety of configurations. For each configuration, we normalize the time for 2 and 4 threads to the time for 1 thread for the same configuration. The native application scales well, achieving a 3.5x speedup on 4 processors. We show in Figure 5(b) that complex analysis adds overhead, but expect the instrumented application to scale when we add more threads. Running under the control of Pin with no instrumentation (JIT) achieves a similar scalability to native. Although the Pin VMM and JIT compiler are guarded with a single lock, it does not appear to impact scalability of the applications. For the heavyweight tools, we see there is a slight drop off in scalability. For MemTrace, we believe the reduced scalability is caused by limited memory bandwidth. The other tools do more work per address and memory bandwidth is less of a problem, but there is contention for tool specific data structures. Figure 7(b) shows the same data for the POVRAY benchmark. The results are similar; the native POVRAY execution and JIT have almost perfect scaling. Adding instrumentation decreases the scaling.

X. RELATED WORK

Several process-level dynamic binary instrumentation systems have been developed over the past decade. Dynamo [2] was developed at Hewlett-Packard Labs to provide dynamic optimization capabilities to PA-RISC applications. DynamoRIO [8] was then ported to IA-32 with support for Windows and Linux applications, again with the motivation of providing dynamic optimization capabilities, but was also used for instrumentation. Valgrind [17] is a Unix-only instrumentation system targeted at dynamic program analyses that

require shadowing of the application computation. Strata [21] is a dynamic modification engine for IA-32 and SPARC on Unix. HDTrans [23] is a lightweight open-source translation framework for IA-32 on Unix and focuses on performance and simplicity at the expense of user-friendliness. Nirvana [5] provides deterministic trace recording and playback and the ability to instrument them on Windows. Detours [12] allows instrumentation similar to Pin's probe mode, but does not have the ability to do fine-grained instrumentation of individual instructions and memory operations. Pin [14] provides fine-grained, dynamic binary instrumentation for IA-32, Intel64, Itanium, and ARM on Windows, Linux, and BSD. Only DynamoRIO, Nirvana, and Pin provide fine-grained instrumentation of Windows applications. Nirvana is not generally available outside of Microsoft and the details of its interface to the operating system have not been published. DynamoRIO is similar in capabilities to Pin. Pin's focus on being a commercial instrumentation system for a broad range of analysis tools has led to different design decisions as is discussed throughout the paper. Examples are its method for injecting early in the process lifetime and its support for access to conventional runtime support by instrumentation tools.

XI. CONCLUSIONS

Dynamic binary instrumentation systems have emerged as invaluable tools for performance analysis, simulation, optimization, and security. Most of the prior work discussing instrumentation infrastructures has focused on executing Unix applications, despite the ubiquity and importance of Windows-based applications. This paper presents the design, implementation, and performance of the Windows version of Pin. The challenges for providing a robust instrumentation system on Windows lie in managing the kernel/application transitions, injecting the runtime agent into the process, and isolating the instrumentation from the application. To manage kernel/application transitions, Pin needs to interact with the Windows kernel at a level below the published APIs. Using a combination of dynamic discovery and worst case assumptions, Pin is able to control and to instrument Windows processes without being tied to a specific Windows release. We examine Pin's overhead by looking at typical Windows applications being instrumented with simple tools as well as Pin-based commercial program analysis products. The single biggest factor affecting performance is the type of analysis performed by the tool. Lightweight tools can have near zero overhead and complex analysis of memory behavior can increase the running time by 90x native execution. Overhead for processing system calls can add as much as 5% to lightweight tools, but in general the operating system overhead is not significant. While the proprietary nature of Windows makes measurement and analysis difficult, Pin opens the door to understanding program behavior.

REFERENCES

- [1] Intel parallel studio. Intel Document 321166-002US.
- [2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Programming Language Design and Implementation*, pages 1–12, Vancouver, Jun 2000.
- [3] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach. IA-32 Execution Layer: A two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems. In *36th Int'l Symposium on Microarchitecture*, pages 191–201, 2003.
- [4] D. Bartholomew. QEMU: A multihost, multitarget emulator. *Linux Journal*, 2006(145):3, 2006.
- [5] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *Virtual Execution Environments*, pages 154–163, Ottawa, 2006.
- [6] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, MIT, Cambridge, MA, Aug 2004.
- [7] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *4th Workshop on Feedback-Directed and Dynamic Optimization*, 2000.
- [8] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Code Generation and Optimization*, pages 265–275, San Francisco, CA, Mar 2003.
- [9] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference*, pages 15–28, Boston, MA, Jun 2004.
- [10] K. Hazelwood, G. Lueck, and R. Cohn. Scalable support for multi-threaded applications on dynamic binary instrumentation systems. In *Int'l Symposium on Memory Management*, Dublin, Ireland, Jun 2009.
- [11] K. Hoste and L. Eeckhout. Characterizing the unique and diverse behaviors in existing and emerging general-purpose and domain-specific benchmark suites. In *Int'l Symposium on Performance Analysis of Systems and Software*, pages 157–168, Apr 2008.
- [12] G. Hunt and D. Brubacher. Detours: binary interception of win32 functions. In *3rd USENIX Windows NT Symposium*, Berkeley, CA, 1999.
- [13] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob. CmpSim: A pin-based on-the-fly multi-core cache simulator. In *4th Workshop on Modeling, Benchmarking and Simulation*, 2008.
- [14] 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 *Programming Language Design and Implementation*, Chicago, IL, Jun 2005.
- [15] J. Moses, K. Aisopos, A. Jaleel, R. Iyer, R. Illikkal, D. Newell, and S. Makineni. CmpschedSim: Evaluating os/cmp interaction on shared cache management. In *Int'l Symposium on Performance Analysis of Systems and Software*, pages 113–122, Apr 2009.
- [16] G. Nebbett. *Windows NT/2000 Native API Reference*. New Riders Publishing, Thousand Oaks, CA, 2000.
- [17] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Programming Language Design and Implementation*, pages 89–100, San Diego, CA, Jun 2007.
- [18] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large intel itanium programs with dynamic instrumentation. In *37th Int'l Symposium on Microarchitecture*, pages 81–92, Dec 2004.
- [19] C. Pereira, H. Patil, and B. Calder. Reproducible simulation of multi-threaded workloads for architecture design exploration. In *Int'l Symposium on Workload Characterization*, pages 173–182, Sep 2008.
- [20] V. J. Reddi, D. Connors, R. Cohn, and M. D. Smith. Persistent code caching: Exploiting code reuse across executions and applications. In *Code Generation and Optimization*, pages 74–88, 2007.
- [21] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *Code Generation and Optimization*, pages 36–47, Mar 2003.
- [22] D. Solomon and M. Russinovich. Microsoft Press, Redmond, WA, 2009.
- [23] S. Sridhar, J. S. Shapiro, E. Northup, and P. P. Bungale. HDTrans: An open source, low-level dynamic instrumentation system. In *Virtual Execution Environments*, pages 175–185, Ottawa, 2006.
- [24] A. Srivastava and A. Eustace. ATOM: a system for building customized program analysis tools. In *Programming Language Design and Implementation*, pages 196–205, Orlando, FL, 1994.
- [25] VMware. *VProbes Programming Reference*.
- [26] D. Ye, J. Ray, C. Harle, and D. Kaeli. Performance characterization of SPEC CPU2006 integer benchmarks on x86-64 architecture. In *Int'l Symposium on Workload Characterization*, pages 120–127, Oct 2006.