Transparent Dynamic Instrumentation

Derek Bruening
Google, Inc.
bruening@google.com

Qin Zhao
Google, Inc
zhaoqin@google.com

Saman Amarasinghe
Massachusetts Institute of Technology
saman@csail.mit.edu

Abstract

Process virtualization provides a virtual execution environment within which an unmodified application can be monitored and controlled while it executes. The provided layer of control can be used for purposes ranging from sandboxing to compatibility to profiling. The additional operations required for this layer are performed clandestinely alongside regular program execution. Software dynamic instrumentation is one method for implementing process virtualization which dynamically instruments an application such that the application's code and the inserted code are interleaved together.

DynamoRIO is a process virtualization system implemented using software code cache techniques that allows users to build customized dynamic instrumentation tools. There are many challenges to building such a runtime system. One major obstacle is *transparency*. In order to support executing arbitrary applications, DynamoRIO must be fully transparent so that an application cannot distinguish between running inside the virtual environment and native execution. In addition, any desired extra operations for a particular tool must avoid interfering with the behavior of the application.

Transparency has historically been provided on an ad-hoc basis, as a reaction to observed problems in target applications. This paper identifies a necessary set of transparency requirements for running mainstream Windows and Linux applications. We discuss possible solutions to each transparency issue, evaluate tradeoffs between different choices, and identify cases where maintaining transparency is not practically solvable. We believe this will provide a guideline for better design and implementation of transparent dynamic instrumentation, as well as other similar process virtualization systems using software code caches.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – Optimization, Run-time environments

Keywords Process Virtualization, Dynamic Instrumentation, Transparency, Runtime System

1. Introduction

DynamoRIO [1] is a runtime code manipulation system that dynamically instruments an application such that the application's code and the inserted code are interleaved together. It provides a virtual execution environment within which arbitrary unmodified applications, even legacy software with no available source code, can be executed, monitored, and controlled. A software code cache is used to implement the virtual environment. The provided layer of control can be used for various purposes including security enforcement [14, 22], software debugging [8, 43], dynamic analysis [42, 45], and many others [40, 41, 44]. The additional operations are performed clandestinely alongside regular program execution.

First presented in 2001 [9], DynamoRIO evolved from a research project [6, 10] for dynamic optimization to the flagship product of a security company [22] and finally an open source project [1]. Presently it is being used to build tools like Dr. Memory [8] that run large applications including proprietary software on both Linux and Windows. There are many challenges to building such a system that supports executing arbitrary unmodified applications. The most critical challenge, however, is *transparency*: preventing a target application's native behavior from changing when running inside DynamoRIO.

1.1 Transparency Challenges

Many applications perform introspective operations such as reading function return addresses, iterating loaded modules, counting the number of threads, inspecting resource usage, etc. If DynamoRIO makes any modifications that violate the expected execution environment, in the best case the application's behavior will deviate slightly, but often it will crash. DynamoRIO must have its hands everywhere to maintain control, yet it must have such a delicate touch that the application cannot tell it is there. Table 1 shows a list of transparency requirements and specific applications that fail to run correctly when DynamoRIO does not preserve each given aspect of transparency.

In addition, DynamoRIO abstracts away the details of the underlying system and provides a simple interface for creating a custom tool, or *client* of the system, that is able to observe and modify a monitored application in a customized manner. In order to be widely usable, it must avoid imposing too many restrictions on clients: the client writer should not need to be a dynamic instrumentation expert and should be able to use traditional programming languages and libraries. Since standard libraries are not written to operate clandestinely, DynamoRIO must not only perform its own operations transparently but must also transform the client and its libraries such that they operate transparently within the monitored application.

Clearly, being transparent is critical in order to run a wide range of applications. While building *efficient* code manipulation (or instrumentation, translation, etc.) systems is a well-studied research topic [2, 6, 13, 15, 25, 32], we strongly believe that making the runtime environment *transparent* is the most important criterion for a successful system. To the best of our knowledge, this is the first paper that focuses on this critical issue.

Traditionally, transparency has been addressed on an ad-hoc basis. As there are no classifications or descriptions of transparency requirements, developers of a system only consider the need for any particular aspect of transparency when a target application starts

VEE'12, March 3–4, 2012, London, England, UK.

Category	Transparency requirement	Application(s) that fail if requirement is not met
Code	execute self-modifying code	Adobe Premiere
	execute code on the stack	MS Management Console
	execute dynamically generated code	Virtualdub, Java, .NET
Data	preserve beyond top of stack	MS Office
	support stack pointer not pointing at stack	Adobe Premiere
	preserve return address	Any application using COM
	hide from self-introspection	Mozilla
	preserve eflags across indirect branches	libc, code generated by Visual Studio compiler
Concurrency	support debugger threads	Visual Studio debugger
	separate system thread local storage slots from app	MS Exchange
Other	separate system libraries from app	Windows graphical applications
	separate system loader from app	Windows graphical applications
	deliver faults	MS Office
	separate system files from app	shells and daemons

Table 1. A list of transparency requirements in process virtualization systems and specific applications that fail to run correctly when each given requirement is not preserved. Section 6.1 elaborates on each of these examples.

misbehaving. In this work we attempt to change that by describing a necessary set of transparency requirements for running mainstream Windows and Linux applications. We believe this will provide a guideline for better design and implementation of future systems.

In this paper, we study the full spectrum of transparency challenges while building DynamoRIO. Our experiences have led us to formulate three *transparency guidelines* that should be followed when designing and implementing systems like DynamoRIO:

1. *Leave the application unchanged whenever possible.* Even when changes can lead to performance gains, in nearly every case such changes cause at least one application to fail. If correctness is a goal, design the system to perturb as little as possible.
2. *When changing the application is unavoidable, ensure the change is imperceivable to the application.* Some changes are unavoidable, in which case every effort should be made to hide the change from the application.
3. *Avoid resource usage conflicts.* Where possible, acquire resources directly from the operating system; if libraries must be used, fully isolate them from the application.

How these three guidelines are applied during the design and implementation of DynamoRIO are discussed throughout the rest of the paper.

1.2 Contributions

The contributions of this paper include:

- We identify three *transparency guidelines* for the design and implementation of transparent instrumentation systems.
- We enumerate and classify the full range of transparency issues.
- We discuss possible solutions to each issue, and evaluate tradeoffs between different choices.
- We identify cases where maintaining transparency is not practically solvable.
- We describe a necessary set of transparency requirements for running mainstream Windows and Linux applications.

We hope this paper can be used as a guide for building dynamic instrumentation tools and other similar process virtualization systems.

1.3 Paper Organization

The rest of the paper focuses on the transparency challenges of building a dynamic instrumentation system and is organized as follows: Section 2 describes the transparency issues when designing a dynamic instrumentation framework, while Section 3 discusses the implementation challenges of a transparent system. Section 4 describes the necessary transparency support for customized user components, and Section 5 discusses the limits of transparent systems. Section 6 evaluates the impact and tradeoffs of different solutions. Section 7 presents related work, and Section 8 concludes the paper.

2. Transparency in System Design

A key principle of designing transparent instrumentation is keeping as many aspects of the application as possible unchanged. This is *Guideline 1*, the first of our *transparency guidelines*. This applies to the original application binary, the application code, the application data, the number of threads, etc. If it is unavoidable to change something, the system should disguise the change to appear to the application as if it were unmodified (*Guideline 2*). In addition, to support arbitrary applications, DynamoRIO, or any similar system, cannot make any assumptions about a program's compilation, source code or annotation availability, stack or heap usage, or any of its dependences on the instruction set architecture or operating system. DynamoRIO can only assume the bare minimum architecture and operating system interfaces, and request resources directly from the operating system to avoid any resources usage conflicts (*Guideline 3*).

In this section, we categorize transparency issues into three groups: code transparency, data transparency, and concurrency transparency. We discuss each type of issue and possible solutions.

2.1 Code Transparency

Code transparency refers to the faithful reproduction of application behavior with respect to changes in or references to its code when executing in a process virtualization system. A process virtualization system implemented as a simulator or interpreter can control the program's execution via software interpretation without affecting its code, but the resulting runtime overhead is too high to run any large applications. Another approach is to use in-place modification of the application code to insert control and tool operations. In addition to presenting transparency challenges, this technique is unable to support the fine granularity of each individual instruc-

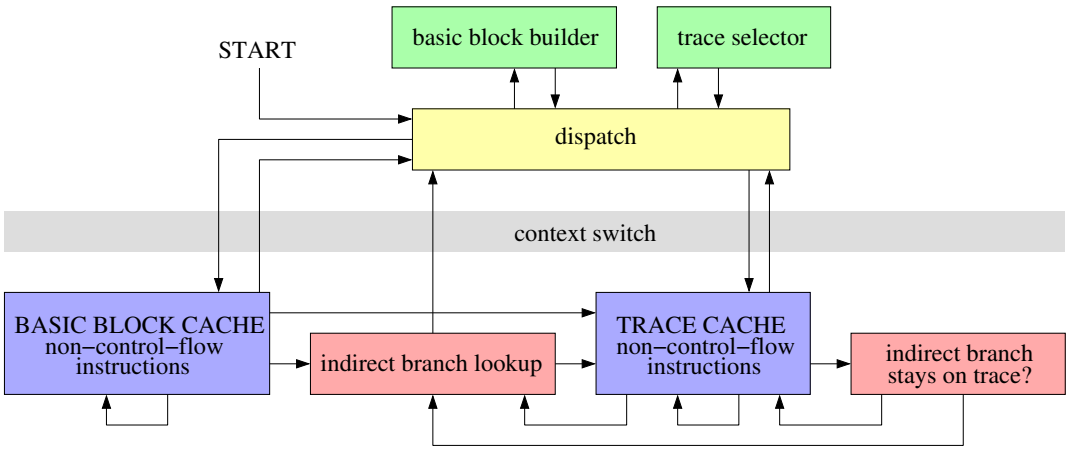

Figure 1. DynamoRIO system overview. A context switch separates the code cache (application state) from DynamoRIO code (system state); both execute in the same process and address space. Application code is copied into the two caches, with control transfers (shown by arrows in the figure) modified in order to retain control.

tion that DynamoRIO supports for its tools to control and manipulate a target program. In contrast to the two mentioned approaches, a software code cache can efficiently manipulate code and insert additional instructions without changing the original program. In addition to being used by DynamoRIO, code caches are also used by many similar systems including Pin [25], QEMU [5], and Valgrind [30].

2.1.1 Software Code Cache

Figure 1 shows the flow of control between the components of DynamoRIO and its code caches. When running an application, DynamoRIO copies the application code one *dynamic basic block* at a time into its basic block code cache and executes it there *in lieu of* the original code. A block that directly targets another block already resident in the cache is linked to that block to avoid the cost of returning to the DynamoRIO dispatcher. Frequently executed sequences of basic blocks are combined into *traces*, which are placed in a separate code cache. In the rest of the paper we will refer to both basic blocks and traces in code caches as code *fragments*. This software code cache and redirection complicates transparency, but no alternative approach provides sufficient flexibility and efficiency.

2.1.2 Application Address Transparency

When using a software code cache, although the application's code is copied into a cache, every address manipulated by the application must remain an original application address. DynamoRIO must translate indirect branch targets from application addresses to code cache addresses, and conversely if a code cache address is ever exposed to the application, DynamoRIO must translate it back to its original application address. The latter occurs when the operating system passes a machine context to a signal or exception handler. In that case both the faulting or interrupted address and the complete register state must be made to look like the signal or exception occurred natively, rather than inside the code cache where it actually occurred. Details of how to accomplish this will be discussed in Section 3.3.

2.1.3 Code Cache Consistency

DynamoRIO must ensure that each cached copy of application code is consistent with the original version in application memory. One common case where consistency is broken is unmapping of a file containing code (typically a shared library) from memory. As the application must make an explicit request to the kernel to accomplish the unmap, DynamoRIO need only monitor each system call that frees an area of the address space, and flush all cache fragments that contain pieces of code from that region.

The original code might also be modified via either self-modifying code or re-use of the memory region for dynamic code generation. DynamoRIO uses hardware page protection to detect such code modification by marking all regions that contain code in the code cache read-only, and keeping a list of all such regions. If such a region is written to, DynamoRIO traps the fault, flushes the code for that region from the code cache, removes the region from the list, marks the region as writable, and then re-executes the faulting write. Additionally, DynamoRIO intercepts Windows' `NtQueryVirtualMemory` system call and modifies the information it returns to pretend that areas it made read-only are in fact writable (*Guideline 2*). If the application changes the protection on a region DynamoRIO has marked read-only, it must update the information so that a later write fault will properly go to the application.

One complication with page protection occurs when the application requests output from a system call at an address that the virtualization system has made read-only. One solution, if all system call parameters are known, is to look for any output parameters that point at read-only pages and swap those pages to use sandboxing prior to invoking the system call.

Handling additional complications, including when the writing instruction and the target of a code modification are on the same page, are described elsewhere [7].

2.2 Data Transparency

Although the application code has been redirected to a code cache, application data can and should remain unchanged (*Guideline 1*).

2.2.1 Application State Preservation

Using a software code cache, the execution of application code is interleaved with the execution of DynamoRIO and the client's code. The application state must be preserved when execution exits the code cache to DynamoRIO and restored when it enters the cache. The application state to be saved includes the complete

register state, conditional flags (`eflags` in IA-32), and floating-point state. If the system shares libraries with the application, per-library persistent state like `errno` in `libc` should also be saved. It is possible to reduce the overhead of each context switch by not preserving all of the state, e.g., not saving and restoring floating-point state when the intervening system code does not perform any floating-point operations. However, such optimizations must be careful to not cause any transparency violations. For example, early versions of DynamoRIO assumed that `XMM` registers were not used by its own code on the IA-32 architecture and did not preserve them on context switches. However, later implementations of `memset` and `strlen` from `gcc` and in the C library used by DynamoRIO and clients began to use register `xmm0`, violating the assumption and causing application crashes.

DynamoRIO simply copies the application code into the code cache with minimal modifications and performs a full context switch when execution transfers between the code cache and DynamoRIO. In contrast, Valgrind maintains virtual application state including the program counter and updates it after every operation translated from original application code. In other words, Valgrind emulates the application's execution while natively executing Valgrind's own code, while DynamoRIO natively executes either one or the other, context switching between the runtime system and the application process. By minimizing the number of context switches via linking code fragments in code caches, the approach adopted by DynamoRIO incurs much lower runtime overhead than Valgrind's approach.

2.2.2 Stack Transparency

DynamoRIO uses a private stack and never touches the application's stack (*Guideline 1*).

It is tempting to assume that the space beyond the top of the application stack is not accessed and can be used for scratch space by a dynamic instrumentation system. For instance, HDTrans [36] spills registers to the application stack. However, some applications do access data beyond the top the stack. For example, Microsoft Office has code that stores data on the top of the stack, moves the stack pointer to the previous location, and then accesses the data. Additionally, some hand-crafted code uses the stack pointer as a general-purpose register, in which case one cannot even assume that the memory pointed at by the stack pointer is safe to access.

Another tempting way to use the application stack is using code cache addresses as return addresses instead of the original application return addresses [20], which allows an instrumentation system to use `ret` instructions directly and avoid any translation cost on function returns. However, this violates application address transparency (Section 2.1.2). Many applications examine their stack. For example, position-independent code obtains the current program counter by making a call to the next instruction and then popping the return address. The system can do pattern matching on this and other common ways the return address is read in order to get many applications to work, but even some of the SPEC 2000 CPU benchmarks [38], like `perlbmk`, read the return address in too many different ways to detect easily. Moreover, the return address replacement makes it difficult to construct a call stack from the application stack, and causes problems when debugging the application.

2.2.3 Heap Transparency

Memory allocated by DynamoRIO must be separate from that used by the application, for two reasons. First, the memory allocated for DynamoRIO should not interfere with the data layout of the application or with application memory bugs, e.g., heap buffer overflows. Additionally, sharing heap allocation routines with the application violates *Guideline 3*. Most heap allocation routines are thread-safe but not re-entrant (Section 3.1) and are thus not safe to call by DynamoRIO. Instead, DynamoRIO obtains its memory directly from system calls and parcels it out internally with a custom memory manager.

2.2.4 Self-Introspection Transparency

Many applications perform various introspective operations: iterating over loaded libraries, counting the number of threads, inspecting resource usage, etc. For instance, Linux's process information pseudo-file system (`/proc`) allows users or applications to inspect the status of a process, including memory maps, file usage, and other information. DynamoRIO intercepts system calls that query information and modifies the results returned to the application in order to hide DynamoRIO's presence. Section 3.2 discusses particular system calls that must be intercepted.

2.3 Concurrency Transparency

Multi-threaded applications have their own transparency issues beyond the code and data transparency discussed so far.

2.3.1 Thread Transparency

There are several options to map multi-threaded application execution onto the runtime environment. DynamoRIO uses application threads as system threads and creates no new threads, which avoids interfering with applications that monitor all threads in the process. DynamoRIO code is executed by application threads, with a context switch to separate its state from application state. Each application thread has its own separate DynamoRIO state and stack.

Using a separate system thread per application thread can be more transparent by truly separating contexts, in particular Thread-Local Storage (TLS), and avoiding having to handle application threads suspending threads that are executing in system code. However, it can also reduce application scalability in applications with hundreds or thousands of threads by doubling the number of threads. In addition, the communication and synchronization between an application thread and its system thread might cause performance problems. It is also possible to multiplex all application threads onto one single system thread, but this arrangement is unlikely to be performant enough. For example, Valgrind serializes all threads' execution, which incurs significant slowdown.

2.3.2 Synchronization Transparency

Sharing locks with the application can cause many problems and should be avoided (*Guideline 3*). Concurrency is challenging enough to get right in a single body of code where protocols can be agreed upon and code changed to match them. When dealing with an arbitrary application, the only viable solution, adopted by DynamoRIO, is to avoid acquiring locks that the application also acquires. Similarly, DynamoRIO must worry about application threads sharing system routines and locks, and cannot allow an application thread to suspend another thread that is inside a non-reentrant system routine or holding a system lock.

Even without sharing locks, the application and system may still interfere with each other and cause deadlock. For example, if one application thread is waiting for an application lock to be released by another thread, which is blocked on acquiring a system lock held by the first thread, neither thread can make forward progress. To prevent such deadlocks, DynamoRIO's solution is to enforce that no DynamoRIO lock can be held when a thread is executing application code inside the code cache.

2.3.3 Memory Ordering Transparency

If a dynamic instrumentation system modifies application memory accesses, or re-orders application instructions, it could cause a change in behavior with respect to the order of memory references seen across threads. Without any particular tool running on top of

it, a dynamic instrumentation system should avoid such changes. DynamoRIO does not make these types of modifications. If a tool decides to do so, it can choose accept the consequences as a trade-off.

Additionally, reading another thread's memory could result in behavior that differs from native execution, due to timing differences. The only application memory locations that need to be read by DynamoRIO are indirect branch targets and certain system call arguments on Windows where arguments are stored in memory. This minimal memory interaction results in minimal interference and does not add any races. Thus, in a race-free application, the dynamic instrumentation system will also be race-free. In an application with races, the behavior when running under the dynamic instrumentation system will contain a valid ordering that can occur natively, though it may not be the behavior that is seen most often natively due to timing changes. This may be unavoidable. Section 5.1 discusses related challenges.

3. Transparency in System Implementation

Beyond the design issues, there are many additional challenges when actually implementing a dynamic instrumentation system like DynamoRIO. Ideally, DynamoRIO's resources should be completely disjoint from the application's. However, that is not possible when DynamoRIO is executing inside the same process as the application. DynamoRIO must do its best to avoid conflicts (*Guideline 3*).

3.1 Library Transparency

Sharing libraries with the application can cause problems with reentrancy and corruption of persistent state like error codes. DynamoRIO code can execute at arbitrary points in the middle of application code. If both the application and DynamoRIO use the same non-re-entrant library routine, DynamoRIO might call the routine while the application is inside it, causing incorrect behavior. For example, if the system calls *malloc* while the application is inside that same routine, *malloc*'s global state may be in an inconsistent state. If a library routine acquires locks, deadlock can easily occur even though the routine is thread-safe.

To avoid the problems described above, early versions of DynamoRIO took the following approach: (*i*) preserve persistent library state (e.g., `errno`) on each context switch; (*ii*) only use reentrant library routines, implementing separate versions of non-trivial routines like `vsnprintf` when necessary; and (*iii*) request resources such as memory and files only from system calls and never from user libraries.

The current implementation of DynamoRIO loads a separate copy of application libraries for its own use and use by clients, to avoid conflicts with the application. However, duplicating libraries does not solve all of the problems. Some routines may still conflict with the application on resource usage, such as `malloc` requesting memory via the `brk` system call, which must be redirected to DynamoRIO's own routines.

Another example is Thread Local Storage (TLS) access. In 64-bit Linux on the AMD64 architecture, TLS is accessed via the `fs` register. If executed unchanged, the two sets of libraries must use the same `fs` register and its value must be swapped to maintain transparency. An alternative approach taken by DynamoRIO is to mangle each application TLS access to not use a segment register, avoiding the expensive swapping.

3.2 System Calls

DynamoRIO must understand and handle system calls properly for two reasons: (*i*) it must execute raw system calls itself to request resources, and (*ii*) it must monitor certain system calls made by the application in order to maintain system state and to enforce transparency.

The system call interface on Linux and most operating systems is a standard mechanism for requesting services (Figure 2a). However, on Windows, the documented method of interacting with the operating system is not via system calls but instead through an application programming interface (the *Win32 API*) built with user libraries on top of the system call interface (Figure 2b). Experience from early versions of DynamoRIO tells us that using the Win32 API interface causes numerous transparency issues including problems described in Section 3.1. The system call interface (Figure 2c) must be used, rather than the API layer, which, unfortunately, binds DynamoRIO to an undocumented interface that may change without notice in future versions of Windows.

DynamoRIO can fully control every process action in user mode, but usually has little control over kernel mode actions. It must intercept and monitor application system calls and modify them if necessary. For example, DynamoRIO must intercept and redirect any `sigaction` system call invoked by the application to avoid losing control on signal delivery. As discussed in Section 2.1.3, DynamoRIO must monitor all system calls that free memory regions and flush all cache fragments accordingly to keep the code cache consistent. These calls include `munmap` and `mremap` on Linux and `NtUnmapViewOfSection`, `NtFreeVirtualMemory`, and `NtFreeUserPhysicalPages` on Windows. Additionally, the Windows system call `NtGetContextThread` enables one thread to obtain the context of another thread. DynamoRIO must intercept this call and translate the returned context (see Section 3.3) so that the target thread appears to be executing natively instead of in the code cache. System calls like thread and process creation should also be monitored so the system can maintain control over all execution.

Furthermore, DynamoRIO must hide itself from introspection. For example, on Windows, applications can iterate over all loaded shared libraries using the system call `NtQueryVirtualMemory` [28] to traverse each region of memory and the routine `GetModuleFileName` to find out if a library is present. DynamoRIO detects such queries to its own addresses and modifies the returned data to make the application think that there is no library there. This trick is required to correctly run certain applications, such as `Mozilla`, which install hooks in loaded libraries.

For application queries to Linux's process information pseudo-file system (`/proc`), it is possible to replace the contents of each file (e.g., `/proc/self/maps`) with custom content by monitoring file access system calls. By doing so, we can create the illusion that the application is executing natively. DynamoRIO currently does not implement this feature as we have not seen any problems caused by real applications accessing `/proc`.

3.3 Context Translation

When handling exceptions and signals, DynamoRIO must translate every machine context that the operating system passes to the application, to pretend that the context originated in application code rather than the code cache.

Systems like Valgrind that emulate the application state can simply hand the emulated machine context to the application because that state is always up-to-date. This comes at a performance cost, however, that we consider too high. In DynamoRIO, if the interrupting event requiring translation can be delayed, e.g., on receiving a timer alarm signal on Linux, DynamoRIO will delay the translation until a controlled point outside of the code cache where the application state is easily obtained. For such cases, the delayed event will still occur at a point it might have occurred natively. Otherwise, context translation takes several steps for events that are not

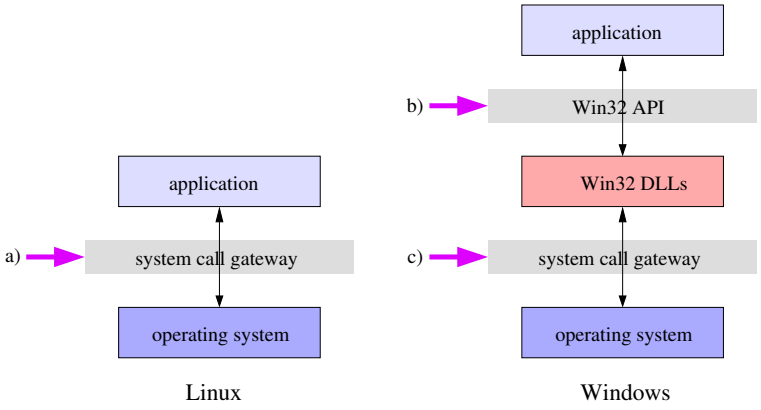

Figure 2. On Linux, the system call interface is the documented method of interacting with the operating system, as shown on the left. On Windows, however, a layer of user libraries makes up the Win32 API, which is the documented interface, while the system call layer is undocumented.

delayable, each bringing the code cache context closer to the state it would contain natively.

The first step is translating the program counter from the code cache to its corresponding application address. One translation method is to store a mapping table for each cache fragment. DynamoRIO's preferred approach, to save memory, is to re-create the cache fragment from application code, and then correlate the code cache address to the recreated cache fragment to obtain the appropriate application address. DynamoRIO rebuilds the cache fragment as though it were encountering new code, making sure to store the original address of each instruction. It then re-applies any instrumentation (this method only works for deterministic instrumentation: a table must be used otherwise). Finally, DynamoRIO walks through the reproduction and the cache fragment in lockstep, until it reaches the target point in the cache fragment. The application address pointed at by the corresponding instruction in the reconstructed cache fragment is the program counter translation.

The second step is ensuring that the registers contain the proper values. In the absence of code transformations, only inserted code for indirect branches by DynamoRIO causes problems here (the load of the indirect branch target could fail, requiring context translation). In this case several registers must have their application values restored to complete the translation. How to recreate registers in the presence of other code transformations is discussed in Section 4.2.

3.4 Error Transparency

Application errors under DynamoRIO must occur as they would natively. There are real-world cases of applications that access invalid memory natively, handle the exception, and carry on. Without error transparency such applications would not work properly. When an error is passed to the application, it must be made to look like it occurred natively, which requires context translation (Section 3.3) and necessary kernel emulation.

An illegal instruction or a jump to invalid memory should not cause DynamoRIO's decoder to crash — rather, the error must be propagated back to the application. The best solution is to have the decoder suppress the exception and stop the basic block construction prior to the faulting instruction. Only if a new basic block is requested whose first instruction faults should it be delivered to the application. This also makes it easier to pass the proper machine context to the application, since the start of a basic block is a clean checkpoint of the application state. To implement error handling for decoding, checking every memory reference prior to accessing it is too expensive. A fault-handling solution is used instead, with a flag set to indicate whether the fault happened while decoding a basic block.

Supporting precise synchronous interrupts in the presence of code modification is challenging. DynamoRIO currently does not do this in every case. As an example, it transforms a far call into a push of the `cs` segment selector, a separate push of the return address, and then a jump. If there is an exception on the selector push or the jump, DynamoRIO need do nothing special. However, if the return address push faults (due to stack overflow, e.g.), the segment selector push should be undone. When executed natively the processor makes the whole sequence atomic. DynamoRIO must emulate this behavior in its fault handler. Today, DynamoRIO does undo the stack pointer adjustment of the selector push, but does not restore the stack value to its prior state. Unlike many transparency corner cases, this one is extreme enough to be ignored for mainstream applications. Systems that perform more aggressive optimizations than DynamoRIO often require hardware support to provide precise interrupts efficiently [18, 23].

Because DynamoRIO uses page protection for code cache consistency, it must distinguish write faults due to its page protection changes from those that would occur natively. When DynamoRIO receives a write fault targeting an area of memory that the application thinks is writable, that fault is guaranteed to belong to DynamoRIO, with all other faults routed to the application.

Error transparency overlaps with heap transparency (Section 2.2.3), stack transparency (stack overflows and underflows, Section 2.2.2), address space transparency (application writes targeting DynamoRIO data, Section 5.2), and context translation (translating contexts presented to application error handlers, Section 3.3).

Error transparency also relates to security vulnerabilities. Many security attacks take advantage of abstraction differences between software conventions and underlying platform enforcement. For example, the classic stack buffer overflow overwrites the return address on the stack. A security attack should work under a dynamic instrumentation system as well as it does natively. Often, address space shifts (see Section 5.2) thwart attacks. These observations lead to the idea of deliberately violating transparency in a tool in order to block security attacks [22].

3.5 Debugging Transparency

A debugger should be able to attach to a process under DynamoRIO's control just like it would natively. This is a reason for

DynamoRIO to not rely on debugging interfaces to control the target application, as only one debugger can be attached to a process at a time. Many debuggers also inject a thread into the debuggee process in order to efficiently access its address space. DynamoRIO would need to identify this thread as a debugger thread, and let it run natively, for full debugging transparency. DynamoRIO currently runs the thread under its control, but most debuggers work fine with DynamoRIO, including `gdb` [19] and the Debugging Tools for Windows [27].

The first transparency guideline serves us well when interacting with a debugger. Stack transparency and data transparency make debugging the application nearly identical to debugging it when running natively, including call stacks. The main difference is, of course, that the program counter and sometimes register values are different. One solution taken by Chaperon, a runtime memory error detector that ships with Insure++ [31], is to modify a debugger like `gdb` to automatically translate the machine context (or at least the program counter) from the code cache to the original application code location.

4. Client Transparency

DynamoRIO is designed to be a general dynamic instrumentation system that supports building custom tools via plug-ins called *clients*. Clients can insert custom instrumentation and execute arbitrary code, including calling standard libraries that can conflict with the application, adding new challenges to transparency enforcement. This section discusses those issues.

4.1 Instrumentation Control

The client writer may not be an expert in transparency, and the client's added instrumentation might have unintended effects on the application's semantics. Similar to Pin and Valgrind, DynamoRIO provides an interface to insert clean calls to C or C++ code, in which operations can be performed. Unlike those other systems, however, DynamoRIO additionally exposes the application's instruction list prior to emitting it into the code cache so that the client can manipulate the application code directly at the granularity of individual instructions and operands. This gives the client more flexibility and power to control the code, allowing for highly efficient instrumentation. However, it leaves the responsibility for transparency in the user's hands.

4.2 State Preservation and Reconstruction

Translating the machine context for an exception or signal handler in the presence of arbitrary client code transformations can become complex. Because DynamoRIO allows the client to manipulate the code directly, it is very difficult for DynamoRIO to recreate the application state by itself. To solve this problem, in the first step of context translation (Section 3.3) DynamoRIO calls on the client to repeat its instrumentation while reconstructing the code fragment. Similarly, DynamoRIO also asks the client for help in the second step to recreate the application register state. This solution only works if the client instrumentation is deterministic. For non-deterministic instrumentation, DynamoRIO stores a mapping table for each code fragment to support state reconstruction.

4.3 Library Usage

If the client calls utility functions from application libraries, the system will encounter the issues discussed in Section 3.1. DynamoRIO uses a custom loader to load separate copies of libraries to avoid sharing libraries with the application. In addition, DynamoRIO exports for client use the methods and resources that it uses itself for better transparency, including memory allocation, reading and writing files, and many others.

4.4 Sideline Threads

The client may want to create new threads to take advantage of multi-core systems for *sideline* computation. DynamoRIO should prevent the application from seeing client threads (Section 3.2) and avoid interfering with applications that monitor all threads in the process. DynamoRIO's client threads on Linux are created in a separate thread group with a separate process identifier from the application, providing isolation from signal delivery, itimers, and introspection.

An instrumentation system should also provide appropriate synchronization to support sideline threads interacting with application threads. DynamoRIO currently only supports simple mutual exclusion locks.

5. Limitations

The further we push transparency, the more difficult it is to implement, while at the same time fewer applications require it. It is challenging and costly to handle all of the corner cases, and many can be ignored if we only want to execute simple programs like the SPEC CPU2006 [39] benchmarks. Yet, for nearly every corner case, there exists an application that depends on it.

We would like to provide *absolute* transparency so that the application cannot distinguish between running inside DynamoRIO and native execution. However, this may not be attainable for some aspects of execution. In this section, we will discuss the limits of transparency in dynamic instrumentation.

5.1 Timing Transparency

Timing transparency is difficult to achieve as it brings efficiency into the transparency equation. The overhead of DynamoRIO is not easy to hide.

Changing the timing of multi-threaded applications can uncover behavior that does not normally happen natively. An example is Microsoft's Removable Storage service, in which under certain timing circumstances when run under DynamoRIO one thread unloads a shared library while another thread returns control to the library *after* it is unloaded. This is not strictly speaking a transparency violation, as the error *could* have occurred without DynamoRIO. Some of these timing violations might also occur natively if some other modification altered the timing, e.g., executing on a different processor.

Another timing related issue is time interrupt transparency. The application may use time interrupts for timeouts or polling. Running in DynamoRIO, the application may experience problems in code that is sensitive to time changes. In addition, the client might use time interrupts for profiling. DynamoRIO must multiplex the client and application uses of the single underlying interrupt system.

5.2 Address Space Transparency

An application bug that accesses invalid memory and generates an exception should do the same thing under DynamoRIO, even if DynamoRIO has allocated its own memory at that location. If the access is a memory write, we can use page protection to detect any inadvertent (or malicious) writes to DynamoRIO memory by the application. DynamoRIO uses this scheme in the context of building a secure execution environment [22]. However, this approach does not work if the access is a memory read, as at least one memory region used by DynamoRIO must be readable throughout the execution, viz., the software code cache. The general solution would be to monitor every application memory access, which would cause prohibitive runtime overhead.

It is also possible for an application to request memory from a specific address that happens to be used by the instrumentation

system. To support this, the system should be able to relocate its own memory and make room for the application's request.

5.3 Resource Limitations

Because the instrumentation system and the application are running in the same process, they share the resources of that process. Thus, any resource limit is a potential source of transparency violations if the resource usage reaches its limit earlier than it would if the application were running natively. For example, Memcheck [33], a memory checking tool built on top of Valgrind, is unable to run certain benchmarks when compiled 32-bit. The *434.zeusmp* benchmark from SPEC CPU2006 [39] contains a 1GB data segment, which is too large for Memcheck to handle [29], for all versions of Memcheck including the most recent version (3.7.0) as of the time of this writing.

In 32-bit architectures where only 4GB or less virtual address space is available, a memory-intensive application may work well natively but crash due to failure to allocate memory when running in a virtual environment. In 64-bit architectures, using up the entire address space is less likely to happen. However, DynamoRIO only requests memory from the first 2GB of the address space, which may cause contention with application memory usage in that range. A memory request might fail if both the application and DynamoRIO keep requesting memory from the same region.

5.4 Client Limitations

We would like to allow a client to modify the application instruction stream in any way it chooses, but unfortunately it may violate transparency. If a client violates transparency, there is often little that DynamoRIO can do about it (though it may be the desired effect for some clients). Application correctness may fail with no chance for recovery. For example, a client that seriously changes basic block control flow can disrupt DynamoRIO's trace creation. DynamoRIO does not disallow this, but it cautions clients to perform dramatic changes to basic block control flow at their own risk.

5.5 Other Limitations

Some types of transparency violations are exceedingly difficult to avoid or disguise. For example, in order to maintain control across callbacks on Windows and system calls that use `sysenter` on Linux, DynamoRIO must modify certain application code to which control is transferred directly by the kernel. The application can read that code and discover whether it is running in DynamoRIO, which can only be prevented by monitoring every memory access.

Another transparency issue is memory modification from another process. One process can change memory in another process via system calls (`NtWriteVirtualMemory` on Windows or `ptrace` on Linux). A kernel module could help here to detect cross-process modifications.

6. Experimental Results

In this section we evaluate the performance impact of several transparency choices made in DynamoRIO. By measuring the tradeoff between correctness and performance, we can understand the cost of a system being fully transparent. We first discuss real application code that violates tempting transparency assumptions.

6.1 Commercial Evaluation

In many cases, performance can be improved for several benchmarks by violating transparency, but the resulting system fails to run all applications. Real-world examples abound, as shown in Table 1. In this section we elaborate on the examples from that table, beginning with code transparency. If true self-modifying code is not handled, Adobe Premiere will not operate correctly as it contains self-modifying code, making this not some never-seen corner case, but actual behavior in a real-world, mainstream application.

```
<generate "ret $0x4" on the stack>
...
104a4: lea    0xc(%ebp), %esp
104a7: popa
104a8: jmp    *-0x28(%esp)
<the jmp target is on the stack:>
 fd04: ret    $0x4
```

Figure 3. Generated code executed beyond the top of the stack in some versions of Microsoft Office.

Further examples of code transparency corner cases are seen in several Windows applications, including versions of the Microsoft Management Console, that execute code on the stack. This code typically takes the form of small trampolines used to create closures for nested functions. As the stack unwinds and is re-extended, these trampolines can be replaced with other trampolines, resulting in different code at the same address. A system that is unable to correctly execute such generated code in memory areas that can be re-used for different code will be unable to run these real-world programs. Similar trampolines are also generated by `gcc` on Linux, and on the heap in other programs.

Large applications such as Microsoft Office contain dynamic languages that generate code during initialization and other normal execution sequences. Another instance of dynamically-generated code comes from applications that store code on disk in a packed format and unpack it at runtime. Virtualdub is an example of a packed application.

Moving on to data transparency, some versions of Microsoft Office contain instances of data being read beyond the top of the stack. In some cases this data is actually code. In one scenario, a return instruction is generated on the stack, the stack pointer is incremented, and then control is transferred to the instruction which is now beyond the top of the stack. Figure 3 shows the code sequence for this. Any system that uses the stack for scratch space runs the risk of breaking transparency for Office.

In addition to containing self-modifying code, Adobe Premiere also contains optimized loops that use the stack pointer as a general-purpose register. Again, any system that assumes it can safely access memory pointed at by the stack pointer may not operate correctly on this mainstream commercial software.

Further examples of data transparency corner cases include glue code used for COM software which manipulates return addresses placed on the stack. Any system that does not preserve the original return addresses will have trouble with COM. Any system that does not hide itself from applications that examine their own address space and loaded libraries may fail to run Mozilla software. A final data transparency example considers condition flags: as will be discussed in Section 6.3, if condition flags are not preserved across indirect branches, even the SPEC CPU benchmarks compiled with the Microsoft compiler will fail.

Regarding concurrency transparency, some Windows applications, including Microsoft Exchange, use a large number of thread-local storage slots and can conflict with a process virtualization system. If Windows thread-local fields are not isolated from client libraries, graphical Windows applications will execute incorrectly, in many cases displaying incorrect pixels and crashing. Other examples of transparency corner cases include applications, like Microsoft Office, that recover from faults as part of normal execution.

Throughout our transparency discussions we have shown which aspects of transparency are necessary by presenting examples of applications whose correctness depends on each aspect. To evaluate

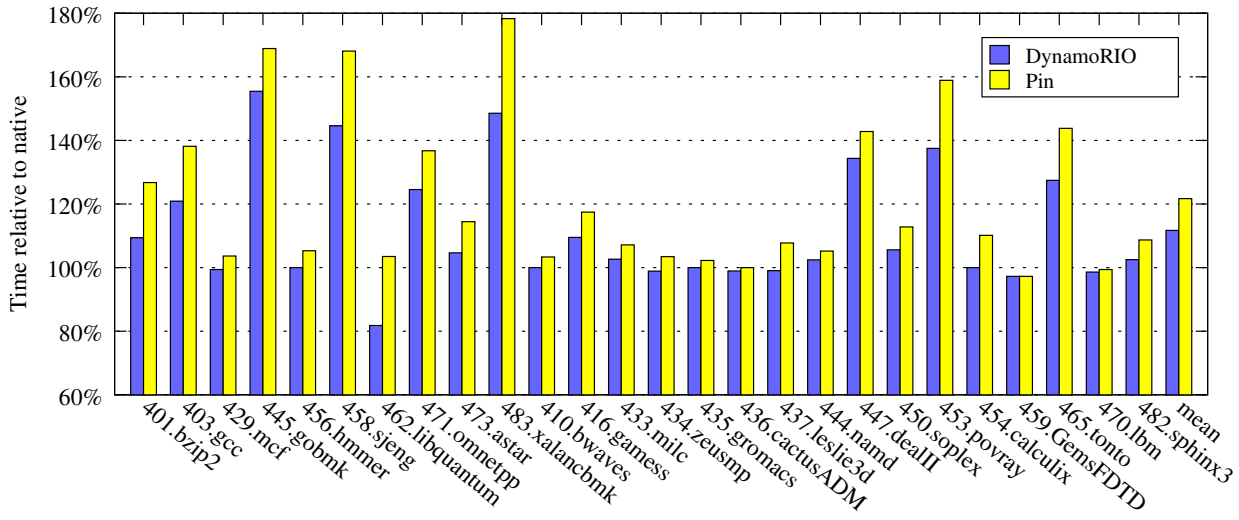

Figure 4. Performance of DynamoRIO and Pin relative to native execution on the SPEC CPU2006 benchmarks compiled as 64-bit.

whether our list of issues is sufficient, we relate our experiences with DynamoRIO in a commercial venture. DynamoRIO was used by Determina in a security product that was installed on tens of thousands of customer machines. When there were application failures, in most cases they were due to resource saturation or clear violations of normal programming practices such as overwriting or patching another application's library entry points (*hooking*) that created problems for many other use cases. We found no additional transparency violations.

6.2 Performance Evaluation

We evaluate the performance of SPEC CPU2006 [39] [1] running in DynamoRIO and Pin. We collected our results on a machine with 8-core Intel Xeon X7550 2 GHz processors, 18M L3 Cache, and 128 GB RAM running 64-bit Linux 2.6.32. We compiled the benchmarks as 64-bit applications. Figure 4 shows the relative performance impact over native execution. The average slowdown of the DynamoRIO base system is merely 11% where Pin has a 21% slowdown. Both systems keep their overhead reasonably low while providing enough transparency to run large applications.

6.3 Application State Transparency

Indirect branch handling is a major source of runtime overhead when executing an application under DynamoRIO. The original single application instruction turns into a lookup routine to locate the correct target code cache fragment. DynamoRIO has tried many ways to minimize overhead without violating transparency, including inlining the lookup routine to avoid a full context switch, using the fastest available method (`lahf` and `seto` on IA-32) to preserve the `eflags` condition flags, and analyzing the code to avoid restoring `eflags` if they are dead in the target.

One experiment we tried that violates transparency in order to gain performance is to simply assume that we do not need to preserve `eflags` across indirect branches. However, we found code in *libc-2.7.so*, shown in Figure 5, that breaks the assumption and causes nearly every application to crash. We instead tried preserving `eflags` over all but `ret` instructions and did manage to run all of our benchmarks. Figure 6 shows the resulting performance

[1] We omit the three benchmarks `400.perlbench`, `464.h264ref`, and `481.wrf` as they fail to run natively.

```
63a3a: lea    0x41(%rip),%rdx # 63a82
...
63a59: lea    -0x21(%rbp),%rax
63a5d: test   %rdi,%rdi
63a60: jmpq   *%rdx
...
63a82: je     63b58
```

Figure 5. An example of code that uses `eflags` across indirect branches.

impact: an average 3%, and as high as 16% on individual benchmarks, performance improvement. However, we found code in several benchmarks in SPEC CPU2000 [38] compiled with Visual C++ 6.0 on Windows, including `vpr` and `eon`, and some versions of large applications like Word and Photoshop, that clearly uses `eflags` across `ret` as follows:

```
call 0x50881d8e
je   0x508eeac7
```

The experiment above shows an example of the tradeoff between performance and transparency: typically one comes at the cost of the other.

7. Related Work

Software virtual machines like Connectix VirtualPC [17], VMWare [12], and Xen [4] perform dynamic binary manipulation, and are able to execute entire operating systems and their workloads. In contrast, process virtualization's goal is to build tools that operate on a single application in a lightweight manner for use on commodity, production platforms. A process virtualization system executes *on top of* the operating system. Perhaps counterintuitively, the technology of building a full system virtualization tool is not sufficient to build a process virtualization tool, which has a completely different set of challenges. When operating on top of the operating system, one must *transparently* operate within the confines of the operating system, intercepting its transfers of control and handling its threads, all the while pretending that one is

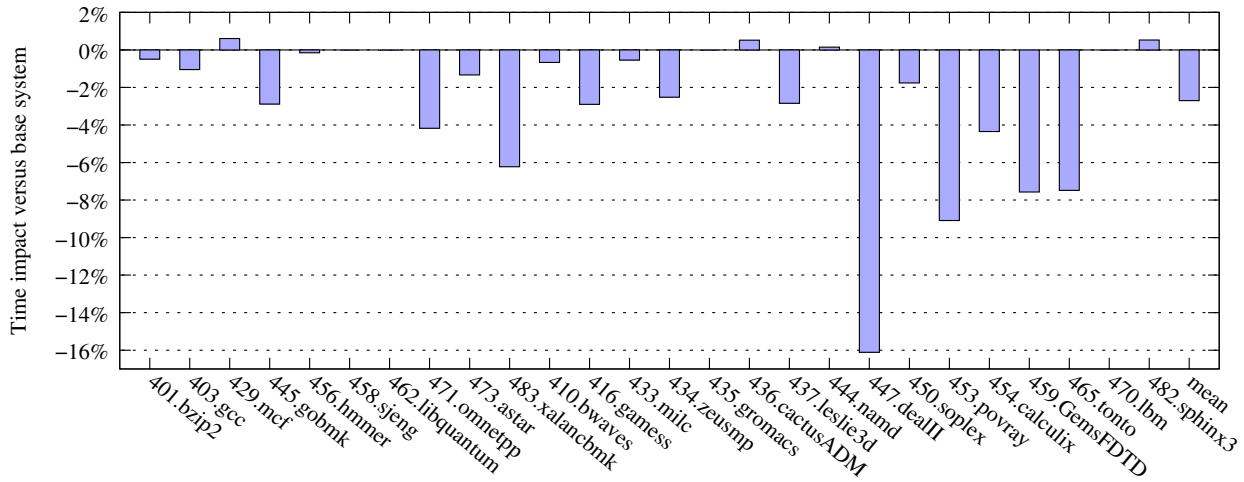

Figure 6. Performance impact of not preserving `eflags` across `ret` instructions.

not occupying the application's address space to avoid interfering with its behavior.

Dynamic instrumentation systems like Pin [25] and Valgrind [30] are implemented with software code caches in a similar manner to DynamoRIO. Other systems like Dyninst [11], Detours [21], and Vulcan [37] modify the original application code in memory by inserting trampolines, which suffer from transparency problems. Additionally, extensive modification of the code quickly becomes unwieldy through these mechanisms, especially in the face of variable-length IA-32 instructions.

Dynamic translators translate from one instruction set to another at runtime, making them similar to instruction set emulators. They include Aries [46] for PA-RISC to IA-64, Walkabout [15] for IA-32 to SPARC, IA32 EL [3] for IA-32 to IA-64, Dynamite [35] for IA-32 to MIPS, and many others including QEMU [5] and HDTrans [36]. Having different host and guest architectures, dynamic translation faces some cross target execution challenges that dynamic instrumentation does not have. For example, an instruction in the source ISA might not have an equivalent corresponding instruction in the target ISA. The host and guest architecture could have different endian-ness. Moreover, misaligned address accesses might cause memory faults on one architecture but not on the other, which adds new challenges to dynamic translation. On the other hand, dynamic translation across architectures often has less pressure for performance and transparency as there is no local native comparison.

Previous work has discussed some aspects of transparency: Pin for Windows [34] discusses transparency issues of Windows system call interception; Shade [16] and Daisy [18] discuss timing and error transparency; Tdb [24] discusses debugging transparency; and Strata [32] uses non-transparent code cache return addresses for performance improvement, accepting the loss of transparency. Machine contexts for signal handlers are translated to their native values in Dynamo [2]. Mojo [13] translates the program counter back to its native value for exception handlers. The transparency of CPU emulators with respect to corner cases of the instruction set has also been explored [26].

To best of our knowledge, there is no prior work that discusses the full range of transparency issues in the design and implementation of a dynamic instrumentation system.

8. Conclusion

DynamoRIO is a powerful runtime code manipulation system that provides a virtual execution environment within which an unmodified application can be monitored and controlled while it executes. *Transparency* is one major challenge when building such a system. Unlike SPEC CPU and other relatively simple benchmarks, many commercial applications rely on a pristine execution environment and a precise ABI. In order to run all of these applications, DynamoRIO must be fully transparent and cannot ignore seemingly obscure corner cases. In this paper, we identified a necessary set of transparency requirements for running mainstream Windows and Linux applications. We discussed possible solutions to each issue, evaluated tradeoffs between different choices, and identified cases where maintaining transparency is not practically achievable. We hope this paper can be used as a guide for building transparent dynamic instrumentation tools and other similar process virtualization systems.

References

[1] DynamoRIO dynamic instrumentation tool platform, Feb. 2009. `http://dynamorio.org/`.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent runtime optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*, pages 1–12, June 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 International Symposium on Microarchitecture*, 2003.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauery, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating System Principles (SOSP '03)*, pages 164–177, Oct. 2003.

[5] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference*. USENIX Association, 2005.

[6] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, M.I.T., Sept. 2004.

[7] D. Bruening and S. Amarasinghe. Maintaining consistency and bounding capacity of software code caches. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '05)*, pages 74–85, Mar. 2005.

[8] D. Bruening and Q. Zhao. Practical memory checking with Dr. Memory. In *The International Symposium on Code Generation and Optimization*, Chamonix, France, Apr 2011.

[9] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for Windows. In *Proceedings of the 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, pages 19–30, Dec. 2001.

[10] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '03)*, pages 265–275, Mar. 2003.

[11] B. R. Buck and J. Hollingsworth. An API for runtime code patching. *Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[12] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP '97)*, pages 143–156, Oct. 1997.

[13] W. Chen, S. Lerner, R. Chaiken, and D. M. Gillies. Mojo: A dynamic optimization system. In *Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, pages 81–90, Dec. 2000.

[14] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *Proceedings of the Proceedings of the 11th IEEE Symposium on Computers and Communications (ISCC '06)*, pages 749–754, 2006.

[15] C. Cifuentes, B. Lewis, and D. Ung. Walkabout – a retargetable dynamic binary translation framework. In *Proceedings of the 4th Workshop on Binary Translation*, Sept. 2002.

[16] R. F. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical Report UWCSE 93-06-06, University of Washington, June 1993.

[17] Connectix. Virtual PC. `http://www.microsoft.com/windows/virtualpc/default.mspx`.

[18] K. Ebcioglu and E. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA '97)*, pages 26–37, June 1997.

[19] GDB. The GNU Project Debugger. `http://www.gnu.org/software/gdb/gdb.html`.

[20] J. D. Hiser, D. Williams, W. Hu, J. W. Davidson, J. Mars, and B. R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 61–73, Washington, DC, USA, 2007. IEEE Computer Society.

[21] G. Hunt and D. Brubacher. Detours: Binary interception of win32 functions. In *Proceedings of the USENIX Windows NT Workshop*, pages 135–144, July 1999.

[22] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Aug. 2002.

[23] A. Klaiber. The technology behind Crusoe processors. Transmeta Corporation, Jan. 2000. `http://www.transmeta.com/crusoe/download/pdf/crusoetechwp.pdf`.

[24] N. Kumar, B. Childers, and M. L. Soffa. Tdb: A source level debugger for dynamically translated programs. In *the Sixth International Symposium on Automated And Analysis-Driven Debugging (AADEBUG)*, 2005.

[25] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, pages 190–200, June 2005.

[26] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing CPU emulators. In *Proceedings of the 2009 International Conference on Software Testing and Analysis (ISSTA)*, pages 261–272. ACM, July 2009. Chicago, Illinois, USA.

[27] Microsoft Debugging Tools for Windows. `http://www.microsoft.com/whdc/devtools/debugging/default.mspx`.

[28] G. Nebbett. *Windows NT/2000 Native API Reference*. Macmillan Technical Publishing, Indianapolis, IN, 2000.

[29] N. Nethercote. Spec2006 zeusmp and dealII on Valgrind 3.3.0, 2008. (These failures are still present in version 3.7.0.) `http://article.gmane.org/gmane.comp.debugging.valgrind/7947/match=spec2006`.

[30] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, pages 89–100, June 2007.

[31] Parasoft. Insure++. `http://www.parasoft.com/jsp/products/insure.jsp?itemId=63`.

[32] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. L. Soffa. Reconfigurable and retargetable software dynamic translation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '03)*, pages 36–47, Mar. 2003.

[33] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX Annual Technical Conference*, pages 2–2, 2005.

[34] A. Skaletsky, T. Devor, N. Chachmon, R. Cohn, K. Hazelwood, V. Vladimirov, and M. Bach. Dynamic program analysis of microsoft windows applications. In *International Symposium on Performance Analysis of Software and Systems (ISPASS)*, 2010.

[35] J. Souloglou. *A Framework for Dynamic Binary Translation*. PhD thesis, University of Manchester, 1996.

[36] S. Sridhar, J. S. Shapiro, E. Northup, and P. P. Bungale. HDTrans: An open source, low-level dynamic instrumentation system. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE '06)*, pages 175–185, New York, NY, USA, 2006. ACM Press.

[37] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, Apr. 2001.

[38] Standard Performance Evaluation Corporation. SPEC CPU2000 benchmark suite, 2000. `http://www.spec.org/osg/cpu2000/`.

[39] Standard Performance Evaluation Corporation. SPEC CPU2006 benchmark suite, 2006. `http://www.spec.org/osg/cpu2006/`.

[40] Q. Zhao, J. E. Sim, L. Rudolph, and W. Wong. Dep: Detailed execution profile. In *International Conference on Parallel Architectures and Compilation Techniques*, Seattle, WA, Sep 2006.

[41] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong. Ubiquitous memory introspection. In *International Symposium on Code Generation and Optimization*, San Jose, CA, Mar 2007.

[42] Q. Zhao, I. Cutcutache, and W.-F. Wong. Pipa: Pipelined profiling and analysis on multi-core systems. In *The International Symposium on Code Generation and Optimization*, Boston, MA, Apr 2008.

[43] Q. Zhao, R. M. Rabbah, S. P. Amarasinghe, L. Rudolph, and W.-F. Wong. How to do a million watchpoints: Efficient debugging using dynamic instrumentation. In *Proceedings of the 17th International Conference on Compiler Construction (CC '08)*, pages 147–162, 2008.

[44] Q. Zhao, D. Bruening, and S. Amarasinghe. Umbra: Efficient and scalable memory shadowing. In *The International Symposium on Code Generation and Optimization*, Toronto, Canada, Apr 2010.

[45] Q. Zhao, D. Koh, S. Raza, D. Bruening, W.-F. Wong, and S. Amarasinghe. Dynamic cache contention detection in multi-threaded applications. In *The International Conference on Virtual Execution Environments*, Newport Beach, CA, Mar 2011.

[46] C. Zheng and C. Thompson. PA-RISC to IA-64: Transparent execution, no recompilation. *IEEE Computer*, 33(3):47–53, Mar. 2000.