PinOS: A Programmable Framework for Whole-System Dynamic Instrumentation

Prashanth P. Bungale
Intel Corporation, Hudson, MA
and Harvard University, Cambridge, MA
Email: prash@eecs.harvard.edu

Chi-Keung Luk
Intel Corporation, Hudson, MA
Email: chi-keung.luk@intel.com

Abstract

PinOS is an extension of the *Pin* dynamic instrumentation framework for whole-system instrumentation, i.e., to instrument both kernel and user-level code. It achieves this by interposing between the subject system and hardware using virtualization techniques. Specifically, PinOS is built on top of the Xen virtual machine monitor with Intel VT technology to allow instrumentation of unmodified OSes. PinOS is based on software dynamic translation and hence can perform pervasive fine-grain instrumentation. By inheriting the powerful instrumentation API from Pin, plus introducing some new API for system-level instrumentation, PinOS can be used to write system-wide instrumentation tools for tasks like program analysis and architectural studies. As of today, PinOS can boot Linux on IA-32 in uniprocessor mode, and can instrument complex applications such as database and web servers.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging - code inspections and walk-throughs, debugging aids, tracing; D.3.4 [*Programming Languages*]: Processors - compilers, incremental compilers

General Terms Performance, Experimentation, Languages

Keywords Whole-system, dynamic instrumentation, program analysis tools, binary translation, virtualization

1. Introduction

As the complexity of computer systems continues to grow, it is increasingly important to have tools that can provide detailed observation of system behavior. *Software instrumentation* frameworks [1] like ATOM [27], Dynins [5], Valgrind [23], Pin [19], and Nirvana [3], have shown to be very useful for developing tools that do program analysis, performance profiling, and architectural simulation. In spite of their success, a major limitation of these frameworks is that they instrument only user-level code—no kernel-level code is instrumented. This limitation becomes significant in applications such as database and web servers, where a majority of execution time is spent inside the operating system.

On the other hand, whereas there do exist a few instrumentation frameworks (Kerninst [28], Dtrace [7], and Dprobes [21]) that can instrument operating systems, we find that these existing whole-system instrumentation frameworks are limited in two ways. First, since they all use the *probe-based* approach of introducing *probes* to perform instrumentation *in-place* in the binary (in contrast to the *software dynamic translation* approach used by user-level instrumentation frameworks like Valgrind [23], Pin [19], and Nirvana [3]), they are generally not suitable for pervasive fine-grain instrumentation (e.g., instrumenting every single instruction ever executed). This limitation prevents using these frameworks for developing tools like memory-leak checkers [23] and instruction tracers [3]. Second, all these frameworks are OS-specific, as their instrumentation engines are built in to the OSes being instrumented (Kerninst and Dtrace for Solaris, Dprobes for Linux). As a result, tools developed for one of these operating systems do not work on another.

In this study, we have extended a popular user-level instrumentation framework, Pin [19], to achieve *whole-system* instrumentation. This extension is called *PinOS*. It uses the same software dynamic translation approach as Pin does, and is therefore able to perform pervasive fine-grain instrumentation. PinOS inherits the same rich instrumentation API [15] from Pin, and adds some new API specific to system-wide instrumentation. By design, PinOS works as a software layer running *underneath* the instrumented OS and does *not* require making any changes to it. This makes it possible for PinOS to instrument unmodified OSes. In addition, running as a separate layer helps achieve two desirable properties of instrumentation: *isolation* and *transparency*. With isolation, the instrumented OS and the instrumentation engine do not share any code and hence avoid any potential re-entrancy problems. With transparency, the execution behavior of the instrumented OS is the same as what we would observe without instrumentation.

The PinOS approach of running underneath the instrumented OS has a major technical challenge: PinOS cannot use the system facilities (mainly memory allocation and I/O services) provided by the instrumented OS. And of course, PinOS like any other software, needs these system facilities as it runs. Implementing these facilities from scratch inside PinOS would be a tremendous task. Fortunately, we observe that *virtualization* offers a neat solution to this challenge. By running the OS that we want to instrument as a *guest OS* inside a virtual machine (instead of on bare hardware), PinOS can obtain system facilities from the *host* OS. Based on this observation, we have built PinOS on top of the Xen virtual machine monitor [14] (with our own modifications). In order to run unmodified (legacy) OSes as guests, we use Xen with the Intel hardware virtualization support on x86 (VT-x) [22]. As of today, PinOS can

[1] We call them instrumentation *frameworks* instead of instrumentation *tools* because each of them is used to build multiple tools using its instrumentation API.

VEE'07, June 13–15, 2007, San Diego, California, USA.
Copyright © 2007 ACM 978-1-59593-630-1/07/0006. . . $5.00.

```
FILE * trace;

// Print a memory write record
VOID RecordMemWrite(VOID * ip,
    VOID * va, VOID * pa, UINT32 size)
{
    Host_fprintf(trace,"%p: W %p %p %d\n",
                        ip, va, pa, size);
}

// Called for every instruction
VOID Instruction(INS ins, VOID *v)
{
    if (INS_IsMemoryWrite(ins))
        INS_InsertCall(
            ins, IPOINT_BEFORE,
            AFUNPTR(RecordMemWrite),
            IARG_INST_PTR, IARG_MEMORYWRITE_VA,
            IARG_MEMORYWRITE_PA,
            IARG_MEMORYWRITE_SIZE, IARG_END);
}

int main(int argc, char *argv[])
{
    PIN_Init(argc, argv);
    trace = Host_fopen("atrace.out", "w");
    INS_AddInstrumentFunction(Instruction, 0);
    // Never returns
    PIN_StartProgram();
    return 0;
}
```

Figure 1. A Pintool for tracing memory writes with PinOS.

instrument unmodified IA32 Linux guests running inside Xen, including instrumenting the entire booting process and real-life applications like database and web servers. We are also planning to instrument Windows guests in the future.

This paper focuses on the design and implementation issues we have encountered in the project and our solutions to them. We begin with a simple PinOS tool in Section 2. Section 3 gives a high-level overview of the PinOS architecture. We then dive into the design issues of the two portions of PinOS: the VMM driver in Section 4, followed by the instrumentation layer in Section 5. To demonstrate the practicality of PinOS, we have ported two popular instrumentation tools (one is a code profiler and another is a cache simulator) from Pin to PinOS, and we present the results of using these tools in Section 6. We relate our work to others' in Section 7 and conclude in Section 8.

2. A Sample PinOS Tool

Prior to discussing how PinOS works, we give a brief introduction to the PinOS instrumentation API through a simple example. PinOS inherits the same API from Pin, which allow a tool to observe all the architectural state of the instrumented system, including the contents of registers, memory, and control flow. In addition, it provides new API for whole-system instrumentation, such as physical addresses of memory accesses and callbacks for events like page faults and interrupts. Following the ATOM [27] model, a Pintool is a C/C++ program with two sets of procedures added for instrumentation. One set is called *analysis routines*, which define what to do with instrumentation, while the other set is called *instrumentation routines*, which determine where to place calls to analysis routines. The arguments to analysis routines can be architectural state or constants.

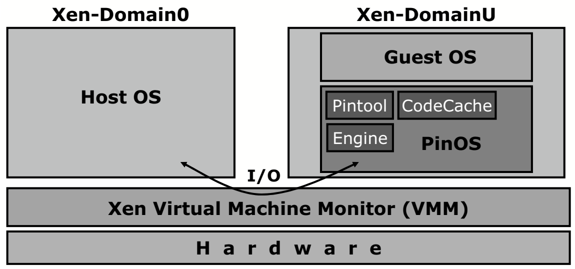


Figure 2. PinOS Architecture

Figure 1 lists the code of a Pintool that traces both the virtual and physical addresses and size of every memory write executed on the system. The `main` procedure initializes PinOS and opens a file on the *host OS* via `Host_fopen` to hold the tracing file. Writing results on the host OS instead of the guest OS helps achieve instrumentation isolation and transparency. We then register the procedure called `Instruction`, and tell PinOS to start executing the system. Whenever a memory write instruction is encountered the first time, PinOS inserts a call to `RecordMemWrite` *before* the instruction (specified by the argument `IPOINT_BEFORE` to `INS_InsertCall`), passing the instruction pointer (specified by `IARG_INST_PTR`), virtual address and physical address for the memory operation (specified by `IARG_MEMORYWRITE_VA` and `IARG_MEMORYWRITE_PA`, respectively), and number of bytes written (specified by `IARG_MEMORYWRITE_SIZE`). Every time an instruction that writes memory executes, it calls `RecordMemWrite` and records information in the trace file on the host OS via `Host_fprintf`.

Note that the user-level Pin version of the above Pintool is nearly identical to the PinOS version except that file I/O functions are replaced by the corresponding `Host_xxx` functions and that `IARG_MEMORYWRITE_PA` is only available in PinOS. This demonstrates the ease of porting existing tools from Pin to PinOS.

3. Architecture

As PinOS strives to observe and instrument every instruction executed in a subject system, it must interpose between the subject operating system and the underlying hardware. We use virtualization technology to introduce the PinOS layer below the guest system layer. Figure 2 illustrates the architecture of the PinOS framework. We run a modified version of the Xen [14] virtual machine monitor (VMM) on bare hardware. In order to run *unmodified* guest operating systems, the hardware itself must be equipped with Intel's hardware virtualization support on x86 (VT-x).

In Xen terminology, virtual machines are called domains; there exists one privileged virtual machine called the *host domain* or *Dom0*, and all other virtual machines are called *guest domains* or *DomU*. We run the subject system, including its operating system and all its applications in a guest domain. Inside this guest domain, we transparently inject PinOS underneath the subject operating system, so that PinOS will be in a position to observe every instruction ever executed in the subject system.

We picked Xen for several compelling reasons: it is a robust, widely-deployed, simple and efficient VMM; it is available via an open-source license; last, but not the least, it provides debugging support for our development efforts. We introduce a PinOS driver inside the Xen VMM by modifying Xen. In this driver, in addition to introducing new hypercalls (i.e., calls into Xen VMM) for PinOS's own use, we exploit a lot of the support that is already implemented in Xen (such as shadow paging and I/O request redirection [14]) and we adapt them for our purposes.

On top of Xen is the PinOS instrumentation layer, which consists of three major components: *Code Cache*, *Instrumentation Engine*, and *Pintool*. PinOS maintains a cache of guest code translations in the code cache component. The technique of *software dynamic translation* is employed to achieve two goals:

- *Always maintain control*: Every control-transfer instruction is translated so that control is either transferred to an existing translated version of the target in the code cache or to the instrumentation engine so that the target can be translated and put into the code cache.
- *Perform instrumentation*: As new code is encountered and translated, the user-programmed Pintool performs the specified instrumentation on the guest code. For instance, a Pintool may demand the instrumentation of every memory load and store instruction in order to simulate cache behavior.

The execution of PinOS-based instrumentation takes the form of an infinite loop of alternating executions of the instrumentation engine and the code cache. The instrumentation engine is responsible for code compilation, code cache maintenance, instruction emulation, invocation of instrumentation routines in the Pintool, etc. The code cache executes translated guest code and also appropriately invokes user-provided analysis routines.

Pin [19] compiles from the given ISA directly into the same ISA (i.e., IA32 to IA32) without going through an intermediate format, and the compiled code is stored in a software-based code cache. Only code residing in the code cache is executed – the original code is never executed. An application is compiled one trace at a time. A trace is a straight-line sequence of instructions which terminates at one of the conditions: (i) an *unconditional* control transfer (branch, call, or return), (ii) a pre-defined number of *conditional* control transfers, or (iii) a pre-defined number of instructions have been fetched in the trace. In addition to the last exit, a trace may have multiple side-exits (the conditional control transfers). Each exit initially branches to a *stub*, which re-directs the control to the engine. The engine determines the target address (which is statically unknown for indirect control transfers), generates a new trace for the target if it has not been generated before, and resumes the execution at the target trace. PinOS retains the compilation techniques that Pin uses including techniques to improve the quality of translated code, such as trace linking, register re-allocation, and inlining analysis routines [19].

Finally, we considered other architectural alternatives for PinOS. The main design question is *where* the PinOS instrumentation layer should be placed with respect to the Xen VMM. Options include putting PinOS as part of the VMM, in the same domain as the guest OS (i.e. the current design), or in another domain. We chose the current design for three reasons: (1) it matches the existing design of user-level Pin so that we do not have to significantly restructure our code, (2) it potentially has better performance because the guest OS and PinOS sit in the same domain, hence minimizing cross-domain operations, and (3) separating PinOS instrumentation from the Xen VMM makes it easier to port PinOS to another VMM in the future.

4. VMM Driver

We introduce a PinOS driver inside the Xen VMM by modifying Xen to provide PinOS four main functionalities: memory stealing, attach/detach, I/O services, and time virtualization. We discuss these design issues in this section.

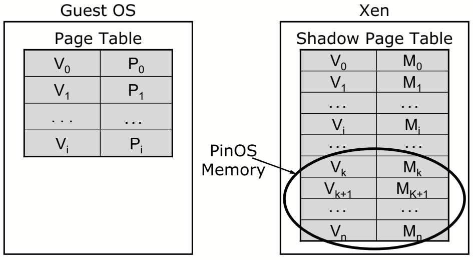


Figure 3. Our memory stealing technique. V_0 to V_n are virtual pages on the guest; P_0 to P_i are physical pages on the guest; M_0 to M_n are physical pages on the host (also known as machine pages).

4.1 Memory Stealing

PinOS requires its own memory for several purposes: the code cache, PinOS executable, stack and heap, and I/O buffers. PinOS needs to steal both physical memory pages and a part of the virtual address space from the guest in order to operate within the guest domain.

In a Xen system, the VMM maintains a shadow of the virtual machine's memory-management data structure, called the *shadow page table* so that it can control which pages of the machine's memory are available to the virtual machine. Thus there are two layers of physical pages: *guest-physical* pages and *machine pages*. The guest domain's page table maps virtual pages to guest-physical pages, i.e., to what it *thinks* are physical pages. The hardware uses Xen's shadow page table for memory address translation so that the VMM can always control what memory each virtual machine is using. Xen assigns real machine pages to guest-physical pages, and the shadow page table takes care of mapping virtual pages directly to machine pages. One such shadow page table is maintained for every guest address space encountered.

We exploit Xen's shadow paging to achieve memory stealing for PinOS. To steal *physical* memory, we modify Xen to pre-allocate a separate range of machine pages for PinOS. These pages are unknown to the guest operating system, thus are completely transparently stolen.

We also need to steal some portion of the guest *virtual* address space. Our current strategy is to statically steal part of the guest's kernel address space (to minimize chances of conflicts), and to detect and report any guest OS mapping activity in the stolen space. We modify Xen to add, in the shadow page table, mappings from these stolen virtual pages to the pre-allocated machine pages. Again, these mappings have to be propagated to every shadow page table used by Xen. Our scheme is illustrated in Figure 3, where M_k to M_n are the pre-allocated machine pages on the host for PinOS and V_k to V_n are the virtual pages we steal from the guest. We have not encountered any address space conflicts so far with this strategy. In the future, it may be desirable to have a more dynamic approach of *relocating* the stolen address space portion as per dynamic availability, although we don't expect this to be much of an issue with 64-bit address spaces.

4.2 Attach / Detach

PinOS can be enabled on the *entire run* of a guest system. However, PinOS also has the ability to attach itself to a running guest system only when the user is interested in performing instrumentation and to detach subsequently. This facility has two major advantages:

- The user can avoid the overhead of PinOS being active during OS boot and shutdown every time she wants to perform whole-system instrumentation on an application run.
- Enabling PinOS on the entire run of the guest system may pollute the instrumentation data collection, or at the very least, may require resetting of instrumentation data collection.

We add hypercalls in Xen to support PinOS attach and detach functionality. Upon an attach request hypercall, typically made through a tool provided on the host operating system, we perform the following steps:

1. Save key parts of processor state of the '*attachee*' guest domain into virtual state maintained by PinOS. The processor state here includes, in addition to the basic state such as general purpose registers, instruction pointer, stack pointer, etc., all the state that PinOS further virtualizes within the Xen guest domain such as the segmentation state, interrupt descriptor table, etc.
2. Set the guest domain state to the initial state of PinOS so that PinOS hijacks control upon execution of that guest domain.
3. Upon gaining control, PinOS starts instrumenting the guest system from the current program counter (PC), i.e., the PC at which attach was performed.

The reading and writing of guest domain state above may involve some hidden processor state, i.e., processor registers that are not visible at the ISA interface. A typical example of such state is the segment descriptor cache maintained alongside each segment register. To overcome this challenge, we exploit Intel VT support, which essentially makes these invisible registers visible to the VMM.

Detach is implemented similarly where the guest domain state is restored with the virtual guest system state that PinOS maintains so that execution can continue natively without the involvement of PinOS.

4.3 I/O Services

PinOS requires I/O services for two purposes: one is to generate log files for debugging and the other is to provide file I/O operations to Pintools. As illustrated in Figure 2, PinOS routes all its I/O requests to the host OS, which actually performs the requests and then sends the results back to PinOS. We set up a daemon process on the host domain (i.e. Domain-0 in Figure 2) to periodically poll and process requests from PinOS. The channels between the host and guest domains are implemented via memory regions that are mapped into both domains.

4.4 Time Virtualization

Depending on the kind of instrumentation being performed by PinOS tools, the run-time instrumentation overhead could be significant. Sometimes, this could slow down the guest system to an extent that the guest will complain about "timeout". Our current strategy is to reduce the Virtual Programmable Interrupt Timer (VPIT) frequency delivered to the guest from Xen by 20x (the VPIT is a compile-time constant in Xen). While this strategy works for the PinOS tools that we currently have, we realize that an adaptive scheme would be needed once we have more tools with a wider range of instrumentation overheads. This is currently ongoing work.

5. Instrumentation Layer

This section discusses design issues relating to the instrumentation layer introduced by PinOS beneath the guest OS in a guest domain. We describe the modifications we made to user-level Pin for system-wide instrumentation along the way.

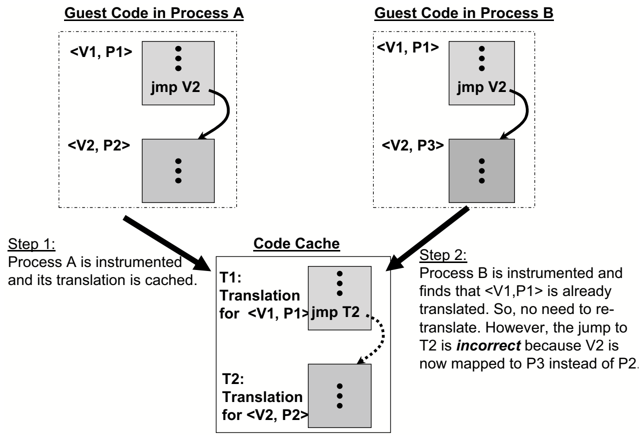
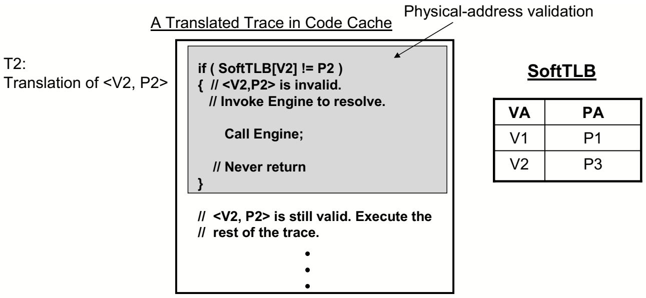

Guest Code in Process A
Guest Code in Process B
<V1, P1>
jmp V2
<V2, P2>
<V1, P1>
jmp V2
<V2, P3>
Step 1:
Process A is instrumented and its translation is cached.
Code Cache
T1:
Translation for <V1, P1>
jmp T2
T2:
Translation for <V2, P2>
Step 2:
Process B is instrumented and finds that <V1,P1> is already translated. So, no need to re-translate. However, the jump to T2 is ***incorrect*** because V2 is now mapped to P3 instead of P2.

Figure 4. A problem in the $\langle VA, PA\rangle$ code-cache indexing scheme with trace linking.



```
if ( SoftTLB[V2] != P2 )
{  // <V2,P2> is invalid.
   // Invoke Engine to resolve.

      Call Engine;

    // Never return
}

//  <V2, P2> is still valid. Execute the
//  rest of the trace.
```

VA	PA
V1	P1
V2	P3

Figure 5. Using a run-time physical-address test solves the problem in Figure 4.

5.1 Code Cache Indexing and Sharing

All existing user-level instrumentation frameworks, including Pin, use application *virtual addresses* to index their code caches. However, for system-wide instrumentation, virtual addresses alone are insufficient to distinguish code as different applications may be assigned the same virtual addresses in different address spaces. Therefore, code-cache indexing in PinOS needs a new scheme that can differentiate address spaces. We have studied two such schemes described below.

The first scheme uses the Virtual Address (VA) and *Address-Space Identifier (ASID)* pair as the code-cache index (denoted as $\langle VA, ASID\rangle$). On the x86 architecture, the ASID is the physical address of the page table used by the current process and is always stored in the control register `cr3`. The main advantage of this scheme is the ease of implementation (as the ASID can be obtained by simply reading `cr3`), while the main disadvantage is that the code cache cannot be shared among address spaces. Thus, the same code has to be retranslated over and over again in different address spaces, incurring unnecessary run-time and code-cache space overhead.

The second scheme uses the Virtual Address and *Physical Address (PA)* pair as the code-cache index (denoted as $\langle VA, PA\rangle$). Compared against the $\langle VA, ASID\rangle$ scheme, this scheme allows sharing the code cache among address spaces as long as the $\langle VA, PA\rangle$ remain the same in different address spaces, which is

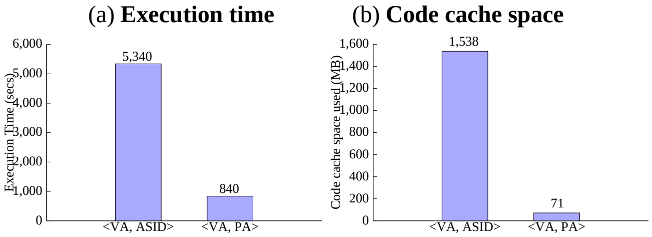

(a) **Execution time** (b) **Code cache space**

Figure 6. Execution time and code cache space comparison of the two indexing schemes (data based on PinOS using these two schemes to boot FC4 Linux).

the case for most kernel-level code (e.g., interrupt handlers, schedulers etc). Nevertheless, this scheme has the following technical challenges.

First, we need a way to get physical addresses efficiently. While we can always make a hypercall to Xen to get the physical address from a virtual address, it is too expensive to do this frequently as each hypercall takes several thousand cycles in the current Xen/VT implementation. We address this challenge by maintaining a Software TLB (called *SoftTLB*) inside PinOS which caches the VA-to-PA mappings of code that PinOS has translated. The SoftTLB is kept coherent using the techniques described in Section 5.4. As a result, most requests of getting physical address translation are satisfied by the SoftTLB; we only need call Xen on SoftTLB misses.

The second challenge of the $\langle VA, PA\rangle$ scheme is its interaction with *trace linking* [1], an optimization used in almost every dynamic binary translator. Trace linking allows jumping from one trace in the code cache to another without going through the instrumentation engine and hence improves performance. The problematic interaction is illustrated using the example in Figure 4. Process A is first executed, and the translations based on the mappings $\langle V1, P1\rangle$ and $\langle V2, P2\rangle$ are generated in the code cache as T1 and T2, respectively. Trace linking connects T1 and T2 via a direct jump. Later on, Process B is executed and its corresponding page mappings are $\langle V1, P1\rangle$ and $\langle V2, P3\rangle$. Since the translation of $\langle V1, P1\rangle$ is already cached, there is no need to re-translate this trace, and so we execute T1 out of the code cache directly. However, as V2 is now mapped to P3 instead of P2, the jump from T1 to T2 is wrong. Instead, we should generate a new translation for $\langle V2, P3\rangle$ and transfer control to it after executing T1.

To solve this problem, we add a run-time test at the entry of each trace to validate the physical address upon which the generation of that particular trace relied. The trace is used only if the actual physical address matches the assumed physical address. How this technique solves the problem in Figure 4 is illustrated in Figure 5, where a physical-address validation test is added to the entry of trace T2. The guest page mappings of Process B are shown in the SoftTLB. In this case, since the assumed physical address (P2) is not equal to the actual physical address (P3), the test will fail and we will enter the instrumentation engine, which will in turn transfer control to a trace corresponding to $\langle V2, P3\rangle$.

We have implemented both the $\langle VA, ASID\rangle$ and $\langle VA, PA\rangle$ schemes, and measured their effectiveness in terms of run-time overhead and code-cache space consumed. Figure 6 shows how much time and code-cache space PinOS takes to boot FC4 Linux under each scheme (the machine used is the "Setup A" described in Table 1). The $\langle VA, PA\rangle$ scheme is the obvious winner as it is faster by 6 times and generates 20 times less code than the $\langle VA, ASID\rangle$ scheme. These results clearly demonstrate the importance of sharing code cache among address spaces.

5.2 Interrupt and Exception Emulation

Interrupts and exceptions are emulated in PinOS for two reasons. The first reason is to ensure that PinOS maintains full control of the guest. For instance, if we do not emulate interrupts, PinOS will lose control after process preemption, which is typically triggered by timer interrupts. The second reason is for instrumentation transparency. For instance, a guest interrupt handler can inspect some registers to tell what the current thread is. PinOS has to present the guest register context (not the hardware register context) to the guest interrupt handler in order for it to find the correct thread.

To emulate interrupts and exceptions, PinOS first installs its own interrupt/exception handlers in the Interrupt Descriptor Table (IDT) so that all interrupts and exceptions are intercepted. Then interrupts and exceptions are handled using different approaches, as explained below.

5.2.1 Interrupt Emulation

Interrupts are *asynchronous* in nature—they could happen at any time during the execution. The most common ones are timer interrupts. The issue with interrupts is that they could happen while we are executing something other than the guest code (e.g., we may be executing the instrumentation engine or the Pintool when a timer interrupt happens). We therefore cannot deliver the interrupt at those points. Moreover, we cannot simply present to the guest interrupt handler the hardware register context when the interrupt happened; instead, we must present a valid guest register context (these two contexts are different because the hardware executes code-cache traces which are positioned at different addresses than the original guest code). Therefore, when we receive an interrupt, we put it on to a queue instead of delivering it to the guest immediately. We also add a run-time test at the entry of each trace to check if the queue has any pending interrupt. When a trace, say T, is executed out of the code cache and there is a pending interrupt, the run-time test will return true. We will call the engine to instrument the guest handler for that pending interrupt. We present the guest register context at the entry of T to the guest interrupt handler, effectively creating an illusion that the interrupt happened at the entry of T. Finally, we transfer control to the instrumented handler.

5.2.2 Exception Emulation

Unlike interrupts, exceptions are *synchronous*—they cannot be delayed and must be handled immediately. The most common exceptions are page faults. Guest exceptions could happen while in the instrumentation engine or in the code cache. The former could be instruction page faults that happen when the instrumentation engine fetches some guest code that has not been paged in by the guest OS. The latter could be data page faults while the guest code is executed out of the code cache. In either case, we need to recover the guest register context from the hardware register context at the exception. Then we instrument the corresponding guest exception handler, using the guest register context we just recovered, and finally transfer control to the instrumented handler. In addition, we have to ensure that our exception emulation is both *precise* and *faithful*.

Precise exception emulation means the following. If PinOS translates one original guest instruction into a multiple-instruction sequence and an exception happens in the middle of the sequence, the guest register context that should be presented to the exception handler is the one at the beginning of the sequence, not the one in the middle. Essentially, because of code translation, PinOS may introduce *pseudo instruction boundaries*, i.e., instruction boundaries that are present on the hardware during execution but are not expected by the guest semantics. If a page fault happens at such a pseudo instruction boundary, we must first rollback the effect of the instructions since the most recent guest instruction boundary

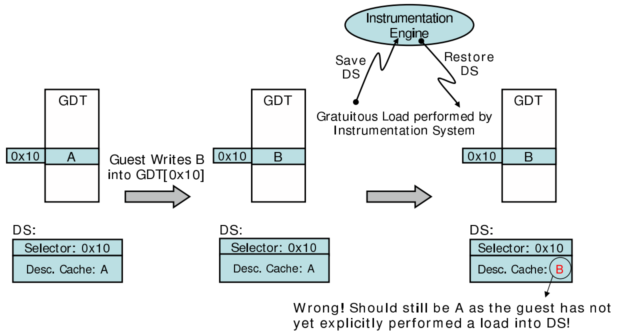

Instrumentation Engine
Save DS
Restore DS
Gratuitous Load performed by Instrumentation System
GDT
0x10
A
Guest Writes B into GDT[0x10]
GDT
0x10
B
GDT
0x10
B
DS:
Selector: 0x10
Desc. Cache: A
DS:
Selector: 0x10
Desc. Cache: A
DS:
Selector: 0x10
Desc. Cache: B
Wrong! Should still be A as the guest has not yet explicitly performed a load into DS!

Figure 7. An illustration of the *Irreversible Segmentation* Problem.

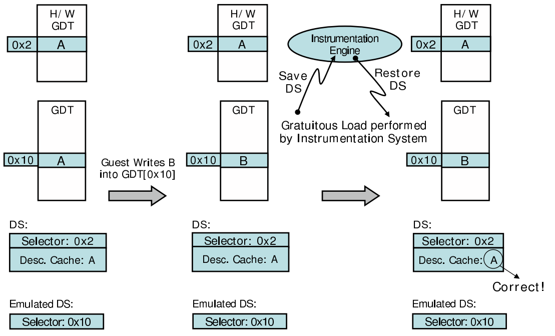


Figure 8. An illustration of how the *Irreversible Segmentation* problem is addressed.

before delivering the page fault to the guest page-fault handler. Implementing rollback in the code cache is highly complicated [6], mainly because of the need to maintain bookkeeping of how much and what kind of progress we have made since the last instruction boundary so that we can rollback that progress; nevertheless, it is relatively straightforward to implement rollback inside the instrumentation engine. For this reason, PinOS never generates any multiple-instruction translation inside the code cache; instead, all such multiple-instruction sequences are always emulated inside the instrumentation engine. And, we have implemented a stack-style logging mechanism inside the instrumentation engine to track all changes to the guest's register and memory state during emulation. Using this log, we can rollback to the last guest instruction boundary in case an exception occurs during emulation.

Faithful exception emulation means the following. When an instruction is executed, the OS may expect the hardware to raise a certain exception when some condition is met. Therefore, if the instruction is emulated by the instrumentation engine (instead of directly running on hardware), we should check all conditions and raise exceptions corresponding to the hardware semantics. One example is the emulation of the `mov into ss` instruction: we need to raise a general-protection fault if the segment sector `ss` has a value of 0.

5.3 Virtualization of Segmentation Support

Due to the memory requirements of PinOS, we need to steal not only virtual and physical pages but also segment descriptor table entries. Also, we need the ability to switch segment registers between guest and PinOS descriptor values as and when needed.

Segmentation state in the Intel x86 architecture consists of primarily a set of segment registers, `CS`, `DS`, `ES`, `FS`, `GS` and `SS`, and two tables, the global (`GDT`) and local (`LDT`) descriptor tables. The segment registers themselves consist of a *visible* segment selector part and a *hidden* segment descriptor cache. The hidden part is cached with a descriptor value from the entry in GDT/LDT that the selector points to. The descriptor cache may be out-of-sync with respect to the current descriptor value in the table, and software often depends on this semantic guarantee. In addition, while a load of a segment register loads both the visible and hidden parts, a store saves only the visible part. This asymmetry in saving and restoring results in the *irreversible segmentation problem* [12], where a segment register's value once replaced cannot be reversed.

A relevant instance of this problem is illustrated in Figure 7. Initially, the `DS` register has selector value `0x10`. The corresponding `GDT` entry has a descriptor value `A` and the `DS` descriptor cache also has been loaded with the value `A` upon the register being loaded. Later, the guest writes `B` into this `GDT` entry. However, the `DS` descriptor cache continues to have the value `A`. But, when the instrumentation system saves and restores `DS` gratuitously, due to context switches from the code cache and back for example, the descriptor cache ends up being loaded with the new value `B`, whereas the guest expects it to still have the value `A`, because `DS` has not yet been explicitly loaded as far as the guest is concerned!

To address the irreversible segmentation problem and to enable stealing of descriptor table entries, we employ a segmentation virtualization scheme where only the descriptor caches of the guest system are shadowed instead of the entire descriptor tables being shadowed, a technique similar to the one proposed in VDebug [6]. PinOS maintains its own `GDT`, which is accessible by the underlying hardware. In this `GDT`, PinOS reserves some entries corresponding to the `x86` segment descriptor caches. The rest of the entries in the table are available for PinOS's own use. As and when the guest explicitly loads segment registers, we copy guest segment descriptors into the corresponding cache entries in PinOS's `GDT`, and issue a segment register load instruction with the selector value modified to point to that cache entry. To implement this scheme, we use dynamic translation of the guest instruction stream to identify and emulate segment-selector-sensitive instructions. This scheme ensures preservation of guest-expected hardware semantics.

Figure 8 shows how this solution addresses the irreversible segmentation problem. The same instance as earlier is shown under the virtualization scheme we employ. This time, when the gratuitous load of the `DS` register is performed by the instrumentation system, `DS` is still loaded with the value `A` because that is what resided in the "hardware-accessible" (i.e., PinOS) `GDT` even though the guest `GDT` entry contained the new value `B`. This is because even though the guest `GDT` entries can keep changing upon guest writes, the PinOS `GDT` entries change only upon explicit segment register loads performed by the guest.

A major advantage of this scheme is that there is no need for tracking guest descriptor table writes [12], as we bring in descriptor values into the caches just-in-time, i.e., upon explicit guest loads of segment registers. The downside lies in the need to use dynamic translation and emulate all segment-selector-sensitive instructions, which increases the system's complexity significantly. However, we have observed that such instructions are usually dynamically rare, and that the scheme does not have a significant performance impact.

5.4 Code Cache Coherence

One of the most effective optimizations that make dynamic translation a viable technique is the reuse of translated code through the maintenance of a code cache. However, the reuse brings an important challenge: keeping the code cache coherent with respect to the original code. In a whole-system scenario, this challenge is two-fold: keeping the code cache coherent in the face of (self)-modifying code and in the face of page mapping changes.

The *content* of a guest code-page may change after PinOS has cached code from that page on to the code cache. In such a situation, reusing the existing translation cache is *incorrect*. We address this problem using the standard write-monitoring solution. Whenever PinOS brings code from a page into the code cache, we mark that page non-writable. Writing to such a monitored page causes a page-fault. PinOS intercepts this page-fault exception, invalidates all translated code belonging to that page in the code cache, and then re-executes the write.

The write-monitoring technique we described above is a widely-employed, standard technique in the application-level space. The application of this technique to an entire system instead of a single application process presents two important issues. The first is maintaining transparency from the guest operating system. We perform the marking of a page as non-writable in the *shadow* page table instead of in the *guest* page table, so that the write-monitoring is performed completely transparently from the guest. The second is the issue of ensuring correctness in the face of alternate mappings. It is not enough to mark the individual page-mapping corresponding to the virtual address of a code fragment that is being brought in to the code cache as non-writable. There may exist other page-mappings that map on to the same physical page because of page-sharing, for instance. No matter which mapping is used to perform the write, if the code-page content is ultimately changed, the code cache must be updated accordingly. Therefore, to ensure correctness of the code cache coherence solution, we must monitor not only that particular page-mapping, but also all present and *future* page-mappings that map on to that *physical page*. To implement such monitoring, we maintain a *reverse page-mapping table* that maps a physical page to all virtual pages that have been observed to map on to it. Thus, translated code present in the code cache is guaranteed to be coherent with respect to the *current* page content.

The code cache must also be kept coherent in the face of page-mapping changes, i.e., if the virtual page is mapped on to a different physical page, the code cache must be updated. As described in section 5.1, we index the code cache using the $\langle VA, PA\rangle$ pair, and introduce a check block to ensure the currency of the $\langle VA, PA\rangle$ mapping at every trace entry to solve the trace linking correctness problem. It turns out that this solution automatically ensures code cache coherence even in the face of page-mapping changes, because if the virtual page is mapped on to a different physical page, control would be passed to the fall-back strategy in the instrumentation engine upon execution of the check block in the incorrect translation.

In addition to keeping the code cache coherent, we must also keep the cached page-mappings in the SoftTLB coherent in the face of page-mapping changes by the guest. Xen already marks guest page-table pages read-only and thus tracks all writes to them. Therefore, to achieve SoftTLB coherence, we rely on Xen to inform PinOS with updates about page-mapping changes so that the SoftTLB that PinOS maintains can be updated accordingly.

6. Evaluation

To demonstrate the usefulness of PinOS, we have ported two tools from Pin to PinOS. One is a code profiler called *Insmix* while the other is a cache simulator called *CMP$im*. In this section, we present some analysis results obtained from these tools. We also present an evaluation of the performance overhead of PinOS.

Setup	**Hardware**	**Xen Dom0 (Host OS)**	**Xen DomU (Guest OS)**
A	2.40GHz, Core 2 Duo 6600, 4096KB L2 Cache 2GB Memory	FC5 with XenoLinux Kernel 2.6.16	FC4 with Linux Kernel 2.6.11
B	2.80GHz, Pentium D, 2048KB L2 Cache 4GB Memory	FC4 with XenoLinux Kernel 2.6.16	FC4 with Linux Kernel 2.6.11

Table 1. Experimental Setup

Benchmark Short Name	**Benchmark Full Name**	**Run With Parameters**	**Run On**
FC4-Boot	FC4-Linux Booting	Default	A
Apache Bench	ab Benchmark on Apache httpd 2.2.3	n=50000	A
AlterTable	MySQL test-alter-table	Default	B
ATIS	MySQL test-ATIS	Default	B
BigTables	MySQL test-big-tables	Default	B
Connect	MySQL test-connect	10000-loop	B
Create	MySQL test-create	1000-loop	B
Select	MySQL test-select	1000-loop	B
Wisconsin	MySQL test-wisconsin	Default	B

Table 2. Benchmark Details

6.1 Experimental Setup

All the results we present in this section were obtained on two machines [2] equipped with Intel VT support and installed with the Xen VMM. Table 1 shows the configurations of the two machines, called setup *A* and *B*.

We performed experiments on several benchmarks, including the boot procedure of FC4-Linux, Apache ab benchmark, and MySQL sql-test benchmark suite. They were picked for their relatively large portion of execution time spent in the kernel. Table 2 shows the details of these benchmarks, along with the short names we use in the rest of this section.

6.2 Insmix

Insmix is a code profiler that collects two pieces of information about program execution. The first is the dynamic frequency of each instruction opcode executed in the program. The second type is the dynamic frequency of all basic blocks executed. Insmix has been used to compare the quality of code generation of different compilers. It has also been used to identify the hot basic blocks in programs for optimization. In the PinOS version, Insmix collects this information in both user and kernel space, and reports them separately in the output file.

Figure 9 shows the user-level vs. kernel-level distribution of dynamic instruction counts obtained by running the Insmix tool on each of our benchmarks. While the kernel-level instructions account for as high as 65% of overall instructions executed in the case of MySQL test-create benchmark, and as high as 40-45% in the case of FC4-Linux boot and Apache ab benchmark, they account for a smaller, yet significant, portion of overall instructions in the rest of the cases. This analysis suggests that while instrument-

[2] The results were obtained on two machines due to lack of time to run all the experiments on the same machine.

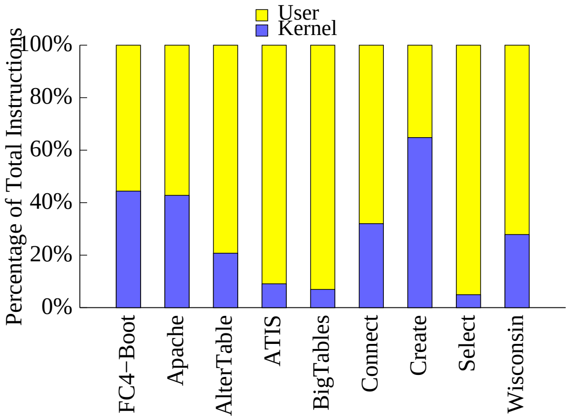


Figure 9. User/kernel distribution of instructions obtained by running the Insmix tool on various benchmarks using PinOS.

ing a subject execution, instrumenting kernel-level instructions as well can be important, and in some cases may be even inevitable.

Table 3 shows the analysis results obtained by running the Insmix tool on MySQL test-alter-table benchmark. The results shown include the top five most frequently executed basic-blocks in the Linux kernel while running MySQL test-alter-table, along with their mapped symbol identifiers, the dynamic execution count observed, the number of instructions contained in that basic-block, and finally, the contribution of that basic-block to overall dynamic instruction count. Among these basic-blocks, the second one serves as an interesting observation: more than a third of a percent of overall instructions executed belong to a basic-block in the function *ext3_do_update_inode*, which is expected given that the MySQL test-alter-table benchmark would have to update disk files frequently.

Table 4 shows the opcode frequency of all x86 privileged instructions observed while running the Insmix tool on MySQL test-alter-table and on FC4-Boot. The Insmix tool was able to gather the dynamic instruction counts, grouped by opcode for the entire instruction set. Here we show only counts for the privileged subset of the x86 instruction set. Note that the tool was not written to be able to distinguish between different kinds of *MOV* instructions, and hence could not gather the counts in the case of *MOV* from/into control registers (CR) and debug registers (DR). The results reveal that some *system state setup* instructions like *LGDT* (Load Global Descriptor Table register) and *LLDT* (Load Local Descriptor Table register) are encountered a few times during the boot procedure, but never encountered later. On the other hand, the *CLI* (Clear Interrupt bit), *STI* (Set Interrupt bit) and *IRETD* (Interrupt Return, usually used for returning to user-level from a system call) instructions are encountered hundreds of thousands of times during execution of both benchmarks.

6.3 CMP$im

CMP$im is a detailed cache simulator that models a multi-level cache hierarchy, including both instruction and data caches. It can model private and shared caches on a multi-core system, although in its current PinOS version only single-core systems are modeled. CMP$im with PinOS simulates cache accesses from both user and kernel code. To distinguish accesses to distinct data that have same virtual addresses, we use physical addresses as cache tags.

Bbl Addr	BBl Symbol	Count	#Insts	Inst. % Contrib.
c0111a40	delay_pit+0x1a	93531291	2	1.17%
c8aac20b	**ext3_do_update_inode+0x82**	**10177398**	**6**	**0.38%**
c011d58f	__might_sleep+0x2a	5170776	5	0.16%
c011d57f	__might_sleep+0x1a	5170776	4	0.13%
c011d565	__might_sleep	5170776	10	0.32%

Table 3. Top five hottest basic-blocks in the Linux kernel obtained by running the Insmix tool on MySQL test-alter-table benchmark using PinOS. The column "Inst. % Contrib." is the percentage of instructions contributed by that basic block (i.e. "Count" × "#Insts") with respect to the total instruction count.

Priv. Instr	Alter Table	FC4 Boot	Priv. Instr	Alter Table	FC4 Boot
CLI	8912950	2806991	LGDT	0	2
STI	2217286	845921	LLDT	0	2
IRETD	599646	574204	LIDT	0	2
OUT	551209	57181	LTR	0	1
OUTSW	990104	9824	INVD	0	0
IN	311762	31994	WBINVD	0	0
INSW	240	48403	RDMSR	0	15
HLT	207	4458	WRMSR	0	0
INVLPG	619	54923	RDPMC	0	0
CLTS	1350	80	LMSW	0	0
RDTSC	7043	1777	MOV CR/DR	NA	NA

Table 4. Opcode frequency of all x86 privileged instructions obtained by running the Insmix tool on MySQL test-alter-table and FC4-Boot using PinOS.

We ran CMP$IM on MySQL (using "test-create" with a loop count of 1000) with a simulated cache configuration of separate 32KB D-cache and 32KB I-cache, a 512KB unified L2-cache, and a 4MB unified L3 cache. Figure 10 shows various metrics on the D-cache behaviors varying over time. We show the results with whole-system simulation in Figure 10(a) and the results with user-mode only simulation in Figure 10(b). First, we note that the whole-system simulation has over 3 billion instructions while the user-mode only simulation has only 1.1 billion instructions. Second, in terms of both misses/1000 instructions and miss rate, the whole-system simulation has different phase behaviors than the user-mode only simulation, although their running averages are similar. This result indicates that memory accesses from the kernel do have measurable impact on cache behavior, and that whole-system instrumentation helps obtain a more accurate picture while performing architectural simulation.

6.4 Performance Evaluation

Figure 11 shows the performance slowdown of PinOS relative to native execution. The first bar in each case corresponds to the PinOS run with no instrumentation, which we call *NullTool*, and the second bar corresponds to the PinOS run with instrumentation being performed by the *Insmix* tool. As can be seen from the figure, while the NullTool slowdown is as little as 12x and 20x in the cases of AlterTable and Create respectively, it is as high as 120x in the case of Wisconsin, and ranges around 50-60x in most of the other cases. The Insmix slowdown is approximately 20-30% more than the NullTool slowdown, because of the extra instrumentation overhead incurred during execution.

Figure 11 also shows the breakup of slowdown in terms of contributions by the two alternately executing components of PinOS: code cache and instrumentation engine. The results indicate that the instrumentation engine accounts for over 60-70% of the perfor-

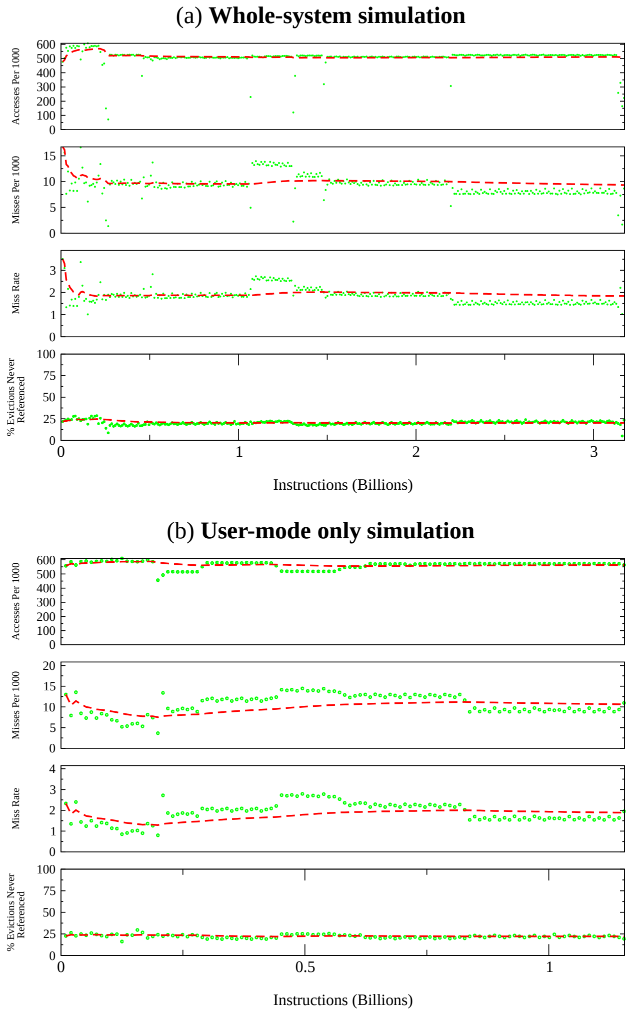


Figure 10. D-cache behaviors of MySQL collected via CMP$im. The x-axis is number of instructions simulated in billions. The y-axis has four metrics: (1) number of memory accesses per 1000 instructions, (2) number of misses per 1000 instructions, (3) miss rate, and (4) percentage of evicted cache lines that are never referenced again in the future. For each metric, the dots represent phase behaviors (one dot per million instructions) while the dashed line is the running average.

mance overhead in all the cases. The instrumentation engine may be entered for several reasons, the two main ones being code compilation and instruction emulation. As future work, we hope to obtain more detailed results to help understand the individual contributions by these tasks towards the overall performance overhead.

Finally, the instrumentation overhead of PinOS reported here is significantly higher than the one reported for user-level Pin [19]. There are three main reasons. First, the benchmarks we chose for PinOS evaluation have much less code reuse than the SPEC2000 benchmarks used in the evaluation of user-level Pin. As a result, the dynamic compilation overhead contributes more to the overall performance in our results. Second, there are two additional run-time tests[3] in each code-cache trace generated by PinOS than by user-level Pin. Third, PinOS emulates many more instructions via the instrumentation engine than user-level Pin. Bringing the overhead of PinOS closer to the level of Pin is on our future work agenda. Nevertheless, we don't view the current level of high overheads as a significant concern, considering that our focus has not been on tuning performance so far. Our focus has rather been on building a new kind of instrumentation framework in the first place.

7. Related Work

In this section, we discuss how our system differs from (or builds upon) various systems that either perform dynamic instrumentation or use software dynamic translation (SDT) or both, as well as systems that simulate, emulate or virtualize machines.

Dynamic optimization systems such as Dynamo [1] and Mojo [9] employ software dynamic translation to perform run-time optimization to improve the performance of native binaries.

SDT infrastructures such as DynamoRio [4], Strata [25],[18] and Walkabout [10] offer a general framework to build new tools that use dynamic binary translation to perform a variety of tasks including optimization, reference monitors and architectural simulation. For example, program shepherding [17] employs the DynamoRio infrastructure to build a tool that enforces security policies on execution of untrusted binaries by monitoring control flow transfers. Another instance is Strata being employed to build a high-performance architectural simulator through a framework of trigger-action mechanisms.

Shade [11] uses SDT to implement high-performance instruction set simulators. However, both Shade and the architectural simulators built with Strata are user-level only simulators.

SDT is employed to build dynamic binary analysis and instrumentation frameworks in the case of Valgrind [23], Pin [19], Nirvana [3] and HDTrans [26]. Valgrind uses sophisticated SDT to perform heavyweight dynamic binary analysis which can be used for performance measurements, profiling, memory analysis and debugging. Pin is a dynamic binary instrumentation tool that provides a high-level API for run-time instrumentation of programs that can be used towards computer architecture research and education, among other things. Nirvana is a run-time framework that can collect a complete user-mode trace of a program's execution that can be re-simulated deterministically. HDTrans is a lightweight dynamic instrumentation system for the IA-32 architecture. Both Valgrind and Pin offer rich APIs towards making their frameworks easily programmable. In all these cases, the instrumentation is performed only on user-level code. That is, all system calls are treated as black-box instructions as far as instrumentation is concerned.

On the other hand, while there do exist a slew of instrumentation frameworks that can instrument *operating systems*, they are mostly *probe-based*, i.e., given a program location that is to be instrumented, they insert *probes* at those locations in-place in the subject binary. The probe can be either a breakpoint instruction or a "software breakpoint" instruction, such as a jump or call to an alternate location. In either case, the destination of the probe instruction would be an instruction sequence that performs instrumentation, executes the originally replaced instruction, and then returns control back to the caller. In most cases, the probes can be inserted, enabled and disabled dynamically. Examples include KernInst [28], DynInst [5], LTT (Linux Trace Toolkit) [31], DProbes [21], KProbes [8], DTrace [7] (on Solaris), and SystemTap [24] (which is the equivalent of DTrace in the Linux world). These tools differ in several ways with respect to our SDT-based whole-system dynamic instrumentation approach. First, these frameworks are usually meant for instrumentation at function boundaries and such, but are generally not suitable for very fine-grain instrumentation. Second, because of their probe-placing mechanism, they generally have limitations about where instrumentation can be added

[3] One test is for validating the page mapping (see Section 5.1 and the other is for checking if there is any pending interrupt (see Section 5.2)

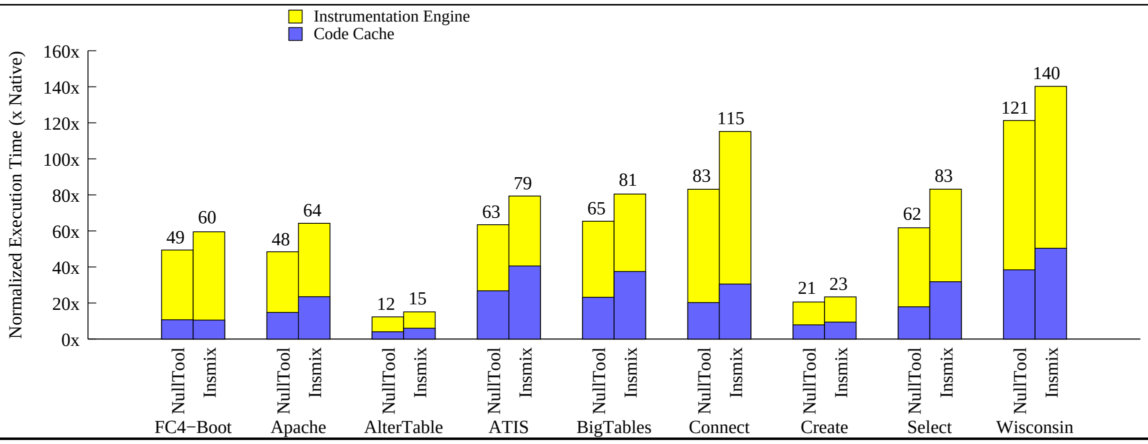


Figure 11. Performance overhead of PinOS on various benchmarks (AlterTable, ATIS, BigTables, Connect, Create, Select and Wisconsin are individual test cases of the MySQL sql-test benchmark suite). The first bar in each case corresponds to a NullTool run and the second bar corresponds to an Insmix run. Each bar further shows the breakup of contributions by code cache execution and by instrumentation engine execution.

on variable instruction-sized architectures such as x86. Third, perhaps most importantly, these frameworks are not suitable to be used for *pervasive* instrumentation such as memory checkers and instruction tracers as this would amount to placing probes at every (memory) instruction that is potentially executed. Finally, these frameworks are OS-specific (e.g., DTrace on Solaris vs. SystemTap on Linux) as their instrumentation engines are built in to the OS being instrumented.

Bochs [16] is an open-source IA-32 PC emulator. Another full-machine emulator is QEmu [2], which uses dynamic translation to implement a fast machine emulator capable of emulating several CPUs on several hosts. While both these systems can emulate the entire machine, they do not facilitate general instrumentation, nor do they provide a programmable framework for performing instrumentation.

Simics [20] is a full machine simulation platform. It provides an API for extensibility in terms of adding new plug-in device models, etc. But, it does not seem straightforward to adapt this API to be used as a programmable instrumentation framework.

Perhaps the closest to our work is Embra [30], which employs whole-system SDT to perform full-machine simulation of a MIPS machine. As they had to overcome the challenges of performing whole-system dynamic translation, they had to solve problems that are similar to the ones we addressed. For instance, they too had to address the complications posed by trace linking in the presence of tricky page mappings. Apart from the technical difference that it did not have to deal with the presence of variable instruction sizes of the x86 architecture, it does not seem to provide a programmable framework for a user to easily build a tool to perform dynamic whole-system instrumentation.

VMware [12] is a full-system virtual machine emulator that uses a combination of native execution for non-privileged code and dynamic translation to emulate privileged-mode behavior. Xen [14] avoids the need to perform dynamic translation by employing *para-virtualization* (which was originally proposed by Denali [29]), i.e., performing modifications to the guest system to make it *Xen-aware* or *Xen-friendly*. Recently, with the help of hardware-assisted virtualization technology such as Intel VT [22], Xen 3.0 [13] can provide the virtualization service to *unmodified* guests. Both VMware and Xen share Bochs and QEmu's goal of simply providing an illusion of a full-machine to their users, and do not facilitate dynamic instrumentation.

8. Conclusion and Future Work

We have presented PinOS, an extension of the Pin dynamic instrumentation framework, for whole-system dynamic instrumentation. By running inside the Xen virtual machine environment using the Intel VT hardware support, PinOS can instrument unmodified operating systems. And by using software dynamic translation techniques, PinOS is able to instrument at a finer granularity and more pervasively than other whole-system instrumentation frameworks. We have demonstrated how PinOS could be used for system-wide program analysis and architectural studies with a code profiler and cache simulator built on top of PinOS.

Future work includes porting PinOS to other platforms, such as 64-bit x86, multiprocessors, and Windows. We are also working on improving our time virtualization techniques and instrumentation performance.

Acknowledgments

We thank the reviewers of VEE 2007 for their helpful critique and comments. We thank the entire Pin team at Intel for giving us a robust Pin to start with. Thanks to Mark Charney for helping us better understand Xed (the x86 encoder/decoder used by Pin and PinOS) and for fixing Xed issues very promptly. Thanks to Greg Lueck for many helpful discussions, especially about signals, and for fixing related bugs in Pin. Thanks to Robert Cohn and Vijay Janapareddi for helpful discussions. We thank Prof. Jonathan Shapiro and Swaroop Sridhar (The Johns Hopkins University) for many many helpful discussions and collaboration on initial ideas about x86 segmentation virtualization, among other things.

References

[1] Bala, V., Duesterwald, E., and Banerjia, S. Dynamo: A transparent dynamic optimization system. In *Proc. 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 2000), pp. 1–12.

[2] Bellard, F. QEMU, a Fast and Portable Dynamic Translator. In *Proc. 2005 USENIX Annual Technical Conference – FREENIX Track* (2005), pp. 41–46.

[3] Bhansali, S., Chen, W.-K., d. Jong, S., Edwards, A., Murray, R., Drinic, M., Mihocka, D., and Chau, J. Framework for Instruction-level Tracing and Analysis of Program Execution. In *Proceedins of the Second International Conference on Virtual Execution Environments* (June 2006), pp. 154–163.

[4] Bruening, D., Garnett, T., and Amarasinghe, S. An Infrastructure for Adaptive Dynamic Optimizations. In *Proc. International Symposium on Code Generation and Optimization* (2003), pp. 265–275.

[5] Buck, B., and Hollingsworth, J. K. An API for Runtime Code Patching. In *International Journal of High Performance Computing Applications* (2000), vol. 14, pp. 317–329.

[6] Bungale, P. P., Sridhar, S., and Shapiro, J. S. Supervisor-Mode Virtualization for x86 in VDebug. Tech. Rep. SRL2004-01, Johns Hopkins University Systems Research Laboratory, May 2004.

[7] Cantrill, B. M., Shapiro, M. W., and Leventhal, A. H. Dynamic Instrumentation of Production Systems. In *Proc. 2004 USENIX Annual Technical Conference.*

[8] Center, L. T. KProbes. `http://sourceware.org/systemtap/kprobes/`.

[9] Chen, W. K., Lerner, S., Chaiken, R., and Gillies, D. M. Mojo: A Dynamic Optimization System. In *Proc. ACM Workshop on Feedback-directed and Dynamic Optimization (FDDO-3)* (Dec 2000).

[10] Cifuentes, C., Lewis, B., and Ung, D. Walkabout–A Retargetable Dynamic Binary Translation Framework. In *Proc. 2002 Workshop on Binary Translation* (September 2002).

[11] Cmelik, B., and Keppel, D. Shade: A fast instruction- set simulator for execution profiling. In *ACM SIGMETRICS Conf. on the Measurement and Modeling of Computer Systems* (1994), pp. 128–137.

[12] Devine, S., Bugnion, E., and Rosenblum, M. Virtualization System Including a Virtual Machine Monitor for a Computer with a Segmented Architecture. In *United States Patent 6,397,242* (May 2002).

[13] Dong, Y., Li, S., Mallick, A., Nakajima, J., Tian, K., Xu, X., Yang, F., and Yu, W. Extending Xen with Intel Virtualization Technology. In *Intel Technology Journal* (August 2006), vol. 10.

[14] Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Pratt, I., Warfield, A., Barham, P., and Neugebauer, R. Xen and the Art of Virtualization. In *Proc. 2003 ACM Symposium on Operating Systems Principles* (Oct. 2003), pp. 164–177.

[15] Intel. *Pin User Manual*. http://rogue.colorado.edu/Pin.

[16] Kevin Lawton. Bochs IA-32 Emulator Project . `http://bochs.sourceforge.net`.

[17] Kiriansky, V., Bruening, D., and Amarasinghe, S. Secure Execution via Program Shepherding. In *11th USENIX Security Symposium* (August 2002).

[18] Kumar, N., Childers, B. R., and Soffa, M. L. Low overhead program monitoring and profiling. In *PASTE '05: The 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (New York, NY, USA, 2005), ACM Press, pp. 28–34.

[19] Luk, C. K., Cohn, R. S., Muth, R., Patil, H., Klauser, A., P. G. Lowney, Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: Building Customized Program Analysis Tools With Dynamic Instrumentation. In *Programming Languages Design and Implementation 2005* (June 2005), pp. 190–200.

[20] Magnusson, P. S., Christensson, M., Eskilson, J., Forsgren, D., Hllberg, G., Hgberg, J., Larsson, F., Moestedt, A., and Werner, B. Simics: A Full System Simulation Platform. In *IEEE Computer* (Feb 2002), pp. 50–58.

[21] Moore, R. A Universal Dynamic Trace for Linux and other Operating Systems. In *Proc. 2001 USENIX Annual Technical Conference – Freenix Track.*

[22] Neiger, G., Santoni, A., Leung, F., Rodgers, D., and Uhlig, R. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. In *Intel Technology Journal* (August 2006), vol. 10.

[23] Nethercote, N. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, November 2004.

[24] Prasad, V., Cohen, W., Eigler, F. C., Hunt, M., Keniston, J., and Chen, B. Locating System Problems Using Dynamic Instrumentation. In *Proc. 2005 Ottawa Linux Symposium (OLS)* (Jul 2005).

[25] Scott, K., Kumar, N., Velusamy, S., Childers, B., Davidson, J., and Soffa, M. Retargetable and Reconfigurable Software Dynamic Translation. In *ACM SIGMICRO Int'l. Conf. on Code Generation and Optimization* (March 2003).

[26] Sridhar, S., Shapiro, J. S., Bungale, P. P., and Northup, E. HDTrans: An Open Source, Low-Level Dynamic Instrumentation System. In *Proc. 2006 International Conference on Virtual Execution Environments (VEE)* (June 2006).

[27] Srivastava, A., and Eustace, A. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 94 Conference on Programming Language Design and Implementation* (1994), pp. 196–205.

[28] Tamches, A., and Miller, B. P. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. In *Proc. 1999 Symposium on Operating Systems Design and Implementation (OSDI).*

[29] Whitaker, A., Shaw, M., and Gribble, S. D. Scale and Performance in the Denali Isolation Kernel. In *Proc. 2002 ACM Symposium on Operating System Design and Implementation (OSDI)* (Dec. 2002).

[30] Witchel, E., and Rosenblum, M. Embra: Fast and flexible machine simulation. In *Measurement and Modeling of Computer Systems* (1996), pp. 68–79.

[31] Yaghmour, K., and Dagenais, M. R. Measuring and Characterizing System Behavior Using Kernel-Level Event Logging. In *Proc. 2000 USENIX Annual Technical Conference.*