Fast Dynamic Binary Translation for the Kernel

Piyus Kedia and Sorav Bansal
Indian Institute of Technology Delhi

Abstract

Dynamic binary translation (DBT) is a powerful technique with several important applications. System-level binary translators have been used for implementing a Virtual Machine Monitor [2] and for instrumentation in the OS kernel [10]. In current designs, the performance overhead of binary translation on kernel-intensive workloads is high. e.g., over 10x slowdowns were reported on the syscall nanobenchmark in [2], 2-5x slowdowns were reported on `lmbench` microbenchmarks in [10]. These overheads are primarily due to the extra work required to correctly handle kernel mechanisms like interrupts, exceptions, and physical CPU concurrency.

We present a kernel-level binary translation mechanism which exhibits near-native performance even on applications with large kernel activity. Our translator relaxes transparency requirements and aggressively takes advantage of kernel invariants to eliminate sources of slowdown. We have implemented our translator as a loadable module in unmodified Linux, and present performance and scalability experiments on multiprocessor hardware. Although our implementation is Linux specific, our mechanisms are quite general; we only take advantage of typical kernel design patterns, not Linux-specific features. For example, our translator performs 3x faster than previous kernel-level DBT implementations while running the Apache web server.

1 Introduction

Dynamic binary translation (DBT) is a popular technique, with applications in virtualization [2], testing/verification [14], debugging [20], profiling [19], sandboxing [12], dynamic optimizations [4], and more. DBT can be implemented both at user-level [7] and at system-level [2, 10]. Current system-level binary translators exhibit large performance overheads on kernel-intensive workloads. For example, VMware's binary translator in a Virtual Machine Monitor (VMM) shows 10x slowdowns for the `syscall` nanobenchmark [2]; corresponding overheads are also observed in macrobenchmarks. VMware's DBT performance also includes the overheads of other virtualization mechanisms, like memory virtualization through shadow page tables, etc. Another kernel-level binary translator, DRK [10], reports 2-5x slowdowns on kernel intensive workloads. Applications requiring high kernel activity (like high performance fileservers, databases, webservers, software routers, etc.) exhibit prohibitive DBT slowdowns, and are thus seldom used with DBT frameworks.

Ideally, a translated system must run at near-native speed. Low-overhead user-level DBT is well understood [6]; kernel-level translation involves correctly handling interrupts, exceptions, CPU concurrency, device/hardware interfaces, and is thus more complex and expensive. We present a kernel-level dynamic binary translator with near-native performance. Like DRK [10], our translator works for the entire kernel including arbitrary devices and their drivers. This is in contrast with virtual-machine based approaches (e.g., VMware [2], PinOS [9], BitBlaze [17]), where translation is only performed for code that runs in a virtualized guest. As also discussed by DRK authors [10], making dynamic binary translation work for arbitrary devices and drivers is important because drivers constitute a large fraction of kernel code, and most of this remains unexercised in a virtual machine. Moreover, most interesting program behaviour (e.g., bugs, security issues, etc.) occurs in drivers. Further, many workloads are incapable of running in virtual environments due to device constraints.

We evaluate the performance of our DBT framework on a number of workloads, and show significant improvements over previous work. We also evaluate the functionality of our DBT framework by implementing a few important applications on it. In particular, we obtained Linux kernel's byte-level memory sharing profile on multiprocessor hardware using our DBT framework.

SOSP'13, Nov. 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522718

Our translator is implemented as a loadable kernel module and can attach to a running OS kernel, ensuring that all kernel instructions run translated thereafter. It does not translate user-level code. Our translator exhibits performance *improvements* of up to 17% over native on certain workloads. Similar improvements have previously been observed for user-level binary translators [6], and have been attributed to improved caching behaviour, especially at the instruction cache. Our translator can be detached from a running system at will, to revert to native execution.

Like previous work [2, 10], our translator provides full kernel code coverage, preserves original concurrency and execution interleaving, and is "transparent" to the kernel. i.e., kernel code does not behave differently or break if it observes the state of the instrumented system. While VMware promises complete transparency, both DRK and our translator have transparency limitations, i.e., it is in general possible for kernel code to inspect translated state/data structures. We only ensure that this does not result in incorrect behaviour during regular kernel execution[1]. Our design differs from VMware and DRK in the following important ways:

Entry Points: We replace kernel entry points (interrupt and exception handlers) directly with their translated counterparts. This is in contrast with VMware and DRK which replace kernel entry points with calls to the DBT's dispatcher (see Figure 1 for the component diagram of a dynamic binary translator), which in turn jumps to the translated handlers *after* restoring native state on the kernel stack. This extra work by DBT dispatcher causes significant overhead on each kernel entry. In our design, kernel entries execute at near-native speed. In doing so, we allow the kernel handlers to potentially observe the non-native state generated by the hardware interrupt on stack.

Interrupts and Exceptions: Previous DBT solutions (VMware, DRK) handle exceptions by emulating precise exceptions[2] in software by rolling back execution to the start of the translation of the current native instruction; and handle interrupts by delaying them till the start of the translation of the next native instruction. These mechanisms are complex and expensive, and are primarily needed to ensure that the interrupt handlers observe consistent state. In our design, we allow imprecise exceptions and interrupts. Relaxing precision greatly simplifies design and improves performance. In our experience, operating systems rarely depend on precise exception and interrupt behaviour.

Reentrancy and Concurrency: The translator's code and data structures need to be reentrant to allow interrupts and exceptions to occur at arbitrary program points. Similarly, physical CPU concurrency needs to be handled carefully. DBT requires maintenance of CPU-private data structures, and migration of a thread from one CPU to another should not cause unsafe concurrent access to common state. In our design, the presence of imprecise exceptions and interrupts introduces more reentrancy and concurrency challenges. We present an efficient mechanism to provide correct translated execution.

Our optimizations result in a very different translator design from both VMware and DRK. We are more aggressive about assumptions on usual kernel behaviour. In doing so, we sometimes relax "transparency"; for us, ensuring *correctness* is enough. Essentially, we show that many transparency requirements are unnecessary and can be relaxed for better performance. Because our translator is implemented as a kernel module, it can be used both for standalone kernel translation (as done in our implementation) or for use with VMMs (through the "guest tools" mechanism).

The paper is organized as follows. We present a background discussion on code generation and other kernel-level DBT mechanisms in Section 2. Section 3 presents our faster and simpler design, and Section 4 discusses its subtleties. Section 5 discusses our implementation and results. Section 6 summarizes our techniques and the OS kernel invariants that we rely upon. Section 7 discusses related work, and finally Section 8 concludes.

2 DBT Background

We first introduce the terminology and provide a basic understanding of how a dynamic binary translator works (also see Figure 1). We refer to the terms and concepts described in this section, while discussing our design and optimizations in the rest of the paper. We call the kernel being translated, the *guest* kernel. Starting at the first instruction, a straight-line native code sequence (*code block*) of the guest is translated by the *dispatcher*. A code block (also called a *trace* in previous work) is a straight-line sequence of instructions which terminates at an unconditional control transfer (branch, call, or return). The instructions in a block are translated using a *translation rulebook*. For quick future access, the translations are stored in a *code cache*. The dispatcher ensures that it regains control when the block

[1]Actually, VMware's binary translator also does not guarantee full transparency. For example, they do not translate user-level code for performance. Most "well-behaved" operating systems work well in this model, but an adversarial guest can expose their transparency limitations.

[2]A precise exception means that before execution of an exception handler, all instructions up to the executing (emulated) instruction have been executed, and the excepting instruction and everything afterwards have not been executed. Previous DBT implementations have preserved precise exception behaviour for architectures that support precise exceptions (e.g., x86).

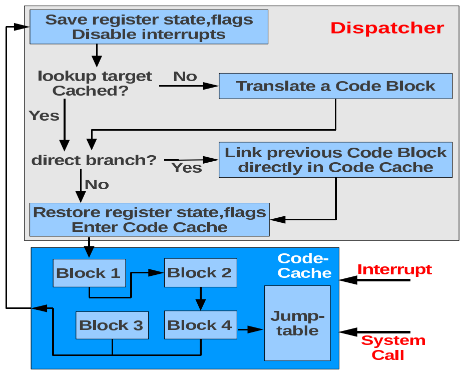

Figure 1: Control Flow of a Dynamic Binary Translator.

```
        jmp .edge0
.edge0: save_registers_and flags
        clear interrupts
        set nextpc
        jump to dispatcher
```

Figure 2: The translated (pseudo) code generated for a direct unconditional branch. After the first execution of this code, the first "`jmp .edge0`" instruction is replaced with "`jmp tx-nextpc`" to implement direct branch chaining.

exits by replacing the terminating control flow instruction by a branch back to the dispatcher after appropriately setting the next native PC (called `nextpc`). The dispatcher looks up the code cache to search if a translation for `nextpc` already exists. If so, it jumps to this translation. If not, the native code block beginning at `nextpc` is translated and the translation is stored in the code cache before jumping to it. We call the translated code corresponding to `nextpc`, `tx-nextpc`.

To improve performance, *direct branch chaining* is used, i.e., before the dispatcher jumps to a translation in the code cache, it checks if the previous executed block performed a direct branch to this address. If so, the corresponding branch instruction in the previous executed block is replaced with a direct jump to the translation of the current program counter. This allows the translated code to directly jump from one block to another within the code cache (without exits to the dispatcher), thus resulting in near-native performance.

Figure 2 shows the translation code for a direct unconditional branch, to illustrate the direct branch chaining mechanism. At the first execution of this translated code, the operand of the first `jmp` instruction is the address of the following instruction (`.edge0`). The code at `.edge0` sets `nextpc` before jumping to the dispatcher. After the first execution, the dispatcher replaces the first instruction with "`jmp tx-nextpc`" to implement direct branch chaining.

If a code block ends with an indirect branch, `nextpc`

```
save flags and clear interrupts
save temporary regs %tempreg0 and %tempreg1
mov *MEM, %tempreg0
%tempreg1 := jumptable_hashfn(%tempreg0)
index %per_cpu:jumptable using %tempreg1
if jumptable hit,
    restore flags and jump to tx-nextpc
else,
    jump to dispatcher
```

Figure 3: Translation for the indirect branch instruction "`jmp *MEM`", which looks up the jumptable to convert `nextpc` to `tx-nextpc`. As discussed in Section 4.1, a separate per-CPU jumptable is maintained, and "`%percpu:jumptable`" obtains the address of the jumptable of the currently executing CPU. `jumptable_hashfn()` represents the jumptable's hash function. On Linux, the `%fs` segment stores the value of the `%per_cpu` segment and is used to store CPU-private variables (like the jumptable in this case). Section 4.1 also discusses the need to clear interrupts before a jumptable lookup.

can only be determined at runtime. As an optimization, a fast lookup table is maintained to convert `nextpc` to `tx-nextpc` without having to exit into the dispatcher. The lookup table, called *jumptable*, is implemented as a small hashtable. Additions to the jumptable are done in the dispatcher, and lookups are done using assembly code (emitted in the code cache for every indirect branch). Figure 3 shows the pseudo-code of the translation of an indirect branch.

2.1 Kernel-level DBT Background

Kernel-level DBT requires more mechanisms to correctly handle interrupts, exceptions, reentrancy and concurrency issues. Interposition on kernel execution is ensured by replacing all kernel entry points (interrupt and exception handlers) with custom handlers. In previous work, these entry points have been replaced with a call to the DBT dispatcher. The dispatcher receives as argument, the original PC value at the entry point. Before the dispatcher translates and executes the handler at this PC value, it performs more work as discussed below. (As we discuss in Section 3, we avoid this extra work in our design).

PC value pushed on the interrupt stack by hardware is translated to its native counterpart by the dispatcher on every interrupt/exception. The value pushed on stack by hardware is the PC value at the time of the interrupt/exception. This value could be a code cache address or a dispatcher address. In either case, the value is replaced with the address of the *native instruction* that must run after the handler finishes execution. Consequently, the return-from-interrupt instruction (`iret`) is translated to obtain `nextpc` from stack and exit to the dispatcher.

If a synchronous exception has occurred in the middle of the translation of an instruction, **precise exception behaviour is emulated**. The procedure requires *rolling back* machine state to its state at the start of the (translation of the) current native instruction. The code to implement this rollback must be provided in the translation rulebook, and is executed in the context of the dispatcher. After executing the rollback code and putting the native instruction address on stack, the exception handler is executed.

If an asynchronous interrupt was received, the **delivery of this interrupt is delayed** until the (translation of the) next native instruction boundary. This is done to ensure *precise interrupts*, i.e., the interrupted native instruction must never be seen "partially executed" by the handler. This delayed interrupt delivery is implemented by patching the translation of the next native instruction with a software-interrupt instruction (to recover control at that point). After recovering control, the interrupt stack is setup to return to this next instruction before executing the interrupt handler.

These mechanisms are discussed in detail in the DRK paper [10] and also previously in a VMware patent [8]. The complexity of these mechanisms is evident from the 5-6 long pages of explanation needed to describe them in previous work. These mechanisms are also expensive, as we discuss next:

First, replacing the PC value pushed by hardware on the interrupt stack, to its native counterpart, on each interrupt, results in significant overhead for an interrupt-intensive application. Similarly, the translation code for the `iret` instruction adds overhead on every return from interrupt.

Second, the rollback operation required to ensure precise exceptions is expensive. There is a direct cost of executing the rollback code on each exception. But more significantly, there is an indirect cost of having to structure a translation in a way that it can be rolled back. Typically, this involves making a copy of the old value of any register that is being overwritten. This cost is incurred on the straight-line non-exceptional execution path on every execution of that instruction, and is thus significant.

Third, the delaying of interrupts involves identifying the next native instruction, patching it, incurring an extra software trap, and then patching the interrupt stack. These are expensive operations.

In our work, we show that a guest kernel rarely relies on the PC value being pushed on stack on an interrupt/exception, and is largely indifferent to imprecise exception and interrupt behaviour, and thus these overheads can be avoided for a vast majority of DBT applications.

3 A Faster Design

In our design, we do not ensure precise exceptions and interrupts. We also do not guarantee that the PC value on the interrupt stack is a valid native address. We simply allow the PC value pushed by hardware to get exposed to the interrupt handler. We also allow the interrupt handler to inspect intermediate machine state if an interrupt/exception occurred in the middle of the translation of a single instruction.

The design is simple. We disallow interrupts and exceptions in the dispatcher (see Section 4.1 for details). Thus, interrupts and exceptions only occur while executing in the code cache, or while executing in user mode. We replace a kernel entry point with the translation of the code block at the original entry point. This causes an interrupt or exception to directly jump into the code cache (see Figure 1). Consequently, we use the identity translation for the `iret` instruction (i.e., `iret` in native code is translated to `iret` in translated code) to return back directly to the code cache. The system thus executes at full speed. But we need more mechanisms to maintain correctness.

The first correctness concern is whether an interrupt or exception handler could behave incorrectly if it observes an unexpected PC value on the interrupt stack. Fortunately, in practice, the answer is no, barring a few exceptions. For example, on Linux, only the page fault handler depends on the value of the faulting PC. The Linux page fault handler uses the faulting PC value to check if the fault is due to a permissible operation (like one of the special `copy_from_user()`, `copy_to_user()` functions) or a kernel bug. To implement this check, the kernel compiler generates an "exception table" representing the PCs that are allowed to fault and the faulting PC is searched against this table at runtime. With DBT, because the code cache addresses will not belong to this table, the page fault handler could incorrectly panic.

Similar patterns, where certain exception handlers are sensitive to the excepting PC value, are also found in other kernels. For example, on some architectures (e.g., MIPS), *restartable atomic sequences* (RAS) [5] are implemented to support fast mutual exclusion on uniprocessors. RAS code regions, indicating critical sections, can be registered with the kernel using PC start and end values. If a thread was context-switched out in the middle of the execution of a RAS region (determined by checking the interrupted PC against the RAS registry), the RAS region is "restarted" by the kernel by overwriting the interrupt return address by the start address of the RAS region. With DBT, this mutual-exclusion mechanism could get violated because the code cache addresses will not belong to the RAS registry. Also, ker-

```
void function_that_can_cause_page_fault()
{
  /* by default, pcb_onfault = 0. */
  push pcb_onfault;
  pcb_onfault = custom_page_fault_handler_pc;

  /* code that could page fault. */

  pop pcb_onfault;
}

void kernel_page_fault_handler()
{
  /* handler invoked on every page fault. */
  if (pcb_onfault) {
    intr_stack[RETADDR_INDEX] = pcb_onfault;
  }
}
```

Figure 4: Pseudo-code showing registry of custom page fault handlers by kernel subsystems in BSD kernels. The `pcb_onfault` variable is set to the PC of the custom page fault handler before execution of potentially faulting code. On a page fault, the kernel's page fault handler overwrites the interrupt return address on stack with `pcb_onfault`.

nels implementing RAS can cause execution of native code as they could potentially overwrite the interrupt's return address with a native value. A similar pattern involving overwriting of the interrupt return address by the handler is also present in the BSD kernels, namely FreeBSD, NetBSD, and OpenBSD. The pattern is shown in Figure 4. As explained in the figure, this is done to allow kernel subsystems to install custom page fault handlers for themselves. As another example of a similar pattern, Microsoft Windows NT Structured Exception Handling model supports a `__try`/`__except` construct which registers the exception handler specified by the `__except` keyword with the code in the `__try` block. These constructs are implemented by maintaining per-thread stacks of exception frames; on entry to a `__try`/`__except` block, an exception frame containing the exception handler pointer is pushed to this stack and on function return, this exception frame is popped off the stack. If an exception occurs, the kernel's exception handler (e.g., page fault handler) traverses this exception stack top-to-bottom to find and execute the appropriate `__except` handler [3]. Because on an exception inside the `__try` block, the kernel's exception handler overwrites the excepting PC, our DBT design can incorrectly cause execution of native untranslated code.

Fortunately, such patterns are few, and can be usually handled as special cases. On Linux for example, the kernel allows loadable modules to register custom exception tables at load time, to extend similar functionality to loadable modules. On a page fault, the faulting PC is also checked against the modules' exception tables. For our DBT implementation, we ensure that the code cache addresses corresponding to the functions that already existed in kernel's exception table belong to our module's exception table. This ensures correct behaviour on kernel page faults. Similarly, DBT for kernels implementing RAS can be handled by manipulating the RAS registry to also include the translated RAS regions. The exception directory in Microsoft Windows for non-x86 architectures can be handled similarly. Further, to avoid execution of native code after interrupt return, due to overwriting of return address by a handler (e.g., custom page fault handler installation in BSD kernels), the `iret` instruction can be translated to also check the return address; if the return address does not belong to the code cache, indicating overwriting by the handler, the translator should jump to the dispatcher to perform the appropriate conversion to its corresponding translated code cache address[4].

In general, we believe that for a well-designed kernel, any interrupt or exception handler whose behaviour depends on the value of the interrupted PC value, should ideally also allow a loadable module to influence the handler's behaviour, because the PC values of the module code are only determined at module load time. For example, Linux provides the module exception table for page fault handling. This allows a DBT module to interpose without violating kernel invariants. In cases where such interposition is not possible, our DBT design will fail.

In some kernels, we also found instances where an excepting PC address is compared for equality with a kernel function address in the exception handler. These checks against hardcoded addresses (as opposed to a table of addresses as in Linux), pose a special problem, as it is no longer possible for the DBT module to manipulate these checks. Fortunately, such patterns are rare, and are primarily used for debugging purposes. If such patterns are known to exist, special checks can be inserted at interrupt entry (by appropriately translating the first basic block pointed to by the interrupt descriptor table) to compare the interrupted PC pushed on stack against translations of these hardcoded addresses. If found equal, the PC pushed on stack should be replaced by their corresponding native code address. Sim-

[3]On non-x86 architectures (e.g., ARM, AMD64, IA64), a somewhat different implementation for `__try`/`__except` is used. A static exception directory in the binary executable contains information about the functions and their `__try`/`__except` blocks. On an exception, the call stack is unwound and the exception directory is consulted for each unwound frame to check if a handler has been registered for the excepting PC.

[4]If the code cache is allocated in a contiguous address range, this translation of `iret` to check the return address is cheap (4-8 instructions). This is much faster than converting native addresses to translated addresses on every interrupt return, as done in previous DBT designs.

OS	Unconventional uses of the interrupted/excepting PC value pushed on stack by hardware
Linux	Found one check against a table of addresses (exception table) in page fault handler.
MS Windows	`__try()/__catch()` blocks implemented by maintaining per-thread stacks of exception frames.
FreeBSD	Found three equality checks against hardcoded function addresses. Found two more uses for debugging purposes. Implements RAS. Overwrites return address to implement custom page fault handlers.
OpenBSD	Implements RAS. Overwrites return address to implement custom page fault handlers.
NetBSD	Found two uses for debugging purposes. Implements RAS. Overwrites return address to implement custom page fault handlers.
BarrelFish	Found no such use.
L4	Found two equality checks against harcoded function addresses in page fault handler.

Table 1: Unconventional uses of the interrupt return address (in ways that need special handling in our DBT design) found in the kernels we studied.

ilar checks should be added on interrupt return with appropriate conversion from native address to its translated counterpart, if needed. Notice that these special-case checks are much cheaper than translations from native addresses to code cache addresses and vice-versa on every interrupt entry and return respectively, as done in previous designs.

Table 1 summarizes our survey findings regarding the use of the interrupted PC address on stack in various kernels. In summary, we allow fast execution of the common case (where interrupted PC value is not read or written), and use special-case handling for the few design patterns where the PC value is known to be read/written in unconventional ways.

The second correctness concern has to do with the presence of code cache addresses in the kernel's data structures. For example, if an interrupt occurs while the translated kernel is executing in the code cache, the code cache address would be pushed on the kernel stack. If the executing thread then gets context-switched out, the code cache address would continue to live in the kernel data structures. If the code cache address becomes invalid in future (due to cache replacement, for example), this can cause a failure.

To solve this problem, we ensure that code cache addresses do not become invalid until they have been removed from all kernel data structures. Firstly, we disallow cache replacement; we assume that the space available for code cache is sufficient to store translations of *all* kernel code. This is not an unreasonable assumption; for example, we use a code cache of 10MB which is sufficient for the Linux kernel, whose entire code section (including code of loadable modules) is typically less than 8MB in size. There may be corner cases, where the size of the code cache may exceed the available space (for example, due to repeated loading and unloading of modules); we discuss how to handle such situations in Section 4.4. Secondly, once a code block is created in the code cache, we do not move or modify it (except the one-time patch for direct branch chaining). This ensures that a code cache address, once created, remains valid for the lifetime of the translated execution. Further, translator switchoff needs to be handled carefully — all code cache addresses should be removed from kernel data structures before effecting a switchoff (see Section 4.4).

The third correctness concern is regarding violation of precise exception and interrupt behaviour. Interestingly, none of the kernel exception handlers we encountered, depend on precise exception behaviour. In practice, the kernel exception handlers at most examine the contents (opcode and operands) of the instruction at the faulting PC to make control flow decisions. As long as the translated code does not cause any extra exceptions or does not suppress any exception that would have occurred in native code, the system behaves correctly. Similarly, the kernel never depends on the PC value for interrupt handling and is thus indifferent to violation of precise interrupts. The handlers are also indifferent to the values of other registers (that are not used by the faulting instruction) at interrupt/exception time[5].

4 Design Subtleties

4.1 Reentrancy and Concurrency

Consider a CPU executing inside the dispatcher. An interrupt or exception at this point could result in a fresh call to the dispatcher, to translate the code in the interrupt/exception handler. Similarly, consider a CPU executing in the code cache. The translated code for an instruction could have multiple instructions and could potentially be using temporary memory locations (scratch space) to store intermediate values (e.g., our translation of the indirect branch instruction uses two scratch space locations). An interrupt/exception in the middle of the translated code could result in race conditions on accesses to this scratch space. We call these reentrancy problems.

In previous work, reentrancy problems were simplified because their designs ensured precise exceptions and interrupts. An interrupt or exception was serviced

[5]System calls depend on register values, but they are implemented as software exceptions in user mode, and we are discussing hardware exceptions/interrupts received in kernel mode; interrupts/exceptions received in user mode are irrelevant to our design as DBT does not influence user-mode behaviour.

```
restore guest registers/stack
restore guest flags (interrupts may get enabled)
jmp *%per_cpu:tx-nextpc-loc
```

Figure 5: The code to exit the dispatcher and enter the code cache at `tx-nextpc` (which is the value stored in `tx-nextpc-loc`). As discussed later, `tx-nextpc-loc` is a per-cpu location accessed using the per-cpu segment.

only after the state of the system reached a native instruction boundary; this meant that a handler (or a dispatcher call made by it) never observed intermediate state. Our design is different, and we discuss the resulting challenges and their solutions.

Firstly, we disallow interrupts and exceptions inside dispatcher execution. Exceptions are disallowed by design; none of the dispatcher instructions are expected to generate exceptions, and page faults are absent because all kernel code pages are expected to be mapped. We also never interpret kernel code within the dispatcher. Interrupts are disallowed by clearing the interrupt flag (using `cli` instruction) before entering the dispatcher. To avoid clobber, the kernel's interrupt flag is saved on dispatcher entry and restored on dispatcher exit. (Notice the clearing of interrupt flag in Figures 2 and 3).

At dispatcher exit (code cache entry), the kernel's flags need to be restored inside the dispatcher before branching to the code cache. This presents a catch-22 situation: restoring kernel flags could cause interrupts to be enabled and thus to preserve reentrancy, there should not be any accesses to a dispatcher memory location after that; and yet we need some space to store `tx-nextpc` (the code cache address to jump to). Figure 5 shows the code at dispatcher exit. Notice that at the last indirect branch in this code, `tx-nextpc` cannot be stored in a register (because the registers are supposed to hold the kernel's values at this point), and cannot be stored on stack (because the stack should not be any different from what the kernel expects it to be). `tx-nextpc` is instead stored at a per-CPU location called `tx-nextpc-loc` (a per-CPU location in the kernel is a location that has separate values for each CPU; we discuss the need for `tx-nextpc-loc` to be per-CPU later in our discussion on concurrency). This code at dispatcher exit is non-reentrant because an interrupt after guest flags are restored and before the last indirect branch to `*tx-nextpc-loc` executes, could clobber `tx-nextpc-loc`.

To solve this problem, we save and restore `tx-nextpc-loc` at interrupt entry and exit respectively. Thus, this one dispatcher memory location has special status. The translation of the first block of all interrupt/exception handlers is augmented to save `tx-nextpc-loc` to stack, and the translation of the last code block before returning from an interrupt (identified by the presence of the `iret` instruction) is augmented to restore `tx-nextpc-loc` from stack. Because some of the interrupt state on stack is pushed by hardware (e.g., code segment, program counter, and flags), simply adding another `push` instruction at interrupt entry (to save `tx-nextpc-loc`) will not work, as that will destroy the interrupt frame layout on stack. On Linux, we identified a redundant location in the stack's interrupt frame structure, and used it to save and restore `tx-nextpc-loc` on interrupt entry and return respectively[6]. If a redundant stack location cannot be identified, the interrupt frame pushed by hardware could be "shifted-down" by one word at interrupt entry (by emitting appropriate code on the interrupt entry path), thus creating space above to save `tx-nextpc-loc`. On interrupt return, `tx-nextpc-loc` could be restored from this location, before "shifting-up" the hardware's interrupt frame and executing `iret`. Alternatively, if modifications to guest's source code are possible, an extra field could be added to the interrupt frame structure for this purpose.

Next, we consider reentrancy problems due to interruption in the middle of a translation in the code cache. To address this, we need to ensure that accesses to scratch space (used in translated code) are reentrant. We mandate that any extra scratch space required by a translation rule should be allocated on the kernel's thread stack. The `push` and `pop` instructions are used to create and reclaim space on stack. Because a kernel follows the stack abstraction (i.e., no value above the stack pointer is clobbered on an interrupt), this ensures reentrant scratch space behaviour. Because typical space allocation for kernel stacks (8KB on Linux) is comfortably more than its utilization, there is no danger of stack overflow due to the small extra space used by our translator.

Finally, accesses to the jumptable need to be reentrant. In Figure 3 which shows the translated code of an indirect branch, consider a situation where the thread executing this translation gets interrupted after it has determined that `nextpc` exists in the jumptable and before it reads the value of `tx-nextpc` from its location. If the interrupt handler gets to run in between, its translation could cause addition of new entries to the jumptable, potentially replacing the mapping between `nextpc` and `tx-nextpc` (that has already been read). Now, when the interrupted thread resumes, it would read an incorrect `tx-nextpc` (because it had previously determined that `nextpc` exists in the table although it has been replaced now), causing a failure. We fix this problem by

[6]On Linux, the interrupt frame field to save and restore the `%ds` segment selector is redundant, because the value in `%ds` register is never overwritten by an interrupt/exception handler. Thus, we translate the instructions that save and restore `%ds` to instructions that save and restore `tx-nextpc-loc` instead.

```
        cmp %reg1, %reg2
        jcc .edge0
        cmp %reg3, %reg4
        jcc .edge1
        mov %reg3, %reg2
        jmp .edge2
.edge0: save_registers_and flags
        clear interrupts
        set nextpc
        jump to dispatcher
.edge1: ... (similar to .edge0)
.edge2: ... (similar to .edge0)
```

Figure 6: **The translated (pseudo) code generated for a code block involving multiple conditional branches (`jcc`).**

clearing the interrupt flag before executing the jumptable lookup logic, and restoring it before branching to `tx-nextpc`, as shown in Figure 3. This was the most subtle issue we encountered in our design.

To avoid concurrency issues arising from execution by multiple CPUs simultaneously, we maintain CPU-private data structures: the dispatcher executes on a CPU-private stack, all temporary variables are stored on the stack, and per-CPU jumptables for indirect branches are used. The dispatcher code is also reentrant and thread-safe (no global variables). The special `tx-nextpc-loc` variable is also maintained per-CPU. The only inter-CPU synchronization required is for mutual exclusion during addition of blocks to the shared code cache[7].

4.2 Code Cache Layout

Figure 6 shows an example of a code block with multiple conditional branches. Notice that `jcc` instructions initially point to the corresponding "edge" code (code which sets `nextpc` and branches to the dispatcher). On the first execution of this edge code, the target of the `jcc` instruction is replaced to point to a code cache address (direct branch chaining). After direct branch chaining, the code cache layout looks very similar to the native code layout, differing only at block termination points.

We experimentally found that the extra edge code introduced for each block results in poorer spatial locality for the instruction cache. This edge code is executed only once at the first execution of the corresponding branch, but shares the same cache lines as frequently executed code. We fix this situation by allocating space for the edge code from a separate memory pool. This allows better icache locality for frequently executed code in the code cache. In our experiments, we observed a noticeable performance improvement after this optimization.

We also found that multiple code copies resulting from CPU-private code caches result in poor icache behaviour. For this reason, we use a shared code cache among CPUs. This does not result in concurrency issues because instructions in code cache are read-only, except the one-time patching of branch instructions for direct branch chaining.

4.3 Function call/return optimization

Our design eliminates most DBT overheads; the biggest remaining overhead is that of indirect branch handling. Each indirect branch is translated into code to first generate `nextpc`, then lookup the jumptable in assembly, and finally, if the jumptable misses, branch to the dispatcher. Even if the jumptable always hits, 2-3x slowdowns are still observed on code containing a high percentage of indirect branches. The most common type of indirect branches are function returns (`ret` instruction on x86). We optimize by using identity translations for `call` and `ret` instructions. In doing so, we let a function call push a code cache address to the stack; at function return, the thread simply returns to the pushed code cache address. This optimization works because after the kernel has fully booted, the return address on stack is only accessed using bracketed call/return instructions. We find that this optimization yields significant improvements.

Because this optimization uses the identity translation for `ret`, all calls *must* push only code cache addresses. This poses a special challenge for calls with indirect operands. Indirect calls of the type "`call *REG`" and "`call *MEM`" are supported on the x86 architecture. Without the call/ret optimization, handling of these instructions is straightforward: the target address (`nextpc`) is obtained at runtime, the jumptable is searched, and if the jumptable hits, the address of the *native* return address is pushed to the stack, and a branch to `tx-nextpc` is executed. If the jumptable misses, the code still pushes the native return address to stack, sets `nextpc` and then exits to the dispatcher; the dispatcher converts `nextpc` to `tx-nextpc` and jumps to it.

With our call/ret optimization, this translation of indirect calls becomes more difficult. First, `nextpc` is obtained at runtime from the operands of the indirect call instruction, and the jumptable is indexed to try and obtain `tx-nextpc`. If the jumptable hits, the address of the *code cache address* corresponding to the *native return address* (let's call this `tx-retaddr`) needs to be pushed to stack. The code at the native return address may not have been translated yet, and so `tx-retaddr` may not even be known at this point. To handle this, on

[7]The cost of this synchronization is small because additions to the code cache are relatively rare in steady state. This synchronization could have been avoided by using multiple per-CPU code caches but that results in poor icache performance as also discussed in Section 4.2.

```
        set nextpc
        obtain tx-nextpc from jumptable
        if not found, jump to .miss
        call tx-nextpc
        <<jmp tx-retaddr>>
.miss:  call dispatcher-entry
        <<jmp tx-retaddr>>
```

Figure 7: The translation code for an indirect call instruction of the type "`call *MEM`" or "`call *REG`", with call/ret optimization. The "`<<jmp tx-retaddr>>`" line represents the full direct branch chaining code (as shown in Figure 2), replacing `tx-retaddr` for `tx-nextpc` (and `retaddr` for `nextpc`).

the jumptable hit path, we emit a "`call tx-nextpc`" instruction immediately followed by an extra unconditional direct branch to `tx-retaddr` (see Figure 7). This extra unconditional direct branch to `tx-retaddr` is supplemented by code to branch to the dispatcher if `tx-retaddr` is not known (similar to how it is done for any other direct branch through "direct branch chaining"). A "`<<jmp tx-retaddr>>`" line in Figure 7 represents the full direct branch chaining code (as shown in Figure 2) for branching to `tx-retaddr`.

If the jumptable misses for `nextpc`, the dispatcher is burdened with having to push `tx-retaddr` to stack before branching to `tx-nextpc`. We handle this case by using a `call` instruction to exit to the dispatcher (instead of using the `jmp` instruction), thus pushing the address of the code cache instruction following the `call` instruction to stack. The dispatcher proceeds as before, converting `nextpc` to `tx-nextpc` and then jumping to it. A future execution of the `ret` instruction will return control to the instruction following the call-into-dispatcher instruction. At this location, we emit a direct unconditional branch to `tx-retaddr` using the same direct branch chaining paradigm, as used for a jumptable hit (see Figure 7).

Note that this call/ret optimization allows code cache addresses to live on globally visible kernel stacks. This global visibility of code cache addresses is acceptable in our design, but will fail if used with previous designs which allow code cache replacement.

4.4 Translator Switchoff and Cache Replacement

Our design creates more complications at switchoff. Because we store code cache addresses in kernel stacks, we must wait for all such addresses to be removed before overwriting the code cache. To ensure this, we iterate over the kernel's list of threads, replacing PC values on each thread's stack to their translated/native values at switchon/switchoff respectively. At switchon, if the translated value of a PC does not already exist, the translation is generated before replacing the value. The PC values are identified by following the stack's frame pointers.

Finally, we discuss code cache replacement. As discussed previously, we do not allow code cache blocks to get replaced in normal operation. It is possible to hit the code cache space limit if translation blocks are frequently created and later invalidated (e.g., due to module loading and unloading). If we hit the code cache space limit, we switchoff the translator and switch it back on to wipeout the code cache to create fresh space. We only need to ensure that no kernel code is executed between the switchoff and switchon; this is done by pausing all CPUs at a kernel entry (except the CPU on which the switchoff/switchon routine is running) till the new cache is operational. We expect such translator reboots to be rare in practice.

5 Implementation and Results

For evaluation, we discuss our implementation, experimental setup, single-core performance, scalability with number of cores, and DBT applications. We finish with a design discussion.

5.1 Implementation

Our translator is implemented as a loadable kernel module in Linux. The module exports DBT functionality by exposing `switchon()` and `switchoff()` `ioctl` calls to the user. A `switchon()` call on a CPU replaces the current interrupt descriptor table (IDT) with its translated counterpart. Similarly, the `switchoff()` call reverts to the original IDT. We also provide `init()` and `finalize()` calls. The `init()` call preallocates code cache memory and initializes the translator's data structures, and the `finalize()` call deallocates memory after ensuring that there are no code cache addresses in kernel data structures.

A user level program is used to start and stop the translator on all CPUs. To start, the program calls `init()` in the beginning. To stop, the program calls `finalize()` at the end. In both cases, the program spawns *n* threads (where *n* is the number of CPUs on the system), pins each thread to its respective CPU (using `setaffinity()` calls), and finally each thread executes `switchon()`/`switchoff()` (for start/stop respectively).

For efficiency, we use a two-level jumptable. Lookup to the first level jumptable does not involve hash collision handling and is thus faster. The second level jumptable is indexed only if the first level jumptable misses. The second level uses linear probing for collision handling and allows up to 4 collisions for a hash location. The most recent access at a location is moved to the front of the collision chain for faster future accesses.

Our code generator is efficient and configurable. It takes as input a set of translation rules. The translation rules are pattern matching rules; patterns can involve multiple native instructions. Our code generator allows codification of all well-known instrumentation applications. Our implementation is stable and we have used it to translate a Linux machine over several weeks without error. Our implementation is freely available for download as a tool called BTKERNEL [1].

5.2 Experimental Setup and Benchmarks

We ran our experiments on a server with 2x6 Intel Xeon X5650 2.67 GHz SMP processor cores, 4GB memory, and 300GB 15K RPM disk. For experiments involving network activity, our client ran on a machine with identical configuration connected through 10Gbps ethernet. We compare DBT slowdowns of our implementation with the slowdowns reported in DRK and VMware's VMM. We could not make direct comparisons as we did not have access to DRK; and VMware's VMM uses more virtualization mechanisms like shadow page tables, which make direct comparisons impossible. Hence, to compare, we use the same workloads as used in the DRK paper [10] (with identical configurations).

All our benchmarks are kernel-intensive; the performance overhead of our system on user-level compute-intensive benchmarks is negligible, as we only interpose on kernel-level execution. We evaluate on both compute-intensive and I/O-intensive applications. I/O-intensive applications result in a large number of interrupts, and are thus expected to expose the gap between our design and previous approaches. Some of our workloads also involve a large number of exceptions/page faults.

We use programs in `lmbench-3.0` and `filebench-1.4.9` benchmark suites as workloads. We also measure performance for `apache-2.2.17` web server with `apachebench-2.3` client, using 500K requests and a concurrency level of 200. We also compare performance overheads during the compilation of a Linux kernel source tree; an example of a desktop-like application with both compute and I/O activity.

We plot performance for two variants of our translator: `default` (all optimizations are enabled), `no-callret` (all except call/ret optimization are enabled). We also implement a profiling client (`prof`) to count the number of instructions executed, the number of indirect branches, the number of hits to the jumptables (first and second level), and the number of dispatcher entries. The corresponding results are labeled `prof-default` (all optimizations enabled) and `prof-no-callret` (all except call/ret optimization enabled) in our figures. Table 3 lists the profiling statistics obtained using the `prof` client.

	System(s)	User(s)	Wall(s)
native.1	249	3633	4280
default.1	235	3625	4257
prof-default.1	263	3631	4295
no-callret.1	417	3647	4565
prof-no-callret.1	504	3670	4666
native.12	275	3704	573
default.12	273	3702	555
prof-default.12	304	3698	560
no-callret.12	491	3726	590
prof-no-callret.12	573	3740	594

Table 2: Linux build time for 1 and 12 CPUs

5.3 Performance

We first discuss the performance overhead on a single core. Figures 8, 9, and 12 plot our performance results. All these workloads intensely exercise the interrupt and exception subsystem of the kernel. The "fast" kernel operations in Figure 8 exhibit less than 20% overhead, except `write` (35% overhead) and `read` (25% overhead). We find 11% improvement in `Protection(Prot)`. Figure 9 plots the performance of fork operations in `lmbench`. Here, we observe 1-1.5% performance improvement with DBT. Similarly, Figure 12 plots the performance on communication-related microbenchmarks. DBT overhead is higher for `tcp` (69%) and `sock` (22%); for others, overhead is less than 15%. DRK exhibited 2-3x slowdowns on all these programs. These experiments confirm the high performance of our design on workloads with high interrupt and exception rates.

5.4 Scalability

To further study the scalability and performance of our translator, we plot performance of different programs with increasing number of processors. Figures 10 and 11 plot the throughput of `filebench` programs with increasing number of cores. To eliminate disk bottlenecks, we used RAMdisk for these experiments. As expected, the throughput increases with more cores, but our translation overheads remain constant. This confirms the scalability of our design (CPU-private structures, minimal synchronization). Interestingly, our translator results in performance improvements of up to 5% for `fileserver` on 8 processors. For other `filebench` workloads, DBT overhead is between 0-10%.

Figure 13 shows the throughput of `apache` webserver, when used with `apachebench` client over net-

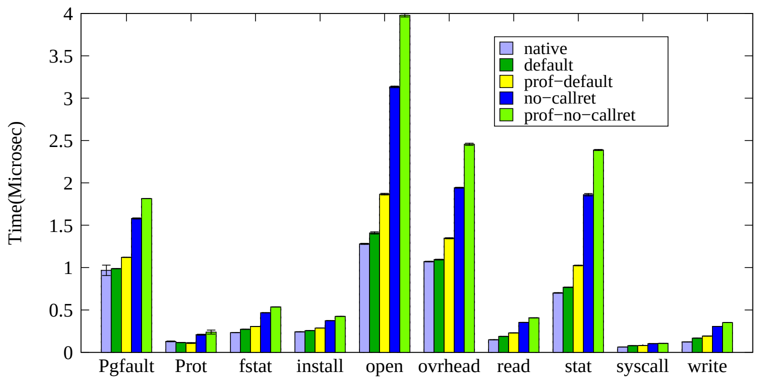

Figure 8: `lmbench` fast operations

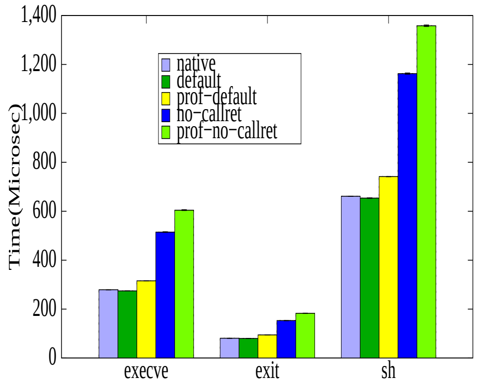

Figure 9: `lmbench` fork operations

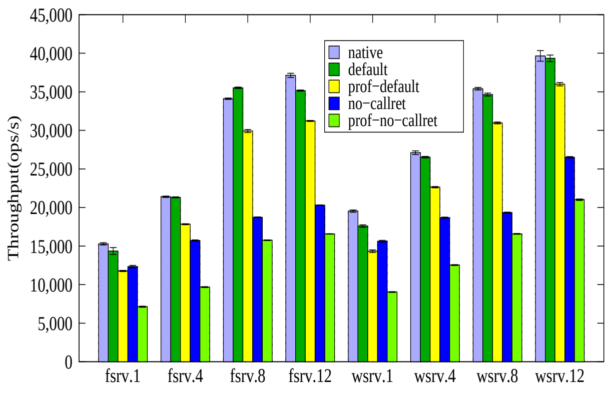

Figure 10: `filebench` on 1, 4, 8, and 12 processors: `fileserver(fsrv)` and `webserver(wsrv)`

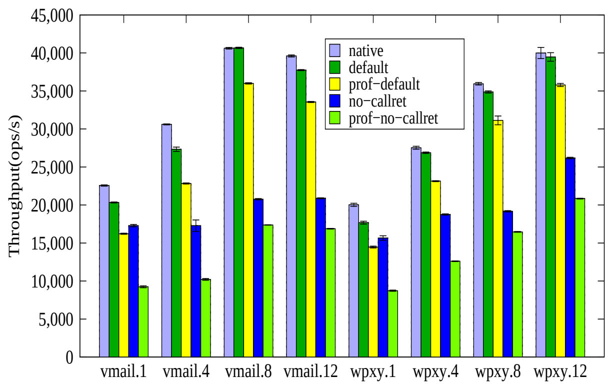

Figure 11: `filebench` on 1, 4, 8, and 12 processors: `varmail(vmail)` and `webproxy(wpxy)`

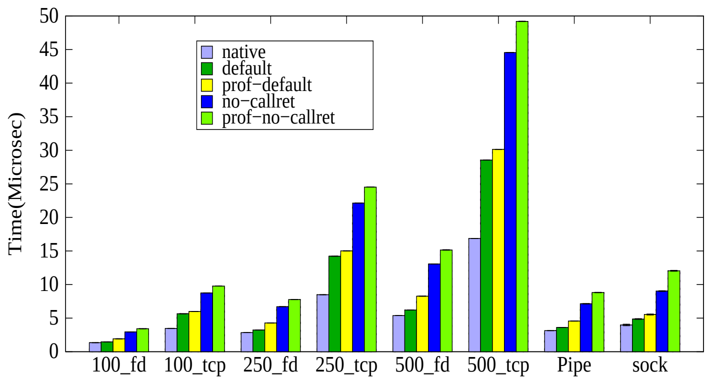

Figure 12: `lmbench` communication related operations

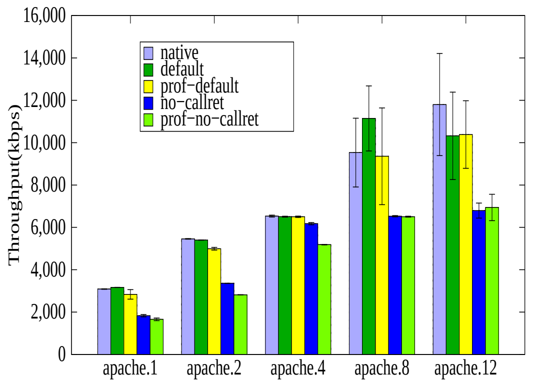

Figure 13: Apache on 1, 2, 4, 8, and 12 processors

	Without Call Optimization					With Call Optimization				
	Total (x1B)	Indirect (x1M)	Jumptable1 (x1K)	Jumptable2 (x1K)	Dispatcher Entries	Total (x1B)	Indirect (x1M)	Jumptable1 (x1K)	Jumptable2 (x1K)	Dispatcher Entries
fileserver	56.54	1285.55	1234107	51205	238907	94.51	337.49	330934	6562	17
webserver	62.25	1335.91	1289146	46674	94351	98.50	401.15	393973	7179	12
webproxy	62.19	1337.71	1287155	50389	169203	100.1	406.47	398994	7485	4
varmail	65.07	1395.25	1337528	57503	224263	109.7	448.73	439561	9170	8
linux build	569.1	16038.0	15622945	342962	72153153	589.9	626.30	613978	12302	33059
apache	55.65	1650.14	1469932	173057	7158220	59.10	202.18	171743	30445	125
tcp500	0.142	3.316	3311	4	1344	0.268	1.934	1934	1	0
pgfault	5.294	158.631	158617	12	2835	5.836	6.915	6915	1	2

Table 3: Statistics on the total number of instructions executed, number of indirect instructions executed, number of first-level and second-level jumptable hits, and the number of dispatcher entries with and without call/ret optimization (obtained by `prof` client). Values in columns labeled `(x1B)` are to be multiplied by one billion, labeled `(x1M)` are to be multiplied by one million and labeled `(x1K)` are to be multiplied by one thousand.

work. DBT overheads are always less than 12%. We observe performance improvement of 17% (for 8 processors) and 2.5% (for 1 processor) on `apache`. DRK reported 3x overhead for this workload. Table 2 shows the time taken to build the Linux source tree using "`make -j`" with and without translation. The time spent in kernel while building Linux improves by 5.6% on one processor, and exhibits near-zero overhead on 12 processors.

Fair comparisons with VMware's VMM are harder, because VMware's VMM also implements many other virtualization mechanisms, namely shadow page tables, device virtualization, etc. However, we qualitatively compare our results with those presented in the VMware paper [2]. The VMware paper reported roughly 36% overheads for Linux build (compared with -5.6% using our tool) and 58% overhead for `apache` (compared with 12% using our tool).

All our performance results confirm that call/ret optimizations result in significant runtime improvements. Table 3 reports statistics on the number of indirect branches (that needed jumptable lookups) with/without the call/ret optimization on a single core. Clearly, the majority of indirect branches are function returns. We also present jumptable hit rates (for both levels) and the number of dispatcher entries for different benchmarks in the table. These statistics were generated in steady state configuration, when the code cache has already warmed up. Without call/ret optimization, the jumptable hit rates for `apache` were 99.56% (89.07% first level, 10.48% second level). With call/ret optimization, the jumptable hit rates were always above 99.99% (84.94% first level, 15.05% second level). In all our experiments, the number of dispatcher entries was roughly equal to the number of jumptable misses.

5.5 Applications

To test functionality, we successfully implemented a shadow memory client using our translator. Our shadow memory implementation maintains type information for each byte in the shadow byte, and this type information flows through memory and registers based on instruction logic. More rules were added to our translation rulebook to implement shadow memory functionality. We modified the kernel to reserve space for shadow memory. We used shadow memory to implement two analyses, namely addressability-checking and sharing-statistics. Like previous work [10], we successfully used addressability checking to ascertain the absence of double-free errors. It is worth noting that relaxing transparency does not effect DBT functionality; our platform is equally usable for bug-finding, or any other type of instrumentation-based analyses. We discuss our second analysis (sharing-statistics) in more detail. For each memory byte, we store in shadow memory its sharing behaviour. In particular, we distinguish CPU-private bytes from CPU-shared bytes. We also track the IDs of the CPUs that accessed that byte and their access type (read-only/read-write). In future work, we are using this information to implement efficient multiprocessor VM record/replay.

To implement fast shadow memory, we allocate a shadow byte at a constant offset from its corresponding physical memory byte. All page table updates are intercepted to make appropriate mappings for shadow memory in virtual address space. The kernel is run translated and each memory instruction is translated to also update the corresponding shadow bytes. Each shadow byte contains two bits per CPU representing `not-accessed`, `read-only-access`, and `read-write-access` states (for a maximum of 4 CPUs). On every memory access, our instrumented code updates the appropriate bits in the corresponding shadow bytes. Because the instrumentation code could clobber CPU flags, we gener-

Num. proc.	read-only private KB	read-only shared KB	read-write private KB	read-write shared KB	System time over-head
1	119.9	0	394,810	0	2.3x
2	130.9	96.1	392,790	1919	3.2x
4	120.5	93.8	392,650	2302	2.7x

Table 4: **Sharing statistics and corresponding performance overheads on `Radix` running on 1, 2, and 4 processors.**

Num. proc.	read-only private KB	read-only shared KB	read-write private KB	read-write shared KB	through-put wrt native
1	274.4	0	460,436	0	43.6%
2	332.1	696.0	427,513	29,664	63.9%
4	378.8	1020.2	387,916	50,199	89.4%

Table 5: **Sharing statistics and corresponding performance overheads on `Fileserver` running on 1, 2, and 4 processors.**

ate code to save/restore flags. We minimize these saves and restores by performing block-level liveness analysis for flags. We also assume that all stack accesses (accesses going through registers `rsp` and `rbp`) are CPU-private and thus do not need to be instrumented. Some translations require temporary registers (i.e., registers that are not present in the native instruction but are needed to store temporary values in the translation); we perform a conservative block-level liveness analysis to intelligently choose these temporary registers at translation time. Instructions with repeat prefix are translated to generate the equivalent code in software. We tested our implementation extensively by running it for different workloads and comparing results. Table 4 and Table 5 presents kernel's byte-level sharing statistics obtained for `Radix` (from SPLASH-2 [18]) and `fileserver` (from `filebench-1.4.9`) respectively. `Radix` has a large memory footprint and exhibits significant kernel activity due to demand paging. These statistics suggest that a large fraction of kernel's memory footprint is CPU-private. We found that generating such statistics at page granularity using page table manipulation (as opposed to byte granularity as done in our shadow memory implementation), presents significantly different results due to false sharing within a page. Generating such statistics through interpretation-based emulators is also error-prone due to the large perturbation in timing behaviour during emulation. The last column in Table 4 and Table 5 lists the performance overhead of our sharing analysis. Our performance overheads are significantly lower than the 10x overhead of DRK's shadow memory implementation [10]. The improvements are due to a combination of a faster kernel-level DBT framework and an optimized shadow memory implementation.

6 Discussion

In summary, our fast DBT design has the following salient features:

- We avoid back-and-forth translation of interrupted/excepting PCs between native and translated values, on interrupt entry and return.
- We assume a large enough code cache, so it can fit all kernel code and does not need cache replacement during normal operation.
- We relax precision requirements on exceptions and interrupts.
- We maintain temporary DBT state on kernel thread stacks and use a reentrant dispatcher.
- We use a cache aware layout for the code cache.
- We use identity translations for function call and return instructions.

Evidently, our DBT design requires knowledge about guest OS internals, to handle special cases appropriately. We also require the guest to obey certain invariants:

- The guest should read the interrupted/excepting PC value (pushed on stack by hardware) mostly through the return-from-interrupt instruction and should be otherwise indifferent to it, except special cases that can be handled specially.
- The guest should not depend on precise exceptions and interrupts.
- The guest should allow a module to access the kernel's list of threads and their call stacks, to allow translation of return address PCs to translated and native values at switchon and switchoff times respectively.
- The guest must obey the stack discipline.
- After it has booted, the guest must use function return addresses only through bracketed call/return instructions, to allow call/ret optimization.

For these reasons, our design is inappropriate for use in VMMs expected to run *any* guest OS. Our scheme can be used however to improve performance for *specific* guest operating systems, using a custom guest-side kernel module in VMMs. Also, our scheme improves performance for several other DBT applications like instrumentation, testing, architecture compatibility, profiling, sandboxing, and dynamic code optimization. On the other hand, some applications which anticipate unconventional guest OS behaviour may not work with our DBT design. For example, it may not be desirable to use our framework for certain security-related applications (e.g., rootkit analysis); such applications may require full transparency to hide from a malicious program, and our framework may violate this requirement.

7 Related Work

User-level DBT frameworks are well understood, with many different systems built on similar techniques: DynamoRio [6], Pin [13], Valgrind [15], vx32 [11], etc. User-level DBT requires stricter transparency requirements, as few assumptions can be made on user program behaviour. In contrast, we show that it is possible to rely on typical kernel behaviour to provide design simplicity and performance for kernel-level DBT.

JIFL [16] is a kernel-level DBT framework that provides an API to instrument system calls. JIFL does not instrument interrupt handlers and kernel threads, making it less comprehensive than our work. Similarly, PinOS is a whole-system instrumentation framework to instrument a guest running paravirtualized in a Xen hypervisor [9], based on the Pin [13] instrumentation framework. Firstly, running in a virtual machine limits execution coverage, as only device drivers for virtual devices get executed. An instrumentation framework for a bare-metal OS (such as ours) can execute drivers for any device, provided the appropriate hardware is available. Secondly, Pin uses a call-based model of instrumentation and so is much slower. PinOS uses similar mechanisms as DRK and VMware to ensure precise exceptions and interrupts. With already high DBT overheads (of Pin), the small overhead of extra mechanisms at interrupts and exceptions (of PinOS) is relatively insignificant.

We compare the differences and similarities between our work and VMware's DBT-based VMM [2, 3] throughout the paper. Unlike VMware, our approach can instrument all device drivers (and not just drivers that get exercised in VM environments), and provides better interrupt/exception performance. However, our design requires guest-specific knowledge. We believe (though do not show) that our techniques can be used in VMMs to improve performance for specific guests, through custom guest kernel modules ("guest tools"). Device passthrough configurations on VMware's DBT-based VMMs can also benefit from our techniques to efficiently interpose on device interrupts.

DRK [10] is perhaps the closest to our work, in our objectives. Unlike DRK, we also provide dynamic translator switchoff functionality. The primary difference, of course, is in our design to handle interrupts and exceptions; our design is simpler and more performant.

8 Conclusion

We present a new design for a kernel-level binary translator that is simpler and performs significantly better than previous work. We take advantage of guest OS properties to relax unnecessary transparency requirements. We have tested our design in a kernel-level binary translator that is capable of attaching/detaching to a running Linux kernel. All workloads perform at near-native speed, in contrast to previous designs which show 2-4x slowdowns on average. We also observe speedups of up to 17% over native, on some programs. We expect the low-overhead translation to enable more kernel-level DBT applications.

Acknowledgements

We thank our anonymous reviewers for their valuable feedback which helped in making this paper significantly better. We especially thank our shepherd Rebecca Isaacs for her detailed and constructive suggestions that greatly helped us in preparing a better camera-ready version. We are grateful to the wonderful people on the FreeBSD, NetBSD, Windows and the BarrelFish mailing lists for their prompt and valuable responses to our queries. We thank DRK [10] authors for the nice figure in their paper explaining DBT, from which our Figure 1 is inspired. The first author is partially supported through the IBM Graduate Fellowship. This work is partially supported by NetApp's Faculty Fellowship program.

References

[1] BTKernel: Fast Dynamic Binary Translation for the Kernel. `https://github.com/piyus/btkernel`, as on September 15, 2013.

[2] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS '06*.

[3] O. Agesen, A. Garthwaite, J. Sheldon, and P. Subrahmanyam. The evolution of an x86 virtual machine monitor. *SIGOPS Oper. Syst. Rev.*, 44(4), Dec. 2010.

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[5] B. N. Bershad, D. D. Redell, and J. R. Ellis. Fast mutual exclusion for uniprocessors. In *ASPLOS '92*.

[6] D. Bruening. *Efficient, Transparent and Comprehensive Runtime Code Manipulation*. PhD thesis, MIT, 2004.

[7] D. Bruening, Q. Zhao, and S. Amarasinghe. Transparent dynamic instrumentation. In *VEE '12*.

[8] E. Bugnion. Binary translator with precise exception synchronization mechanism. US Patent 7516453, filed June 2000.

[9] P. P. Bungale and C.-K. Luk. PinOS: a programmable framework for whole-system dynamic instrumentation. In *VEE '07*.

[10] P. Feiner, A. D. Brown, and A. Goel. Comprehensive kernel instrumentation via dynamic binary translation. In *ASPLOS '12*.

[11] B. Ford and R. Cox. Vx32: lightweight user-level sandboxing on the x86. In *USENIX ATC'08*.

[12] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *USENIX Security '02*.

[13] 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 *PLDI '05*.

[14] S. Nagarakatte, S. Burckhardt, M. M. Martin, and M. Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *PLDI '12*.

[15] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[16] M. Olszewski, K. Mierle, A. Czajkowski, and A. D. Brown. Jit instrumentation: a novel approach to dynamically instrument operating systems. In *EuroSys '07*.

[17] D. Song et. al. Bitblaze: A new approach to computer security via binary analysis. In *ICISS '08*.

[18] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA '95*.

[19] Q. Zhao, D. Koh, S. Raza, D. Bruening, W.-F. Wong, and S. Amarasinghe. Dynamic cache contention detection in multi-threaded applications. In *VEE '11*.

[20] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong. How to do a million watchpoints: efficient debugging using dynamic instrumentation. In *CC'08/ETAPS'08*.