Fast Dynamic Binary Translation for the Kernel

Piyus Kedia and Sorav Bansal
IIT Delhi

This is a PDF version with notes. You can find the PPTX version at http://www.cse.iitd.ernet.in/~sbansal/talks/btkernel.pptx
Firstly, what is Dynamic Binary Translation and what is it used for. Dynamic Binary Translation or DBT is the technique to transform the code as it executes, and this is done for a variety of purposes. Some examples are OS virtualization, testing and verification of compiled programs, profiling and debugging, software fault isolation, dynamic optimizations, program shepherding, and many more. DBT is used for a variety of purposes in various application domains.

Applications of Dynamic Binary Translation (DBT)

- OS Virtualization
- Testing and Verification of Compiled Programs
- Profiling and Debugging
- Software Fault Isolation
- Dynamic Optimizations
- Program Shepherding
- ... and more
Here is a short introduction to how Dynamic Binary Translation, or DBT, works. Execution typically starts at the dispatcher, which translates one basic block at a time, and transfers control to it. The block executes, but terminates with a branch to dispatcher instruction, thus returning control back to the dispatcher. This loop continues forever.

Of course, translating every basic block on every execution is expensive, and so translation is typically done only once, and then cached for future executions in a code cache.
Before translating a block, the dispatcher first checks if the block is already cached. If so, it jumps to it.Else, it takes the slower path of actual translation. Because a piece of code usually executes thousands, millions or even billions of times, the small cost of one-time translation is easily amortized.
User-level DBT is relatively well understood, and many previous works have demonstrated near-native performance for several application-level workloads.

Kernel-level DBT however requires mechanisms to also efficiently handle exceptions and interrupts. The problem is bigger at the kernel-level because the expected rate of interrupts and exceptions in the kernel is significantly higher than the expected rate of signals in user-level processes.

Current kernel-level binary translators simply import the signal-handling mechanisms used at user-level, to the kernel. As I will show next, this imposes huge overheads on many performance-critical applications. Some case studies that we look at are the Vmware’s software virtualization platform which uses DBT to virtualize the guest OS, and DynamoRio-Kernel, which implements DBT-based instrumentation for the kernel.
I next discuss in more detail how kernel-level DBT works. DBT is typically implemented through a loadable kernel module. For full translation coverage, a DBT module needs to interpose on all the entry points of a kernel, i.e., all the gates from which execution can enter the kernel.
For example, this means, that it needs to interpose on all entries through the interrupt descriptor table. Hence the original interrupt descriptor table which points to the appropriate handlers needs to be replaced with another shadow table that now points to the dispatcher instead.
Let’s look in more detail at what does a dispatcher do, on an entry through the interrupt descriptor table. Before transferring control to the code cache, the dispatcher first converts interrupt state on stack pushed by hardware to its native values.

**Before transferring control to the code cache, the dispatcher:**
1. Converts interrupt state on stack to native values (e.g., PC)
Here is a figure showing the program counter PC pushed by hardware on stack. Notice that with DBT, the pushed address will always be a code cache address, and the dispatcher is required to convert it into its corresponding native guest value. This is required so that if the guest ever inspects the stack, it always observes expected values there.

**Before transferring control to the code cache, the dispatcher:**
1. Converts interrupt state on stack to native values (e.g., PC)
What does the dispatcher do?

Before transferring control to the code cache, the dispatcher:
1. Converts interrupt state on stack to native values (e.g., PC)
2. Emulates Precise Exceptions

The second thing that the dispatcher does is emulating precise exceptions.
A precise exception is the property of an architecture, whereby the hardware guarantees that before the execution of an exception handler, all instructions up to the executing instruction should have executed and everything afterwards must not have executed.

**What does the dispatcher do?**

**Precise Exceptions**

Before the execution of an exception handler, all instructions up to the executing instruction should have executed, and everything afterwards must not have executed.

**Before transferring control to the code cache, the dispatcher:**

1. Converts interrupt state on stack to native values (e.g., PC)
2. Emulates Precise Exceptions
For example, if an exception occurred in the middle of the execution of a push instruction, all earlier changes made by this instruction are undone, or rolled back, before transferring control to the exception handler.

In a binary translated environment, a single guest instruction could be translated to multiple host instructions. If an exception occurs at one of these host instructions, all state updates made by the previous instructions need to be rolled back. This involves not just a direct cost of emulating the precise exception, but also the indirect cost of having to structure a translation such that it can be rolled back.
Finally, a binary translator needs to provide the guarantee of precise interrupts. This is a guarantee by the translator that the execution of an interrupt handler will only commence at a valid guest instruction boundary.

**What does the dispatcher do?**

**Before transferring control to the code cache, the dispatcher:**
1. Converts interrupt state on stack to native values (e.g., PC)
2. Emulates Precise Exceptions
   - Rollback partially executed translations
3. Emulates Precise Interrupts
Thus, if an interrupt was received in the middle of the emulation of a push instruction, the interrupt is “delayed” till the next guest instruction boundary. The implementation of delaying an interrupt involves incurring extra traps and invalidations of the code cache, and is expensive.

Overall, while these mechanisms are necessary to guarantee complete transparency, they also result in significant overhead.
Effect on Performance

Applications with high interrupt and exception activity exhibit large DBT overheads.

As an end result, applications with high interrupt and exception activity exhibit large DBT overheads.
Here is some data from Adams and Ogesen’s paper from Vmware in ASPLOS 2006, where they reported up to 123% DBT overhead for an Apache webserver. Notice that applications that incur fewer interrupts show less overhead. For example, the compute-intensive SPEC Int benchmark suite shows only 2.9% overhead, while compiling a Linux kernel exhibits around 27% overhead. Notice that the overhead is largely proportional to the interrupt activity of the workload. While Apache experiences a large number of interrupts due to network activity, SPECInt experiences almost no interrupts, except perhaps the timer interrupt. Also note that in all these experiments, only the kernel’s code is translated and the user-level code is run natively or untranslated.
The same paper also showed overhead results on microbenchmarks, which make it clearer that the overhead is largely correlated to the interrupt and exception activity. For example, the largeRAM microbenchmark results in a large number of page faults, and shows roughly 90% overhead over native. Similarly, the forkwait microbenchmark, which involves forking a large number of processes before joining them, exhibits 600% or 6x overhead, due to the large exception activity in this microbenchmark.
This becomes even clearer with “nanobenchmarks”, wherein one opcode is repeatedly run in a loop. In this case, I show two nanobenchmarks: divzero, which executes an instruction that causes a div-by-zero exception, and syscall that invokes a software interrupt. Translation overheads for the two are 260% and 850% respectively, confirming that exceptions and interrupts are the primary culprits behind the translation overheads.
Similar overheads have been reported in another work on kernel-level binary translation, DynamoRio Kernel or DRK, published at ASPLOS 2012. They reported up to 350% overhead for workloads like fileserver, webserver, webproxy, and apache. These overheads were also attributed to the overheads of interrupt and exception handling.
In contrast, our dynamic binary translator, which we call BTKernel, achieves near-native performance on all these benchmarks. The orange bars, which are hardly visible here show the overheads of our translator. Our overheads are typically less than 2%, with the maximum overhead of around 10% for the varmail benchmark.

BTKernel is implemented as a loadable kernel module in unmodified Linux.
The central observation behind our work is that fully transparent execution is not required. The OS kernel rarely relies on precise exceptions. The kernel rarely relies on precise interrupts. The kernel seldom inspects the PC address pushed on stack. It is only used at the time of returning from interrupt using the _iret_ instruction.

Fully Transparent Execution is *not* required

- The OS kernel _rarely_ relies on precise exceptions
- The OS kernel _rarely_ relies on precise interrupts
- The OS kernel _seldom_ inspects the PC address pushed on stack. It is only used at the time of returning from interrupt using the _iret_ instruction.

The central observation behind our work is that fully transparent execution is not required. The OS kernel rarely relies on precise exceptions. The kernel rarely relies on precise interrupts. The kernel seldom inspects the PC address pushed on stack, and it is mostly used only at the time of interrupt return in bracketed call/return patterns.
Using these observations, we show that faster execution is possible. We leave the code cache addresses in kernel stacks by making an interrupt or exception jump directly into the code cache, bypassing the dispatcher. This also means that we allow imprecise interrupts and exceptions. And the special cases where the kernel indeed relies on the correctness of PC values on stack, we handle them specially. In the rest of the talk, I will discuss how this is done in more detail.

As an aside, it is interesting to know that both previous DBT implementations, namely Vmware and DRK, also do not provide full transparency, in that it is possible for a guest to determine if it is running natively or translated. Our work further relaxes transparency to achieve better performance.

Faster Execution is Possible

- Leave code cache addresses in kernel stacks.
  - An interrupt/exception directly jumps into the code cache, bypassing the dispatcher.

- Allow imprecise interrupts and exceptions.

- Handle special cases **specially**.
Firstly, the shadow interrupt descriptor table which pointed to the dispatcher in the previous designs, is now made to point directly to the respective code cache addresses.
For this, the first blocks of the appropriate handler code is pre-translated and stored in the code-cache. As you may imagine, this brings out many correctness concerns.
The first correctness concern is that a read or write of the interrupted PC address on stack will return incorrect values. As I said earlier, fortunately this is rare in practice and can be handled specially. I will use an example to illustrate this more clearly.
Consider an exception handler, that performs a load on the address where the interrupted PC is stored. If control flow decisions are made based on the value returned by this load, the kernel could potentially crash. One concrete example, where this can happen, is Linux's exception tables used in its page fault handler.
Exception Tables in Linux

- Page faults are allowed in certain functions
  - e.g., `copy_from_user()`, `copy_to_user()`.

- An exception table is constructed at compile time
  - contains the range of PC addresses that are allowed to page fault.

- At runtime, the faulting PC value is compared against the exception table
  - Panic only if PC not present in exception table

On Linux, page faults are allowed in certain functions handling user pointers, such as `copy_from_user`, and `copy_to_user`. An exception table is constructed at compile time, which contains ranges of PC address that are allowed to fault. For example, this table will contain the PC addresses corresponding to the `copy_from_user` and `copy_to_user` functions. At runtime, the faulting PC value is compared against the exception table, and a panic is triggered only if the PC is not present in the exception table. If the faulting PC is found in the exception table, the corresponding user-level fault handler is invoked.
Notice that the faulting PC in our system will now be a code cache address, and thus will not be present in the exception table. This means that the kernel will now incorrectly panic.

We implemented a simple solution to this problem by adding the code cache addresses corresponding to the addresses already present in the exception table, also to the same exception table. This ensures correct execution in this case.
Another example with a similar access pattern of the interrupted PC value, is present in Microsoft’s Windows NT Structured Exception Handling implementation, where __try and __except constructs are used in C and C++ for exception handling.
The syntax for the try/except construct is given on the left, where potentially faulting code is enveloped with a try/except block, where the except block implements the fault handler.

On your right is an example usage of this pattern in the kernel. The copy_from_user() function from our previous example can now be put inside the try block, while the handler code which signals the process can be put in the except block.

This try and except mechanism in the Windows kernel is also implemented using per-function exception tables similar to Linux, and will also cause similar problems for our binary translator. Fortunately, the same solution of modifying the exception tables appropriately works here too.

In general, for a well-designed OS, any part of the kernel that relies on the interrupted PC value must also allow a kernel module to also influence its behavior, because the PC value of a kernel module is only determined at module load time. This capability provides enough power to our DBT module to handle these special cases.
We tested this hypothesis by studying a variety of operating systems. There are more examples in the paper of such patterns that we found and their solutions. In our experience, all such cases can be nicely handled without the need of expensively maintaining full transparency.
Correctness Concerns

1. Read / Write of the faulting PC address on stack will return incorrect values.

2. Code-cache addresses will now live in kernel stacks.
   • What if code-cache addresses become invalid?

The second correctness concern has to do with the code-cache addresses now living in kernel stacks. For example, what will happen if code-cache addresses become invalid due to cache replacement. A later return through that address will create panic.
To explain this better, consider this scenario, where Thread1 is executing in the kernel, and its interrupted PC, which will now be a code cache address, is present on stack. A context switch occurs at this point, and another thread, Thread2 resumes execution. Meanwhile, the code cache block for the translated PC gets replaced. When Thread1 resumes back again, it will cause a crash, as the translated PC is no longer valid.
This problem is handled by disallowing cache replacement in the code cache. Typically kernel code sections are small, and cache replacement is usually not required. For example, a code cache of around 10MB suffices for Linux. We also do not move or modify code cache blocks, once they are created which ensures that a code cache address remains valid for the execution lifetime.

In the rare event that the code cache does get full, due to repeated module loading and unloading for example, we implement a dynamic “translator switchoff” feature. We allow DBT to be switched on and off at runtime, in a dynamic manner. A consecutive switchoff and switchon effectively causes a translator reboot, which effectively results in flushing the code cache and starting afresh.
I discuss the dynamic switchon / switchoff feature in a bit more detail. This feature is unique to our translator implementation, and has not been supported by previous DBT implementations to the best of our knowledge. Essentially, dynamic switchoff and switchon is implemented by replacing all kernel entry points by shadow or original values respectively. We also iterate over the list of kernel threads to convert PC values stored on thread stacks to native or translated values respectively. A translator reboot, which is effectively a switchoff followed by a switchon, results in flushing the code cache.

Dynamic Switchon / Switchoff

- Replace all entry points with shadow / original values
  - e.g., for switchoff, replace shadow interrupt descriptor table with original

- Iterate over the kernel's list of threads
  - Identify PC values in thread stacks and convert them to code cache / native values

- Translator reboot (switchoff followed by switchon) flushes the code cache
Correctness Concerns

1. Read / Write of the faulting PC address on stack will return incorrect values.

2. Code-cache addresses will now live in kernel stacks. What if code-cache addresses become invalid?

3. Imprecise Interrupts and Exceptions.

Finally, the last correctness concern is the violation of precise interrupts and exceptions.
Interestingly, for all the kernels we studied, we did not find an instance where the kernel depends on precise exception and interrupt behavior.
Finally, I point out that direct entries into the code cache introduce new reentrancy and concurrency challenges that we have correctly handled in our implementation, and there is a discussion in the paper.
I next talk about some more optimizations that helped in making our DBT tool faster. The first optimization is an L1-cache aware code cache layout. And the second optimization has to do with optimized translations for function calls and returns.

Optimizations that worked

- L1 cache-aware Code Cache Layout

- Function call/return optimization
This figure shows the dispatcher together with the code cache. The white rectangles represent code blocks terminated by code to jump back to the dispatcher, shown as yellow lines. A translator typically implements direct branch chaining, which means that after the first execution of a block, the block is chained directly to its successor, thus eliminating the need to branch to the dispatcher on every block execution. We call this code to branch to the dispatcher shown in yellow, the edge code. Notice that the edge code is executed only once, on the first execution of a block; however, it shares the same cache lines as all other code. This is a classic case of hot code (represented by white boxes) sharing cache lines with cold code (represented by yellow lines), resulting in overall poor instruction cache utilization. We mitigate this problem by allocating edge code from a separate memory pool, which we call the edge cache. This effectively causes hot code to remain spatially close and thus result in much better instruction cache utilization. This we found results in noticeably better performance.
The second optimization we implemented was to use identity translations for the function call and return instructions. In any dynamic binary translator, user-level or kernel-level, indirect branches cause a big overhead, as they involve translating guest PC values to code cache addresses at runtime. Function returns are by-far the most common type of indirect branches. Also, function returns are typically only used through bracketed call/return patterns. Because we allow code cache addresses to live in kernel stacks, without fear of panic, we can also support identity translations for call and return instructions. This means that a guest call instruction is translated to a host call instruction, thus resulting in the push of a code cache address to stack. Similarly, a return instruction pops the code cache address and jumps to it. As I will discuss in my results, this improves performance significantly.

This optimization requires careful handling of call instructions with indirect targets, and there are more details in the paper.
I now come to our experiments. I will show you three types of results. First, I will talk about BTKernel’s performance w.r.t. native execution. Then I will show some statistics collected through our binary translator, and correlate them with its performance. Finally, I will discuss our experience with some applications of kernel-level binary translation.
This chart shows the throughput of an Apache webserver natively and with BTKernel in blue and orange respectively, higher is better. The third bar in gray shows BTKernel's performance with the function call/ret optimization disabled. BTKernel performs within a few percentage points of native execution. As I have already discussed, this is a significant improvement over the 2-3x overheads reported in previous work. Notice that BTKernel's throughput is _higher_ than native, for 8 processors in this experiment. We attribute this performance gain to perhaps better icache locality. The near-native performance of BTKernel makes it possible to use dynamic binary translation for runtime optimizations.
This graph shows the throughput of the fileserver benchmark. Notice that while BTKernel's performance is always close to that of native, turning off the callret optimization makes a significant difference to the performance. This confirms the high utility of our callret optimization.

Also, even without callret optimization, our tool performs significantly better than previous work, due to our faster handling of interrupts and exceptions.
Finally, here are some timing results on microbenchmarks that intensively exercise the interrupt and exception subsystems of the kernel. Here, lower is better. Notice that BTKernel adds only small overheads, if any, even for highly interrupt and exception intensive applications. In fact, in all these three benchmarks, DBT actually improves performance because of better icache locality in the code cache! There are many more such results in the paper.
This table shows some statistics collected by BTKernel for Apache and another benchmark involving a Linux kernel build. These statistics were collected through a profiling instrumentation client implemented on top of BTKernel. The left table shows statistics without callret optimization, and the right table shows statistics with callret optimization enabled. Without callret optimization, Apache executes around 56 billion instructions and exits to the dispatcher around 7 million times. In contrast, with the callret optimization, Apache exits to the dispatcher only 125 times. For any binary translator, reducing the number of dispatcher exits is one of the most significant optimizations. Here we see that the callret optimization drastically reduces the number of dispatcher exits.
We implemented a few different applications using BTKernel, and I discuss one here, namely a shadow memory implementation that identifies the CPU private and CPU shared bytes in the kernel address space. This data is useful for many purposes, including studies on kernel scalability on multicore architectures and multiprocessor VM record replay. Our overheads for this implementation of shadow memory range from 20% to 300%. This is a significant improvement over a similar shadow memory implementation done by DRK, which shows 10x overheads in comparison.
To summarize, we used the following techniques to implement fast dynamic binary translation for the kernel:
We avoid back-and-forth translation between native and translated values of interrupted PC
We relax precision requirements on exceptions and interrupts
We use cache-aware layout for the code cache
We use identity translations for the function call/ret instructions

This results in near native performance implementation of a dynamic binary translator for the kernel. Our tool requires guest-specific knowledge, but can work for unmodified guests. Hence, it is suitable for use in Virtual Machine Monitors to optimize performance for specific guests. Our implementation is quite stable and we have run BTKernel on our desktop for several weeks without error. Our code is available publicly at this URL.
Thank You.