Presentation of ”Fast Dynamic Binary Translation for the Kernel” by Piyus Kedia and Sorav Bansal

Jan Wróblewski

21 października 2015
Virtual machine is an application that emulates guest operating system on the host operating system.

Full virtualization occurs when 100% of the hardware is emulated by the virtual machine and no OS components (e.g. kernel) are shared by the guest and host OSes.

To enable that, all instructions run on the guest OS have to be emulated by the VM. The process of translating guest instructions into appropriate instructions/calls on the host is called binary translation.

When that translation happens at runtime, it is called dynamic binary translation.
Dynamic binary translation

We can split the virtualization into:

- user-level code virtualization – efficient solutions with overhead 0%-30% that use code caching already exist;
- kernel-level code virtualization – efficient solution was created, described in this presentation.

The obstacles in kernel-level code virtualization include:

- interrupts and exceptions;
- concurrency and reentrancy (ability of methods to be interrupted and then resumed);
- kernel entry points (e.g. syscalls);
- hardware drivers.
Current solutions

"Efficient, transparent, and comprehensive runtime code manipulation” PhD thesis by Derek Bruening contains description of an efficient DBT.

A **dispatcher** is the code that translates guest code blocks into host code. A **code block** is a sequence of instructions with an unconditional jump at the end.

Currently ran code blocks from the guest are translated straightforwardly and cached for future reuse. The important part is swapping jump instructions for a jump back to the dispatcher code.
Translation example

**original:**
```assembly
add %eax, %ecx
cmp $4, %eax
jle 0x40106f
```

**fragment7:**
```assembly
add %eax, %ecx
cmp $4, %eax
jle stub0
jmp stub1
stub0:  mov %eax, eax-slot
mov &dstub0, %eax
jmp context_switch
stub1:  mov %eax, eax-slot
mov &dstub1, %eax
jmp context_switch
```
Optimization in current solutions

A fast translation uses an optimization called **direct branch chaining**.

After making a jump to cached code block from a cached one, the jump to dispatcher in the previous one is replaced with the address of the subsequent block.

\[
\text{block1 \{code; jmp dispatcher\} } \rightarrow \text{dispatcher } \rightarrow \text{block2 \{code\}}
\]

is replaced with

\[
\text{block1 \{code; jmp cachedBlock2\} } \rightarrow \text{block2 \{code\}}
\]
Interrupts

Kernel-level DBT is mostly about interrupts. When an interrupt/exception occurs in the guest OS, currently ran code is supposed to be interrupted to run the code that handles it.

Normally this is done by pushing the program counter (PC) register on the interrupt stack, invoking the interrupt code and running the return-from-interrupt iret instruction.
Current kernel-level DBT

Emulation is done by

- rolling back machine state to native instruction boundary from when the interrupt occured;
- inserting a software interrupt to the translation;
- invoking the handler.

This is to ensure that the interrupt won’t hapen while inside of a translation of a single native instruction.

Emulation of `iret` instruction plus rolling back the machine state is costly, resulting in 2-5x slowdowns on kernel-intensive code.
Faster kernel-level DBT is done by:

- disallowing interruptions in the dispatcher (turning them off), making them appear only in user-level code or cached code;
- translating `iret` instruction into host `iret` instruction instead of a dispatcher call.

To achieve that, translation transparency is sacrificed.
Faster design - problem 1

While that seems straightforward, it contains the problem of properly handling PC value. After such emulation it will contain address of the code from the code block cache, which is different from the VM’s memory address with the loaded code.

However, PC value is rarely read with other purpose than jumping back to the code. Study of kernels has shown that it occurs only at few places, e.g.:

- Handling page fault in linux (e.g. copy_from_user()) to check if wrong memory was supplied or it was a kernel bug;
- Handling Restartable Atomic Sequences on some architectures;
- try/except mechanism in Windows NT.

The solution to this was to handle each PC-sensitive case separately by a kernel module.
The second problem was that code block cache addresses started appearing in kernel structures, so referenced cached blocks couldn’t be removed from the cache.

The problem is solved by assuming that the whole kernel will fit into the cache. That is pretty realistic, since Linux kernel with few loadable modules fits in 10MB.

In rare cases where e.g. modules were loaded/unloaded repeatedly, cache would eventually fill. In such cases a costly operation of cleaning everything up and fixing kernel addresses was performed.
The last problem is that the interrupts are not handled precisely, i.e. at different time.

This did not matter in practice, since study has shown that the interrupt code in guest OS kernels did not depend on PC value or other registers.
This design was implemented in a loadable Linux kernel module.

The speed was further improved by removing emulation of call and ret instructions – making them native instructions instead of emulated jumps (which slowed everything 2-3x).
Benchmark - fileserver and webserver

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

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

The end results were very good:

- some operations showed no overhead or even small improvement (forking);
- some showed a small slowdown (15% for most, 20% for fast kernel operations, 25% for read, 35% for write, 22% for sock networking);
- some showed substantial slowdown (69% for tcp networking); This is still much better than previous 2-3x slowdown.

To compare, VMware paper published results of 58% overhead for apache tests, and this translation showed only 12% overhead.