Same bits, different meaning – when direct execution based simulation becomes complicated

Evgeny Yulyugin, Intel Corporation, Stockholm, Sweden (evgeny.yulyugin@intel.com)

Abstract— **Intel® 64 architecture processors are constantly evolving, with new generations regularly being introduced on the market. A new processor is usually backwards compatible and includes all the software-visible functionality of previous generations. This allows existing software to run on the new hardware without recompilation or modifications. However, the guarantee of compatibility can break in certain scenarios when software is running in a virtual machine. We discovered that certain machine instructions behave differently on past and present generations of AMD and Intel processors. We have developed new methods and tools in Wind River® Simics® to allow correct and fast execution across generations.**

Keywords—component; virtual machine monitor, virtualization, Intel VT-x, Simics, live migration, virtual platform

I. INTRODUCTION

To make sure that software support is present when new hardware is released to the market, software development must be shifted left in the product life cycle to the pre-silicon phase. Therefore a high performance virtual environment is required to run and debug full software stacks far ahead of silicon availability. A preferred way to achieve high speed pre-silicon virtual platforms for Intel 64 platforms is to use Intel® Virtualization Technology for Intel® 64 and IA-32 architectures (Intel® VT-x) to execute instructions directly on the host [1].

A classic understanding of hardware-assisted virtualization developed by Popek and Goldberg in 1974 [2] is that all processor instructions can be classified depending to how they lend themselves to virtualization. *Innocuous* instructions execute similarly both inside and outside a virtual machine (VM) and pose no threat to correctness. *Privileged* instructions may affect critical system resources and should not be allowed to be executed directly inside a VM. Therefore, processors typically support trapping on privileged instructions before they are executed, so that they can be analyzed and safely emulated if needed.

Popek and Goldberg's theory is applicable to the case of virtualizing a processor of a particular generation on the same processor generation. But the theory is no longer applicable for the case when the guest and host processors are from different generations. This is caused by the fact that some *innocuous* (and thus not interceptable) instruction encoding byte sequences behave differently on different processor generations.

Software simulation is an important use case when the virtual machine guest's processor generation can differ from the host. Software simulation or virtual platform solutions like Simics [1] use hardware supported virtualization (which itself corresponds to Popek and Goldberg's theory) to run portions of guest code directly on the host processor when possible. This usually makes the simulation significantly faster than a JIT-based implementation. In case of software simulation, an older host is usually used to emulate newer hardware.

The only previously known mitigation technique for the problem of instruction encodings that behave differently on different generations without a trap is to limit the guest processor capabilities (available instruction set extensions) declared inside a virtual machine to a safe subset [3]. This places unnecessary constrains on system performance as the most recent instruction sets are not allowed. Moreover this approach is not applicable for virtual platforms like Simics which aim to provide virtualized environments representative of future architectural generations. An alternative approach would be a pure simulation using interpretation [4] or just-in-time (JIT) compilation of target code to host code [5] of all machine instructions, but that would be prohibitively slow for production environments.

The differences in behavior between the problematic instructions we have identified are very subtle, making it impossible to identify and work around after-the-fact. We found that existing hardware virtualization mechanisms alone are inadequate to prevent and/or detect cases of incorrect execution of such machine instructions. This research was aimed to extend a standard trap-and-emulate virtualization technique [6] with support for the non-interceptable instructions we have identified as problematic.

The solution for the identified problem is developed, implemented and tested in Simics [1, 5]. Simics is a software-based solution that runs on existing Intel 64 platforms while providing access to features from future platforms. Simics can run unmodified firmware, Unified Extensible Firmware Interface (UEFI), Basic Input/Output System (BIOS), and operating system code and simulate both instruction-set and platform-level differences between generations and variants of Intel and other platforms. Simics has three ways to run target instructions: an interpreter, a JIT compiler, and *VMP*.

Simics VMP uses Intel VT-x to run Intel 64 code efficiently on host Intel 64 processors. Simics VMP runs in VMX root mode – the hypervisor mode while guest software runs in VMX non-root mode. Processor behavior in VMX non-root mode is restricted and modified to facilitate virtualization. Instead of their ordinary operations certain instructions and events cause VM exits – transition to a virtual machine monitor (VMM) running in VMX root mode [7]. The VMM uses the virtual machine control structure (VMCS) to manage guest software running in VMX non-root mode. This allows the VMM to retain control of processor resources.

Simics combines Intel VT-x execution with JIT compilation and interpretation in order to handle instructions not found on the host but present in the target architecture. This implementation relies on getting VM exits for unknown or modified instructions. It also limits VMP to updating the processor and memory state that the host instructions update.

II. PROBLEM STATEMENT

A. *Reused encodings*

It was found that in AMD* Family 10h processors, released in 2007 [8], AMD decided to take a valid encoding (`REP` (byte `0xF3`) prefix + `BSR` (bit scan reverse)) that was unlikely to be used by software and repurpose it for a new instructions – `LZCNT` (count the number of leading zero bits). The prefix had no functional effect on `BSR`. Basically, this means that the same encoding corresponds to different instructions depending on the processor generation. These instructions have different semantics and cannot be safely interchanged, so an attempt to execute `LZCNT` on older hosts using a technique like VMP that just passes the guest code directly to the host will silently provide incorrect results. The instruction was later added to 4th generation Intel® Core™ processors (formerly Haswell, released in 2013). Presence of the instruction can be determined through CPU identification (`CPUID`) information. Figure 1 shows a simple program in C that can be used to verify the presence of `LZCNT` support.

```
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>

int main () {
        uint32_t ecx, cpuid_leaf = 0x80000001;
        asm volatile("cpuid" :"=c"(ecx) :"a"(cpuid_leaf) :"ebx", "edx");
        bool has_lzcnt = ecx & (1 << 5);
        printf("LZCNT is%s supported.", has_lzcnt ? "" : " not");

        uint32_t out, in = 0x11aa00bb;
        // "lzcnt ebx, eax" or "bsr ebx, eax" depending on CPU generation
        asm volatile(".byte 0xf3, 0x0f, 0xbd, 0xd8" :"=b"(out) :"a"(in));
        bool lzcnt = out == 0x3;
        printf(" 0xF30FBD corresponds to %s.\n", lzcnt ? "LZCNT" : "BSR");

        return 0;
}
```

Figure 1. LZCNT consistency verification tool

The verification tool runs the `CPUID` command to check whether the `LZCNT` instruction is supported or not. Then it executes byte sequence `0xF30FBDD8` that corresponds to either "`LZCNT EBX, EAX`" or "`REP BSR EBX,` EAX" depending on the processor generation. The execution outcome in EBX register is used to determine which instruction was executed. Expected outputs of the tool are either "LZCNT is not supported. 0xF30FBD corresponds to BSR." or "LZCNT is supported. 0xF30FBD corresponds to LZCNT."

The result of the application running inside a virtual machine powered by Microsoft Hyper-V* running in live migration mode depends on the host's processor generation. Execution on 3rd generation Intel Core processor or older produces expected result – "LZCNT is not supported. 0xF30FBD corresponds to BSR." But execution on 4th generation Intel Core or newer results in unexpected and inconsistent output – "LZCNT is not supported. 0xF30FBD corresponds to LZCNT." This is caused by the fact that CPUID bit corresponding to LZCNT is hidden from the virtualized environment [3], but the encoding still gets executed as the new instruction. Such replacement leads to an incorrect execution results in a virtual environment. However the error cannot cause any harm to a virtual machine monitor or any other application running outside of the VM.

An opposite situation happens when software simulation frameworks like Simics use direct execution to speed up simulation. A model of a newer processor reports support for the new instruction to the code running inside the simulation, while the older hardware used to directly execute the code does not support the new instruction. The hardware silently executes the encoding as an old instruction. The test program running inside a simulated machine supporting LZCNT on a hardware not implementing the instruction will produce unexpected result – "LZCNT is supported. 0xF30FBD corresponds to BSR."

Together with LZCNT, 4th generation Intel Core processors have introduced TZCNT (count the number of trailing zero bits) instruction as part of BMI (bit manipulation) instruction set architecture extension. TZCNT encoding denotes to REP prefix + BSF (bit scan forward) on older hardware. Similarly to LZCNT and BSR, these instructions denote different operations and cannot be safely interchanged.

B. *XSAVE instruction family*

The XSAVE instruction set architecture extension first introduced in 2nd generation Intel Core processors (formerly Sandy Bridge) together with Intel® Advanced Vector Extension (Intel® AVX). The extension supports saving and restoring of processor state components (registers or parts of registers) and is typically used for context switch and process initialization by system software. The extension includes instructions that save processor state to memory (XSAVE, XSAVEC, XSAVEOPT, XSAVES), restore processor state from memory (XRSTOR, XRSTORS) and instructions operating on extended control registers (XSETBV, XGETBV). All the instructions have XCR0 (extended control register 0) as an implicit parameter. The XCR0 register contains a state-component bitmap that specifies state components that software has enabled. State components supported by a processor may vary from generation to generation. In general, each state component corresponds to a processor feature. Features that require use of the XSAVE feature set for their enabling are **XSAVE-enabled** features. XSAVE-enabled features include Intel AVX, Intel AVX2, Intel AVX-512 and Intel® Memory Protection Extension (Intel® MPX). This list can be extended in future Intel 64 processors.

The XSAVE feature set virtualization through direct execution does not cause any issues if guest software has only enabled state components supported by host hardware. Otherwise direct execution of XSAVE management instructions should not be allowed because actions that should be done on the new state will be lost. This is caused by the fact that hardware does not allow to set unsupported feature bits in XCR0 control register and the XSAVE instructions ignore all requested through arguments but not set in XCR0 register features. XSAVES and XRSTORS execution in VMX non-root (guest) mode is controlled by VMCS (virtual machine control structure). XSETBV is unconditionally intercepted by a virtual machine monitor running in VMX root mode. Other instructions do not have any specific treatment in VMX non-root mode. Their execution is controlled by CR4 (control register 4).

Unlike LZCNT and TZCNT, execution of XSAVE feature set instructions can be disabled and thus intercepted by a virtual machine monitor. But this will also disable execution of **all** XSAVE-enabled features and therefore significantly limit the number of instructions that can be simulated by direct execution, seriously reducing virtualization performance. Currently XSAVE-enabled features include several hundred instructions and all of them can't be executed directly because of the XSAVE feature set instructions if guest software has enabled any state component not supported by the hardware.

C. User-mode instruction prevention (UMIP) feature

User-mode instruction prevention (UMIP) is a new security feature present in the latest Intel 64 processors. If enabled, it prevents the execution of the following instructions if current privilege level (CPL) is greater than zero: SGDT (Store Global Descriptor Table), SIDT (Store Interrupt Descriptor Table), SLDT (Store Local Descriptor Table), SMSW (Store Machine Status Word), STR (Store Task Register). If any of these instructions is executed with CPL > 0, a general protection exception is issued if UMIP is enabled.

Behavior of SGDT, SIDT, SLDT and STR instructions in VMX non-root mode is controlled using "Descriptor-table exiting" field of virtual machine control structure, thus making them interceptable by a virtual machine monitor but no such possibility exists for SMSW. The SMSW instruction reads a value of control register 0 (CR0) and a virtual machine monitor can manage the value that the guest software can get, but it cannot intercept the execution of the instruction using existing hardware virtualization technique. A virtual machine monitor running on hardware that does not support UMIP should not allow direct execution of the instruction to be able to model the security check correctly if guest has enabled it.

D. Known use cases

We actually observed incorrect behavior of major operating systems inside a virtual machine. As a first example, the Android* 4.4 powered by Linux kernel version 3.10 is known to use LZCNT instruction regardless of whether hardware support through CPUID. An attempt to limit reported processor capabilities artificially will not help in this case. Virtualization through direct execution will cause Android running the guest to crash in case if the host hardware does not support LZCNT instruction because it will be executed as BSR and thus will produce incorrect results.

It was also discovered that recent Android* and Linux* systems use the TZCNT instruction. The TZCNT instruction encoding will be executed as BSF instruction on old hardware. The key difference between TZCNT and BSF instructions is that TZCNT provides operand size as output when source operand is zero while the content of destination operand is undefined in the case of BSF instruction. TZCNT instruction also sets carry flag if the input was zero and clears it otherwise while BSF keeps it undefined [7]. Experiments on Intel Core i7-2600 processor that does not support TZCNT show that BSF instruction always clears carry flag and does not modify output register if the source operand is zero. However GCC compiler can emit TZCNT instruction instead of BSF even if the target processor does not support it [10], thus making it relatively popular in modern software. Use of TZCNT instruction is typically harmless in a virtualized environment because of the subtle difference but still can produce incorrect execution results when replaced with BSF.

A family of XSAVE instructions is widely used by all known contemporary operating systems including Microsoft Windows*, Linux*, FreeBSD*, etc. The instructions are mostly used for context switch by operating systems. Major hypervisors as well as Simics itself use XSAVE feature set instructions to manage guest states.

Finally, User-mode instruction prevention is known to be used by the latest Linux* operating systems. UMIP support was introduced in Linux* kernel in November 2017 [11].

III. PAGE SCANNING

The identified instructions may seem innocuous at first, but their operation differs on different processor generations. Similarly to the class of privileged, identified instructions threaten execution correctness in virtualized environments, but unlike the former, they cannot be trapped by hardware.

The key to solving this problem is to make sure that no such instruction is allowed to be directly executed in a virtualized environment – they all have to be emulated the same way as privileged instructions. Their detection has to be done in advance since the problem cannot be corrected after the fact. It is hard to detect presence of an instruction in memory for several reasons: variable length instructions with unspecified boundaries, intermixing of code and data, and cross-page effects. However, it is possible to prove the opposite – that no given opcode is present on a page.

In hardware, virtual machines code is organized into code pages – aligned blocks of fixed size (typical size is 4kB). Every page has permission flags attached to it. Safe execution is achieved through a combination of scanning code memory in advance before it is allowed to be executed, and emulating the code on pages that are suspected to contain unsafe instructions.

Before allowing any code from a new page to be executed, the whole page is scanned against patterns describing known dangerous instructions. For example, the following byte sequences are treated as potential `LZCNT`: [`0xF3, 0x0F, 0xBD`] and [`0xF3, REX.W, 0x0F, 0xBD`], where `REX.W` is a `REX` prefix and has a value between `0x40` and `0x4F`. If the page has no matches, then it is marked as safe, and any further VM execution from it is performed directly by hardware. If a page is deemed unsafe, it is removed from the direct execution mode and is processed using software emulation techniques.

To maintain high execution speed, a new pattern matching decoder has been developed. The decoder goes through each code page of the simulated system trying to find byte sequences corresponding to dangerous instructions. If a code page contains at least one such sequence, execution from it has to be emulated in software. Otherwise, hardware-based virtualization techniques can be applied to the page. The scanner stops execution once it finds the first pattern corresponding to a potentially dangerous instruction. The aim of the pattern matching decoder is to find whether a page has at least one sought-for pattern or not.

Code pages of a virtual machine can be classified as three different types:

- Safe – pages that contain no problematic instructions,
- Unsafe – pages possibly containing dangerous instructions,
- Not yet executed pages.

It should be noted that the pattern decoder does not do any control-flow or history analysis, so it cannot definitely determine that a page does contain a dangerous instruction. For example, figure 2 shows instruction sequence that will be identified as potential `LZCNT` by the decoder because the sequence contains continuous byte sequence `0xF30FBD`. Similarly data stored on code pages can be incorrectly classified as an instruction, giving false result.

```
Encoding:      Instruction:
0fbaf30f       btr ebx, 0xf
bdddccbbaa     mov ebp, 0xaabbccdd
```

Figure 2. Instruction sequence incorrectly identified as LZCNT

The content of a code page has to be rescanned if there are modifications to the page, so all virtual machine pages accessible by a direct execution mode should be write-protected. This is required to protect against self-modifying code generating dangerous instructions. Overall code page life cycle is illustrated on figure 3. The initial state is “Not allocated page”.

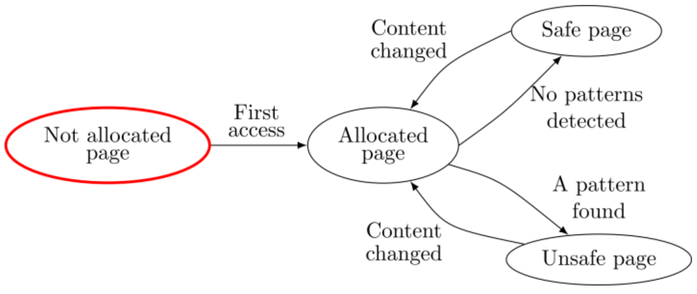

Figure 3. Code page life cycle

Guest software can’t determine whether it is running in a virtualized environment or not because any execution of the described problematic instructions will be always intercepted and emulated in sofware thus always providing the same result. Previously execution of the instructions could be done either using hardware or software virtualization techniques depending on guest code because of optimization techniques aimed to avoid hardware VM exits [12].

IV. Results

Figure 4 shows time required to boot various operating systems using different simulation modes: hardware virtualization (baseline VMP), virtualization with the described technique enabled (adapted VMP) and pure software simulation (disabled VMP). The measurements were done for a Skylake-based simulated platform on a Skylake-based host system – 3.6 GHz Intel® Xeon® E3-1270 v5 processor with 64 GB memory. To estimate the performance impact of new approach, the simulator was forced to do the scanning for unsafe instructions and emulation of execution from unsafe code pages despite the fact that it was not required because host and guest processor generations were identical.

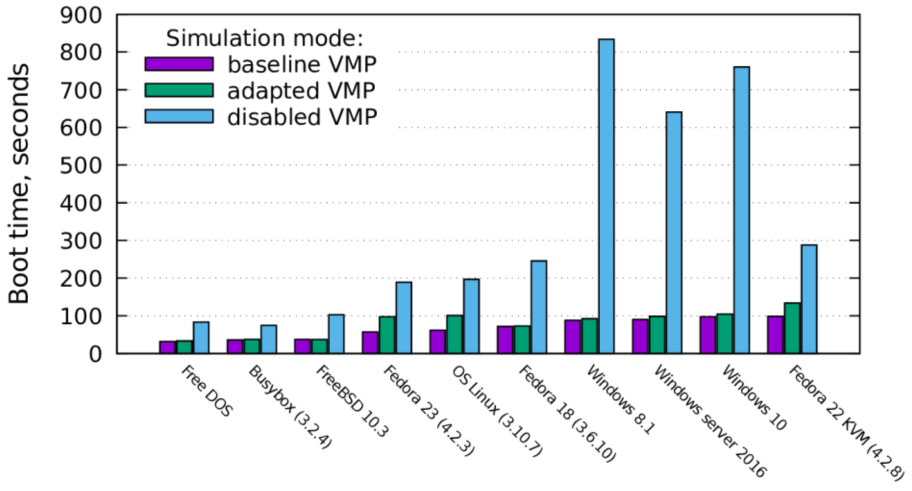

Figure 4. Boot time in different simulation modes

The measurements show that the new approach is on average 1.18 times slower than the traditional (but unsafe) virtualization but 3.32 times faster than pure software emulation. The scanning itself does not introduce any visible performance degradation. For example, boot time of FreeBSD 10.3 didn't change and no unsafe pages were detected during the boot.

A. *Fedora 23 boot*

Fedora 23 (kernel version 4.2.3) boot was investigated deeply because of the biggest performance degradation of adopted VMP execution mode – 1.70 times slower than baseline VMP. The virtual system had two simulated Intel Xeon processors (formerly Skylake) and 4 GB memory. 66 of total 8706 scanned pages were excluded from direct execution because of a pattern corresponding to `TZCNT`. Almost 9.7 million `TZCNT` of 40.1 billion total instructions were executed from 64 different code pages. `LZCNT` pattern was found only once and no `LZCNT` instruction was simulated during the boot.

The measurements show that the dangerous instructions are not actively used by the software – only 0.02% of overall executed instructions were unsafe. Only 0.77% of code pages were excluded from direct execution as potentially containing dangerous instructions and 99.5% of them actually contained `TZCNT` instruction. The number of blocked pages can look negligible from the first point of view, but the blocking of the pages noticeably increased number of guest code emulated using the interpreter and JIT from 1.3% to 9.7%. Direct execution mode exclusion of the pages also led to a significant raise of rather expensive switches between direct execution and software emulation modes – 3.7 million switches for baseline VMP and 4.3 million for adapted VMP. Increased number of simulation mode switches and amount of software emulated instruction together led to 1.70 times raise of the boot time.

V. CONCLUSIONS

Existing hardware-based virtualization technology alone no longer can be trusted to correctly simulate future hardware. The combination of hardware-assisted virtualization and described software techniques is required to achieve correct and fast execution in virtual environments.

We have started this work when we were working on Haswell processor model which introduced first two known instructions with described property (`LZCNT` and `TZCNT`) in Intel 64 processors. The presented technique allowed us to provide fast enough software models that were used for pre-silicon software development inside and outside of Intel. The new technique allowed us to successfully boot all major operating systems and hypervisors using a model of 4th generation Intel Core processor while running on 2nd generation hosts, which would have been impossible with the traditional virtualization technique.

We discovered that modern Intel 64 processors contain eight instructions with this property. Our experiments show that these instructions are not used frequently, so only a small number of code pages will be affected in a virtualized environment, and our performance data support this statement.

ACKNOWLEDGMENT

The author would like to thank members of Simics team who participated in discussions related to the discovered problem and who provided valuable feedback about the article.

REFERENCES

[1] D. Aarno, J. Engblom, “Software and System Development using Virtual Platforms – Full System Simulation with Wind River Simics,” Morgan Kaufmann Publishers, 2014.

[2] G.J. Popek, R.P. Goldberg, “Formal Requirements for Virtualizable Third Generation Architectures,” Communications of the ACM, val. 17, pp. 412-421, July 1974.

[3] T. Mitch, “Understanding Microsoft Virtualization Solutions, From the Desktop to the Datacenters,” 2nd ed., Microsoft Press, 2010.

[4] D. Mihoka, S. Shwartsman, “Virtualization Without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure,” ISCA-35 Proceedings of the 1st Workshop on Architectural and Microarchitectural Support for Binary Translation, 2008.

[5] P. Magnusson, F. Dahlgren, F. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, B. Werner, ”SimICS/sun4m: A Virtual Workstation,” Proceedings of the 1998 USENIX Annual Technical Conference, June 1998.

[6] K. Adams, O. Agesen, “A Comparison of Software and Hardware Techniques for x86 Virtualization,” In Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS XII). ACM, pp. 2-13, October 2006.

[7] Intel® Corporation, “Intel® 64 and IA-32 Architectures Software Developer’s Manual,” volumes 1-4, May 2018.

[8] AMD, “Software Optimization Guide for AMD Family 10h and 12h Processors”, 2011.

[9] Intel® Corporation, “Intel® Architecture Instruction Set Extensions and Future Features Programming Reference,” May 2018.

[10] P. Bonzini, “x86: emit `TZCNT` unconditionally,” April, 2012. URL: https://gcc.gnu.org/ml/gcc-patches/2012-04/msg01765.html

[11] N. Ricardo, “x86: enable User-Mode Instruction Prevention,” LWN.net, November 2017. URL: https://lwn.net/Articles/705877

[12] O. Agesen, J. Mattson, R. Rugina, J. Sheldon, “Software Techniques for Avoiding Hardware Virtualization Exits,” Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12), pp. 373-385, 2012.