Measuring the Performance Impact of Branching Instructions

Lukas Beierlieb[1], Lukas Iffländer[2], Aleksandar Milenkoski[3], Thomas Prantl[1] and Samuel Kounev[1]

[1]*University of Würzburg, Am Hubland, Würzburg, 97074, Germany*

[2]*Deutsches Zentrum für Schienenverkehrsforschung, August-Bebel-Straße 10, Dresden, 01069, Germany*

[2]*Cybereason, Theresienhöhe 28, München, 80339, Germany*

Abstract

With the continuing rise of cloud technology, hypervisors play a vital role in the performance and reliability of current services. Hypervisors implement interfaces providing call-based connectivity to hosted virtualization-aware virtual machines. One of them is the hypercall interface, allowing hypervisor service requests from virtual machines. A hypercall injection tool measuring hypercall execution times must minimize internal overhead. Among other things, limiting logging to strictly required information is crucial. However, checks for values to log for every injection requires executing many branch instructions. We assess the performance difference between using and avoiding such branching, measured by the hypercall throughput of the injector tool.

Keywords

measurement, performance, branching

1. Introduction

Today, hypervisors are virtually omnipresent. They are widespread throughout data centers, functioning as the backbone of cloud computing [1] by allowing for server consolidation with huge benefits in efficiency and flexibility. Hypervisors are also prevalent in modern desktop and workstation infrastructures [2]. This extends to Microsoft shipping their Hyper-V hypervisor directly with many versions of the Windows Operating System (OS), and in some cases, even activating it by default [3].

Virtualization allows to create virtual instances of physical devices called Virtual Machines (VMs). In a virtualized environment, governed by a hypervisor, VMs share resources. Hypervisors implement interfaces providing call-based connectivity to virtualization-aware hosted VMs. One of them is the hypercall interface, allowing for VMs to request services from the

SSP'21: Symposium on Software Performance, November 09–10, 2021, Leipzig, Germany

lukas.beierlieb@uni-wuerzburg.de (L. Beierlieb); IfflaenderL@dzsf.bund.de (L. Iffländer); milenkoski.aleksandar@gmail.com (A. Milenkoski); thomas.prantl@uni-wuerzburg.de (T. Prantl); samuel.kounev@uni-wuerzburg.de (S. Kounev)
https://go.uniwue.de/beierlieb (L. Beierlieb); https://go.uniwue.de/prantl (T. Prantl); https://go.uniwue.de/kounev (S. Kounev)
0000-0002-0877-7063 (L. Beierlieb); 0000-0002-8506-2758 (L. Iffländer); 0000-0002-4780-6963 (A. Milenkoski); 0000-0003-4044-8494 (T. Prantl); 0000-0001-9742-2063 (S. Kounev)

hypervisor. Hypercalls are software traps from a VM to the hypervisor. Before introducing x86 hardware virtualization in 2006, hypercalls were one solution to run virtualized OSs. Nowadays, while technically not required, hypercalls are still a common utility to improve efficiency or offer additional features.

The crucial role that hypervisor play in today's infrastructure requires robustness and high performance, among other properties. We proposed a framework to help testing these qualities [4]. On the one hand, the framework supports the logging of values and execution times if required, on the other, it should be able to inject calls at a high rate, due to low overhead. In this paper, we investigate whether or not there is a performance penalty to evaluating multiple `if` statements for optional logging.

Other works concentrate on reducing the cost of branches if that cannot be removed [5, 6, 7].

The remainder of this paper is structured as follows: Section 2 presents background knowledge about virtualization, Hyper-V, hypercalls, and branching in processors. Next, Section 3 introduces the measurement approach and environmental conditions during its realization; Section 4 then presents and discusses the measurements. Finally, Section 5 concludes the paper.

2. Background

In a non-virtualized scenario, an OS manages physical hardware (i.e., processor, memory, and IO devices) and provides and schedules physical resource accesses for applications running on top. Virtualization describes the concept of introducing an abstraction layer above the hardware. That layer, called the hypervisor or Virtual Machine Monitor (VMM), provides a set of virtual resources, which can form multiple virtual machines managed by independent OSs.

One way to classify hypervisors is by whether directly control the hardware or run on top of an OS [8]. The former approach is called a Type-1 or bare-metal hypervisor and can utilize its full control for increased performance. Type-2 or hosted hypervisors, on the other hand, can reduce their complexity by relying on the OS to take care of most of the hardware management.

Para-virtualization applies changes to the source code of OSs themselves. These modifications allow the hypervisor and VMs to interact more efficiently, e.g., by using abstract IO interfaces instead of emulating existing, physical devices, reducing overhead and improving performance [9].

Hyper-V is an x86_64 hypervisor developed by Microsoft [10]. It is a Type-1 hypervisor. Thus, it directly controls the hardware. However, to avoid limiting it to specific hardware configurations or bloat the code base with countless device drivers, Hyper-V uses a microkernel-based architecture. A specialized VM called the root partition always runs an instance of Windows on top of Hyper-V to provide management features and device drivers. Guest VMs (also called guest partitions) can run para-virtualized if they support it but also can use unmodified OSs, in which case Hyper-V provides emulated devices.

Hyper-V offers various interfaces for VM-Hypervisor and VM-VM communication [11]: Privileged register and memory accesses, emulation of privileged instructions, IO ports, inter-VM communication via the VMBUS, and hypercalls. Similar to applications requesting services from the OS by issuing system calls, guest OSs can call into the hypervisor with hypercalls.

Hypercalls are triggered by special processor instructions that transfers execution control from VM to hypervisor. Hyper-V expects a call code present in a specified register beforehand. Also, parameters for the call can be placed in registers and memory, depending on the calling convention. After processing a hypercall, Hyper-V returns a result value, which indicates either a successful execution or an error code (e.g., missing privilege, out of memory).

Apart from virtualization terms, some background knowledge about CPU execution is required to understand the goal of this paper. Modern processors try to execute as many instructions as possible at a time. Branching instructions are problematic because the processor does not know which instructions follow until the branch is fully executed. CPUs deploy branch prediction to guess and speculatively execute further instructions, which helps to keep the performance penalty down for well predictable branches [6].

3. Methodology

The hypercall injector consists of three parts. There has to be a description of the hypercall workload, called the campaign file. It consists of binary data, describing which hypercalls with which parameters should be executed, and also if any delays should be waited between calls. A Windows kernel driver, the so-called injector module, executes the campaign. It loads the campaign into memory, executes the call and delay instructions one by one, and in between logging values if required and storing them in a file in the end. These two are connected by a desktop application. It takes the path to the campaign file as an argument, along with which values to log. Then, it prepares and loads the injector kernel module, and passes all the information required to execute the campaign correctly.

We support the logging of timestamp pairs for each action (hypercall, delay), execution times (difference between the timestamps), as well as result values and all output values. Hyper-V stores output values on a dedicated memory pages. Saving these values therefore requires storing 4MB of data, which is slow. Thus, always storing all values is not viable; the slowdown would be too big for a test campaign, e.g., a stress test ignoring log values and injecting hypercalls as fast as possible. That is why the required log values are passed to the desktop application. Yet, there are two different ways of handling optional logging values during execution. Firstly, the probably more natural approach is too place `if` statements in the main injection loop. Listing 1 shows a pseudo-C-code of this approach. The loop works through the actions of the campaign. We are interested in issuing as many calls as possible, so there must not be any delays in the workload. Accordingly, we can skip the handling of delays. Without any logging, hypercalls need their parameters prepared in memory and the processor registers, and the call must be invoked. Additionally, depending on the requested log values, the loop has to optionally take timestamps, calculate execution time, and store values to the log buffer. These `if` statements introduce a lot of branches. It has to be noted however that they always take the same branch as the requested log values cannot change during the campaign. The speculative execution engine of the processor should have no problem getting the branch right.

The alternative approach is the implementation of the loop for every combination of log values, which does not require any `if`s. This still requires branching depending on the log values, but only once at the start. As a drawback, this approach causes lots of duplicate code, increasing driver size (not really an issue at this scale) but also drastically hurts maintainability. To test

```
// prepare
while (/* more to execute */) {
    switch (/* type */) {
    case TYPE_WAIT:
        // sleep and maybe log times, details left out
        break;
    case TYPE_CALL:
        // prepare memory for call
        if (/* timesteps or execution time requested */)
            // take start time
        // issue hypercall
        if (/* timesteps or execution time requested */)
            // take end time
        if (/* timesteps requested */)
            // store time stamps
        if (/* execution time requested */)
            // calculate execution time and store
        if (/* result value requested */)
            // store result value
        if (/* output values requested /*)
            // store output memory page
    }
}
```

Listing 1: Main injection loop with `if` statements

if the chaos of copy-pasted injection loops is worth it, we want to determine the maximum hypercall throughput of both configurations.

We tried focusing the workload to spend as much time as possible injecting hypercalls in order to spot an eventually existing performance difference. By choosing an invalid hypercall code (hypervisor detects quickly and returns with an error) and passing no parameters, time for other tasks is reduced as much as possible. The measurements are performed on an Lenovo ThinkPad P1 with an Intel i7 9850H CPU and 2×16GB of 2667MHz DDR4 RAM. Unfortunately, the laptop form factor means that the CPU will probably thermally throttle under high load, i.e., rapid hypercall injection. Maximum throughput during short burst as well as continuously sustained high load throughput are of interest, so we chose the following methodology:

A single test campaign contains 50 million invalid hypercalls. Each configuration starts with a processor that has been idling for at least five minutes, to ensure fair starting temperatures in the heat sink. The campaign is executed 35 times in a row without breaks. The desktop application is adapted to log the execution time the kernel driver requires to perform the workload. Tested configurations are of course the branching and branchless variants, each compiled once as debug executables and once with Release optimizations by Microsoft's C++ compiler. The next section is now going to illustrate the measurement results.

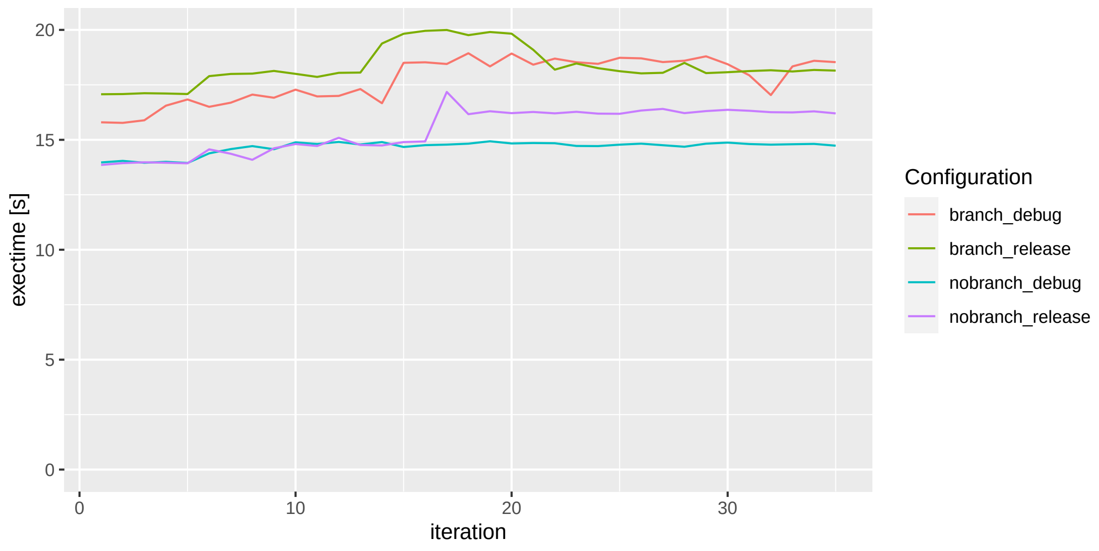

Figure 1: Comparison of different injection variants; 35 successive execution times for 50 million invalid hypercalls each

4. Results

Figure 1 shows the results of the measurements. The x-axis shows the number of iterations of the 50 million-call campaign, the y-axis shows how long it took to execute the particular workload. Each configuration has its own line and color.

The first few iterations show the peak performance numbers, with a non-throttled CPU. The debug-compile of the branch variant managed 3.17 million calls per second (mcps) at best. The release version actually performed worse, with only 2.93 mcps. Both versions of the no-branch implementation performed significantly better, within measurement tolerance of each other around 3.6 mcps.

Except for the debug-compile without branches, all the lines showed similar throttling behavior. Minor penalties after around five runs, and yet more slowdown at around 15 repetitions. Interestingly, the branchless debug build was the least affected by thermal conditions.

Overall, the original research question can be clearly answered. Removing the branches yields a significant benefit in this scenario, even though there should not occur any mispredictions.

5. Conclusion

In this work, we shared our findings regarding the cost of including `if` statements into the main loop of a hypercall injector tool. Results showed that even though the same branches are always taken, the penalty is high enough to make an significant difference in hypercall throughput. In conclusion however, it should be stated that working dozens of slight variations of the same loop is not practical if changes happen to the code occasionally. However, a compromise of maintenance and performance should be achievable: In performance-critical cases that require no logging or only time measurements, a dedicated, no-branch loop can be implemented. For other scenarios, which are more concerned with values that hypercalls return, performance is hindered by logging but usually not of great importance - here, a branching implementation can save lots of duplicate code.

Acknowledgments

This work was written in Cooperation with SPEC RG Security. Lukas Iffländer made his initial contributions as group leader with University of Würzburg.

References

[1] S. Srivastava, S. Singh, A survey on virtualization and hypervisor-based technology in cloud computing environment, International Journal of Advanced Research in Computer Engineering & Technology (IJARCET) 5 (2016).

[2] K. Miller, M. Pegah, Virtualization: virtually at the desktop, in: Proceedings of the 35th annual ACM SIGUCCS fall conference, ACM, 2007, pp. 255–260.

[3] Virtualization-Based Security: Enabled by Default, https://techcommunity.microsoft.com/t5/Virtualization/Virtualization-Based-Security-Enabled-by-Default/ba-p/890167, ???? Accessed: 2019-10-27.

[4] L. Beierlieb, L. Ifflander, A. Milenkoski, S. Kounev, Towards Testing the Performance Influence of Hypervisor Hypercall Interface Behavior (2019).

[5] W. Hwu, T. M. Conte, P. P. Chang, Comparing software and hardware schemes for reducing the cost of branches, ACM SIGARCH Computer Architecture News 17 (1989) 224–233.

[6] S. McFarling, J. Hennesey, Reducing the cost of branches, ACM SIGARCH Computer Architecture News 14 (1986) 396–403.

[7] H. Kim, J. A. Joao, O. Mutlu, C. J. Lee, Y. N. Patt, R. Cohn, Vpc prediction: reducing the cost of indirect branches via hardware-based dynamic devirtualization, ACM SIGARCH Computer Architecture News 35 (2007) 424–435.

[8] Z. Gu, Q. Zhao, A State-of-the-art Survey on Real-time Issues in Embedded Systems Virtualization, Journal of Software Engineering and Applications 05 (2012) 277–290. doi:10.4236/jsea.2012.54033.

[9] H. Fayyad-Kazan, L. Perneel, M. Timmerman, Full and para-virtualization with xen: a performance comparison, Journal of Emerging Trends in Computing and Information Sciences 4 (2013) 719–727.

[10] H. Fayyad-Kazan, L. Perneel, M. Timmerman, Benchmarking the performance of microsoft hyper-v server, vmware esxi and xen hypervisors, Journal of Emerging Trends in Computing and Information Sciences 4 (2013) 922–933.

[11] Hyper-V Top Level Function Specifications, 2019. URL: https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/tlfs, [Online; accessed 5. May. 2021].