Università degli Studi di Torino

DIPARTIMENTO DI INFORMATICA
Corso di Laurea Triennale in Informatica

Tesi di laurea

Comparing KVM and Xen tracing overhead

Candidato:
Lorenzo Dentis
Matricola 914833

Relatore:
Enrico Bini
University of Turin

Correlatore:
Dario Faggioli
SUSE Software Solutions Italy

Anno Accademico 2020/2021

Contents

List of Figures

Acknowledgments

Part of Chapter 1 is borrowed from the thesis by Cristian Monticone [23]. Part of Chapter 2 is borrowed from the thesis by Giuseppe Eletto [22]. Part of Chapter 3 is borrowed from the thesis by Marco Perronet [24] and Lorenzo Brescia [18].

To my parents, for supporting me and believing in me. To my professors who inspiered me and fed my curiosity, to my colleagues and friends with whom I have grown. And finally I would like to thank Bini Enrico and Faggioli Dario, who helped me develop this research and write this thesis.

Declaration of originality:

> *"Dichiaro di essere responsabile del contenuto dell'elaborato che presento al fine del conseguimento del titolo, di non avere plagiato in tutto o in parte il lavoro prodotto da altri e di aver citato le fonti originali in modo congruente alle normative vigenti in materia di plagio e di diritto d'autore. Sono inoltre consapevole che nel caso la mia dichiarazione risultasse mendace, potrei incorrere nelle sanzioni previste dalla legge e la mia ammissione alla prova finale potrebbe essere negata."*

Abstract

Virtualization is expanding in fields where tracing is crucial, for instance self-driving cars, not only in software development but for quality certification too. In those systems the computational power is low, therefore it is really important to keep the tracing overhead as low as possible. This thesis analyzes how tracing impacts two different hypervisors: Xen and KVM. Different benchmarks were run with the aim to stress both the host system and the virtualized guest, using Ftrace and Xentrace as tracing frameworks. The benchmarks were run on every combination of host and guest with enabled and disabled tracing, generating the results that were later analyzed.
Chapter 1 describes in details the notion of virtualization, its various shades and its components. Chapter 2 makes an introduction to the Xen hypervisor. Chapter 3 contains an overview of tracing, ftrace, xentrace and others component to make the tracing. Chapter 4 is about comparing the overhead generateb by tracing in KVM versus the overhead generated by tracing in Xen.
During this work, the configuration of the machines and the versions of software was as follows:

- Host's Linux kernel: 5.14.9 (distro openSUSE Tumbleweed)
- Xen Hypervisor: 4.15.0
- Virtual machine manager: 3.2.0
- Guest's Linux kernel: 5.14.9 (distro openSUSE Tumbleweed)
- MMTest: 0.24

Chapter 1

Virtualization

1.1 Introduction

Through *virtualization*, it is possible to make different virtual computing environments over the same physical machine. Each of these virtual computer systems is called *guest machine* with the inner operative system called *guest OS*. Similarly, we refer to the physical machine as *host machine* and, if present, its operative system is called *host OS*.

Generally speaking, virtualization means running multiple operating systems simultaneously, isolated from each other, with specific applications running on top of them. Implementing such a mechanism may not be easy. Guest OSes may be different, so the computer architecture of the guest machines. With different guest and machine Instruction Set Architectures (ISAs), the guest low level instructions has to be translated on the fly and than executed back in the host processor. This process, called *emulation*, highlights a key problem: the efficiency. A virtualization technology needs to be fast with a low execution overhead. But there are many other significant problems, for example hardware and software resources as computational power, network infrastructure, memory and others must be shared among multiple guest machines.

To achieve this complex and ambitious goals, it is necessary to implement abstract layers between the guest environments and the physical one. Specifically, the virtualization infrastructure consists of at least two software layers: virtualization and management. The virtualization layer aggregates and abstracts the host hardware to a standardized set of resources. The management one lets the user manage and monitor the hosts resources and all the physical and virtual infrastructure. From the virtualization point of view there are many different technologies, such as emulation or full-virtualization. All the various kind of technologies will be discussed further in a dedicated section.

1.2 Advantages and disadvantages

Using virtualization technologies has a number of significant advantages:

Costs First, due to the virtualization, the set of hardware resources are reduced, decreasing operative and maintenance costs. Energy demand and cooling system are reduced together with the the environmental impact. In a society whose demand for computation is growing day by day this aspect is crucial.

Efficient resource usage Sharing hardware resources along multiple systems allows the infrastructure to make a better use of them, reducing unused resources.

Security Dedicating individual applications to single virtual machines involves an easier to maintain, monitor and configure system. Having a more complete view of every single application isolated in its environment means an improvement in security.

Backups Another aspect is the simplicity of doing backups. In fact, snapshots of the disk and memory image can be done in an automated way.

System recovery The last point bring to another advantage: the disaster recovery and the business continuity. In a virtualized infrastructure backups, recovery plans and high redundancy maximizes the virtual machine uptime minimizing risks.

Scalability & Load balancing Hardware independence allows for flexible management of virtual machines, increasing their resources or duplicating them on the fly.

Testing & Deployment A virtual machine can be easily shared between developers thus having a common environment in which develop and than distribute software in a stable way.

Legacy support Virtualization is a good strategy to keep legacy system running. Some organizations retain dependencies on legacy platforms and the software that runs on them.

Migrations Another advantage is the virtual machine migration capability, when needed a guest environment can be migrated entirely from one place to another via network in a totally transparent way, keeping the services always up. Imagine move all your virtual infrastructure from one datacenter to another, maybe dislocated in different country far away. This can be done to cope with changes in load or for economic reasons, such as changes in costs.

However the process of transition to this technology is the main disadvantage. It can be expensive and time consuming. Every migration to new technologies brings with it challenges that need to be addressed. Existing software

may need to be modified and adapted to the new infrastructure. This is not always possible in every case, such as the case of the constraints imposed by software licenses.

1.3 A brief history of virtualization

The concept of virtualization is not recent and owes its origins to the era of mainframes back to the late 1960s and early 1970s, when IBM was developing time-sharing techniques for its mainframes. The IBM 360/67 with the CP/CMS system made use of virtualization. Each user had his own virtual machine and the space was partitioned into virtual disks. Virtualization of mainframes remained popular in the 70's and it seemed to disappear between the 80's and 90's. In 1990 Sun microsystem started the "Stealth" project. This project was born from the frustration of using C/C++ API's thinking there was a better way to develop and run applications. During the following years the project was renamed several times until 1995, when it was named Java. The applications written in Java are independent from the physical hardware and are executed (in form of Bytecode) on the Java Virtual Machine. In 1998, the company VMware Inc. was founded, which a year later introduced virtualization for x86 systems. In 2001 it released GSX Server and ESX Server. The former was capable of running within an operating system such as GNU/Linux (Hosted hypervisor), while the latter was capable of running directly on the hardware (bare-metal hypervisor). VMware is one of the leaders in the market of virtualization technologies. Today, service providers and datacenters make extensive use of virtualization technologies to abstract their physical resources and make their systems scalable and robust. During the evolution of the various technologies the primary purpose of virtualization has remained the same: to run multiple independent systems at the same time in a single computational environment.

1.4 Hypervisor

An *Hypervisor*, also known as *Virtual Machine Monitor* (VMM) is the central component of a virtualization based system. It manages and runs the guest machine virtualized environments in a transparently and efficiently way not to impact the performance. It carries out control activities above each system, allowing it to be used also as a monitor and debugger. There are two types of *hypervisor* families: bare-metal and hosted. A bare-metal or native *hypervisor* runs directly on the host machine hardware abstracting the guest machines from the physical resources. Xen and VMware ESX/ESXi are examples of native VMMs. A hosted *hypervisor*, instead, runs on a operating system just like normal applications do. It abstracts guest environments and resources from the host OS. For example QEMU [26] or VirtualBox [19] can be classified as hosted VMM. This classification is not always clear [17], for example the Linux KVM (Kernel-based Virtual Machine) module acts as a bare-metal hypervisor.

However, the VMM still competes with other processes in the host machine, as in a common hosted hypervisor environment.

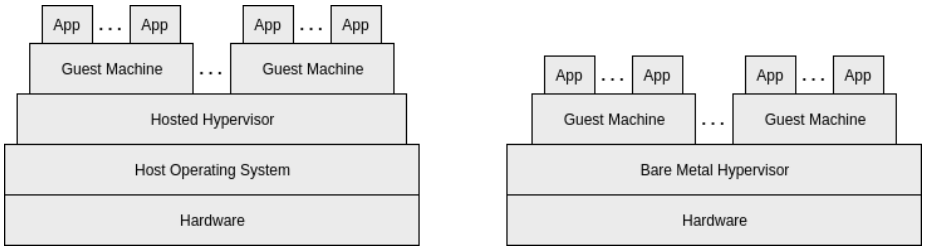


Figure 1.1: Hosted and bare-metal hypervisor architectures

1.5 Virtual machines

In general, a virtual machine is a software that creates and emulates the behavior of a physical machine. Specifically, there are two types of different virtual machines:

Process virtual machine Sometime called application virtual machine, runs inside the host userspace as a normal OS process. Its purpose is not to virtualize an isolated operating system but instead support the execution of an application in a platform-independent programming environment. An example is the Java Virtual Machine for the Java language or Wine (Wine Is Not an Emulator), which allows running Windows native applications in a GNU/Linux environment.

System virtual machine Often when mentioning virtual machines, we refer to this type of VMs, whose purpose is to virtualize an entire operating system isolated and independent from the host machine one. VirtualBox is one of the most common System virtual machine software.

A system virtual machine is not a monolith block and is divided in two sub components:

Host virtual machine It is the server component of the virtual machine and provides computing resources such as processing power, memory, disk and network I/O to a particular guest VM.

Guest virtual machine It is the operating system instance of the guest machine and represents the client component in the system VM architecture.

Together, with the VMM orchestration, Guest and Host VM make up a Virtual Machine.

1.6 Popek and Goldberg theorems

In 1974 Gerald J. Popek and Robert P. Goldberg introduced the conditions that a platform must meet in order to be properly virtualized [25]. Even today, these parameters remain valid guidelines followed when designing efficient virtualization architectures. This are the three fundamental properties that must be respected when an arbitrary program runs in a virtualized environment:

The equivalence An arbitrary program that runs in a virtualization environment behaves exactly as if it is on an equivalent physical machine.

The efficiency All non-privileged and innocuous instructions are executed directly by the hardware, without Hypervisor intervention.

The resource control The virtualization environment must have complete control of the resources.

Before enunciating the first theorem it is necessary to introduce some concepts

1. First, a *third generation computer* is a machine with a processor with at least two operating modes: Supervisor and User. In user mode only a subset of the available instructions is available while in supervisor mode the entire instruction set can be accessed.
2. Second, the classification of the instructions:
   - *Privileged instructions* are the ones not accessible in user mode that require the intervention of the VMM.
   - *Sensitive instructions*, instead, are those that attempt to change the configuration of system resources or those whose output depend on them.

Theorem 1 *For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.*

In other words, privileged instructions are captured and executed by the VMM while non-privileged instructions can be executed directly on the host in accordance with the principle of efficiency. This feature allows to realize a recursive virtualization, that is the capability to virtualize a copy of the VMM on itself, as specified in the next theorem:

Theorem 2 *A conventional third generation computer is recursively virtualizable if it is:*

1. *virtualizable, and*
2. *a VMM without any timing dependencies can be constructed for it.*

So, if a virtual machine can make a copy of the VMM, then it is recursively virtualizable. Theoretically, the limit of nested VMMs is defined only by the amount of memory available in the system. Nowadays, the concept of recursive virtualization is also called *Hierarchical virtualization*.

1.7 Different kinds of virtualization

Indeed, there exist many kinds of virtualization, which are briefly reviewed next.

1.7.1 Emulation

Although emulation and virtualization are slightly different, the notion of emulation is here recalled as it is the foundation to understand more advanced forms of virtualization. An emulator simulates completely a computer architecture making it possible to run software on machines with different ISAs. Guest machine instructions are therefore incomprehensible to the physical processor and must be translated on the fly by the emulator. This process takes time and adds a significant overhead that makes this mechanism less efficient than the others. However, this mechanism is particularly useful when running legacy systems written for outdated architectures. Typically, an emulator is divided into different modules that correspond to the subsystems of the emulated architecture. Generally, an emulator comprises the following modules:

- A CPU simulator that abstracts the desired architecture via software.
- The subsystem that manages memory.
- Various I/O (Input/Output) device emulators.

An example of an emulator is QEMU, which emulates the hardware of the host system by dynamically examining the executed code [26]. QEMU can emulate many hardware architectures, including x86, x86_64, RISC-V, ARM, SPARC, PowerPC, and MIPS.

1.7.2 Full-virtualization

Full Virtualization is the most used mechanism because it enables the achievement of very good performance with the highest level of portability. Portability is given by the fact that the operating system is minimally modified and therefore can run in the same way both in native and virtualized mode. Operations that are not dangerous are performed directly on the hardware very quickly, while those that may involve problems are trapped and dynamically transformed into secure operations through binary translation and stored in a cache to speed up any future accesses.

1.7.3 Para-virtualization

In this technology the hypervisor does not simulate hardware, but offers special APIs to the guest system for interaction with the underlying host. The hypervisor must make available to the hosted operating system a virtual interface used to access the resources:

- The guest operating system kernel must be modified in order to access the particular VMM interface.

- The structure of the VMM is simplified because it has no longer to worry about identifying the privileged operations of the guest machine.

Virtualized system applications remain unchanged, despite it is required to port the existing operating systems. This type of approach allows obtaining the minimum performance loss compared to the non-virtualized execution, since almost all instructions coming from the virtual machines are executed directly onto the processor, without the intervention of an hypervisor that places itself between the virtual machine and the physical resources. In particular, the two main Intel technologies Support for hardware-based virtualization can further increase efficiency. Intel and AMD have independently developed the x86 architecture virtualization extensions. They are not fully compatible with each other, but support roughly the same instructions. They both allow a virtual machine to run a host operating system without incurring major performance losses caused by software emulation. In particular, this extensions are called Intel Vanderpool (or Intel VT, IVT, VT-x) and AMD Pacifica.

1.7.4 Containers and OS-level virtualization

The operating system level virtualization is one of the cheapest and most efficient technique. Still, one of the most limited. Instead of having a fully virtualized operating system all the virtual environments are executed inside the same OS instance. Hence, only one kernel is loaded in memory. Such a kernel is shared among all the virtualized systems, which are called *containers*. This means less memory usage an more efficient guest processes execution, in fact they runs as in a normal operating system without an additional heavy virtualization layer. Sharing the kernel brings, however, some important drawbacks. Containers operating systems must use the same host kernel and the same host computer architecture. Isolation and management of virtualization environments is delegated to the host operating system and to the container engine. One of the most famous OS-level virtualization software is *OpenVZ* that provides an ad-hoc patched linux kernel [28]. Another important software is *Docker* that use containers to deliver software packages [21].

Chapter 2

The Xen hypervisor

2.1 Introduction

The **Xen Project** hypervisor is an open-source *type-1 or baremetal hypervisor*, which makes it possible to run many instances of an operating system or indeed different operating systems in parallel on a single machine (or host). The Xen Project hypervisor is the only type-1 hypervisor that is available as open source. It is used as the basis for a number of different commercial and open source applications, such as: server virtualization, Infrastructure as a Service (IaaS), desktop virtualization, security applications, embedded and hardware appliances [12].

Some of the Xen hypervisor's key features are:

- Small footprint
- Operating system agnostic
- Driver isolation
- Paravirtualization

The project is maintained as free and open-source software, subject to the *GNU General Public License* version 2.

2.2 History

Xen originated as a research project at the **University of Cambridge** led by *Ian Pratt*, a senior lecturer in the Computer Laboratory, and his PhD student *Keir Fraser*. The first public release of Xen was made in 2003, with v1.0 following in 2004. Soon after, Pratt and Fraser along with other Cambridge alumni including *Simon Crosby* and founding CEO *Nick Gault* created **XenSource Inc.** to turn Xen into a competitive enterprise product. In 2007, **Citrix Systems** completed its acquisition of XenSource. Finally, in 2013, it was announced

that the hypervisor developement was moved under the auspices of the **Linux Foundation** as a Collaborative Project, under the name "*Xen Project*". The members at the time of the announcement included: Amazon, AMD, Bromium, CA Technologies, Calxeda, Cisco, Citrix, Google, Intel, Oracle, Samsung, and Verizon [11].

2.3 Architecture

The Xen hypervisor runs directly on the hardware. It is the first piece of software running after the bootloader and is responsible for managing memory partitioning, timers, interrupts, and CPU scheduling of a number of virtual machines (called "*domains*" or "*guests*") running on it.

All domains must request Xen to provide them with any resources, and because the hypervisor itself has no knowledge of I/O capabilities such as networking and storage, they must install virtual-device drivers for the hardware devices they want to communicate with.

Xen supports multiple instances of operating systems with native support for most of them, including: Linux, BSD and Windows.

2.3.1 Domain 0

The Domain 0 runs on a modified Linux kernel. It is the unique virtual machine on the Xen hypervisor that has special rights to access physical I/O resources as well as interact with the other virtual machines (*Domain U*) running on the system. All Xen virtualization environments require Domain 0 to be running before any other virtual machines can be started, in large part because it has two drivers to support network and local disk requests from Domain U PV and HVM Guests [3]:

- The **Network Backend Driver** that communicates directly with the local networking hardware to process all virtual machines requests coming from the Domain U guests;
- The **Block Backend Driver** which communicates with the local storage disk to read and write data from the drive based upon Domain U requests.

2.3.2 Memory and CPU

Domain 0 and all the virtual machines share the same hardware resources.
Virtual machine's processes run directly on the hardware CPU, dom0 controls scheduling and intercept every interrupts, but it does not have the power to see or interfere with the processes. So CPU's are assigned to different Domain as needed, while keeping the scheduling fair and allowing dom0 (that runs in a more privileged CPU state) to change scheduling priority.
On the contrary RAM portions are dedicated to a single Domain (dom0 included).Every virtual machine has a fixed ammount of RAM that cannot change

while the machine is running. There actually is a technique called "Ballooning" that allows dom0 to change guest memory size "on the go" but it's out of the scope of this thesis.

2.3.3 Domain U

These are all virtual machines that do not have direct access to physical hardware on the host machine and are therefore called *Unprivileged*.

A distinction can be made between Domain U types based on the virtualization technology they use.

Domain U PV Guests

The operating system, usually modified Linux operating systems, Solaris, FreeBSD, and other UNIX operating systems is aware that it does not have direct access to the hardware and recognizes that other virtual machines are running on the same machine.

A Domain U PV Guest contains two drivers for network and disk access, **PV Network Driver** and **PV Block Driver**.

Domain U HVM Guests

The operating system, usually Windows or other non-editable kernels, is unaware that it is sharing processing time on the hardware and that other virtual machines are present.

A Domain U HVM Guest does not have the *PV drivers* located within the virtual machine; instead a special daemon is started for each HVM Guest in Domain 0, **qemu-dm**. It supports the Domain U HVM Guest for networking and disk access requests.

The Domain U HVM Guest must initialize as it would do on a typical machine so software is added to the Domain U HVM Guest, **Xen virtual firmware**, to simulate the BIOS an operating system would expect on startup.

Chapter 3

Tracing events

3.1 Introduction

It is important to understand the difference between simple event logging and tracing. The former is often used by system administrators to catch and resolve high-level issues (e.g. failed installation of programs or intrusion detection). It must be easy, so the logs must not be "noisy". On the other hand, tracing is consumed primarily by developers and logs low-level information (e.g. thrown exception). Since it handles lower level information it is necessarily "noisy", this means that reading the logs is not always intuitive or simple.

In operating system based computing environments a significant amount of a process' behaviour is defined by its interface with the operating system. This interface typically defines the process' environment such as the current directory, the input/output operations including operations of files, the execution of sub-processes and inter-process communication. Logging and interpreting the transactions between a process and the operating system it runs on, can provide data that can be used for a variety of purposes. Some of them are [27]:

Debugging A listing of a program's operating system calls allows the programmer to analyse its behaviour at a low but well defined level. Program errors can be explained in terms of the operating system calls that were (or were not) issued, and these can in turn point to the source of the error.

Profiling System calls can consume a significant amount of program run time, since the state of the machine must be saved and restored between calls. A listing of the system calls can provide hints on areas of a program that can be optimised to enhance a program's speed.

Program verification A log of a program's transactions with the operating system can be used to verify a program against its specifications or a run of a previous version. In addition, the log can be used to detect the use of non-portable functions, or programs that have been infected by viruses.

In other words, tracing is mainly used to understand what is happening and how a given system behaves meanwhile performing a certain operation. To do tracing you must always, for better or worse, be able to intercept some operations and make sure that a "trace" remains (from here the name) somewhere (e.g. file, buffer in memory). What operations are traced and how they actually do so define the tracing mechanism itself. But is tracing really useful? Computer systems, both at the hardware and software-levels, are becoming increasingly complex. In the case of Linux, used in a large range of applications, from small embedded devices to high-end servers, the size of the operating system kernels increases, libraries are added, and major software redesign is required to benefit from multi-core architectures, which are found everywhere. As a result, the software development industry and individual developers are facing problems which resolution requires to understand the interaction between applications and all components of an operating system [20].

3.2 Tracing tools

Before starting to investigate the tracing made directly by the operating system with *ftrace*, it may be interesting to briefly analyze some tools that allow a higher level of tracing. *strace* is a utility which allows you to trace the system calls that an application makes. When an application makes a system call, it is basically asking the kernel to do something (e.g. file access). Meanwhile *ftrace* is a tool used during kernel development and allows the developer to see what functions are being called within the kernel.

3.2.1 strace

strace is a diagnostic, debugging and instructional userspace utility for Linux. It is used to monitor and tamper with interactions between processes and the Linux kernel, which include system calls, signal deliveries, and changes of process state. System administrators, diagnosticians and trouble-shooters will find it invaluable for solving problems with programs for which the source is not readily available since they do not need to be recompiled in order to trace them. The operation of strace is made possible by the kernel feature known as *ptrace* [7].

Let us have this simple source code:

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4
5  int main(int argc, char** argv)
6  {
7      int testInteger;
8      pid_t pid = getpid();
9
10     printf("My PID is: %d\n", pid);
11     printf("Enter an integer: ");
12     scanf("%d", &testInteger);
```

```
13      printf("Number = %d\n", testInteger);
14
15      return 0;
16  }
```

Once done, it will first print its *pid*. At this point, opening another terminal and giving the command:

```
$ strace -p pid
```

And the output will be seems like:

```
strace: Process 1530 attached
read(0, "5\n", 1024)                    = 2
write(1, "Number = 5\n", 11)            = 11
lseek(0, -1, SEEK_CUR)                  = -1
exit_group(0)                           = ?
+++ exited with 0 +++
```

In this specific example, there are mainly two system calls. Before, the process reads an integer with `read()` (in this case: 5) and then it prints the same integer with ther `write()`; where `read()` and `write()` are the system calls.

3.2.2 ltrace

ltrace intercepts and records dynamic library calls which are called by an executed process and the signals received by that process. It can also intercept and print the system calls executed by the program, like *strace*. Keeping in mind the same source code used by strace, let us see what happens with *ltrace*. The procedure is very similar but this time the output will be:

```
printf("Number = %d\n", 2)              = 11
+++ exited (status 0) +++
```

Unlike strace, this time `printf()` does not mean a system call but a library call. Both tools, *strace* and *ltrace*, can be useful for developers to analyze high-level software and understand where problems arise. But they certainly do not say anything about what happens inside the kernel, for this reason step by step we will now analyze a much more powerful tool useful for the described purpose.

3.3 Tracing in Linux

Before starting the discussion of ftrace it is good to specify that it is not the only existing tool, but there are many others that are based more or less on the same principles (e.g. *perf events* [6], *eBPF* [2], *sysdig* [9], *LTTng* [4]). But *ftrace* is

what we are actually going to use, since it is the one that has (or rather will have) this phantom feature of allowing the **transverse tracing from guest to host** (Section 3.6).

Kernel debugging is a big challenge even for experienced kernel developers. For example, one difficulty is that if the system has latencies or synchronization issues (undetected race conditions), it is really hard to pinpoint where these issues are originated. Which subsystems are involved? In which conditions does the problem arise? When the system is running, there is not always a way to know the answer. *ftrace* is a debugger designed specifically to solve the issue and to ease the developer's debugging effort. Also, it is a great educational tool, not just to peek at what happens in the kernel, but also to help approach the source code.

The name comes from "function tracer", which is one of its features among many others. Each mode of tracing is simply called a *tracer*, and each one comes with many options to be tuned. Therefore *ftrace* is also very extensible because it is possible to write new tracers that can be added like a module.

3.3.1 Interfacing with ftrace

Whenever tracing, the events that need to be monitored are so frequent that an extremely lightweight mechanism is needed. *ftrace* offers this possibility because it is self-contained and entirely implemented in the kernel, requiring no user space tools whatsoever. As stated earlier, the *ftrace* output, which is produced from the kernel, is read from user space. How can we read it without a specialized program?

The solution is to use a dedicated special filesystem on which the kernel and the user can easily read/write: this creates a sort of shared memory between the user and the kernel. This practice is very common on Unix-like systems such as Linux; so common, in fact, that kernel process information is (almost) always accessed in this way. This is done through the `procfs` filesystem, which is found at `/proc`, as shown in Figure 3.1: every information about processes is stored here and it is fully accessible from user space. You can see that there is some generic information and also per-process information, with a folder for each current pid.

The alternative approach to get this information would be to use a special-purpose syscall, which is what BSD and MacOS do: the syscall will return a kernel structure with all the information that needs to be parsed. The approach used by Linux is more straightforward: the information is (mostly) in human-readable form, so you simply read the files in `/proc` and parse the results as strings. By doing so, no syscall is needed, except, of course, `open()` and `read()` to interact with the filesystem. On Linux, when commands such as `ps`, `top` or `pgrep` are invoked, they internally query the `procfs` filesystem. You could always do the same operation manually by doing something like `cat /proc/1337/info_that_you_need | grep specific_info`, but it would be tedious: this is why utilities like `ps` are convenient front-ends for the user.

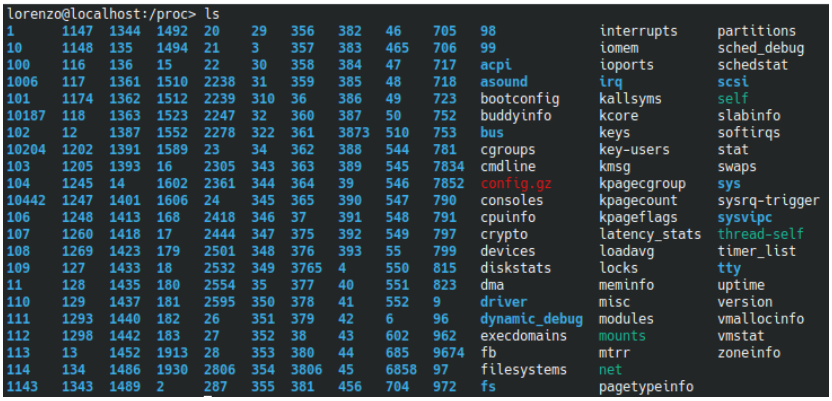


Figure 3.1: The procfs special filesystem

There are also other specialized filesystems, for example `sysfs`, which contains system information; but what iterests us is `debugfs`, which contains kernel debug information and allows interatcion with ftrace. This filesystem is mounted by executing `mount -t debugfs nodev /sys/kernel/debug/`: since there is not an actual device that is being mounted, we use "`nodev`" as target device; `/sys/kernel/debug/` is the target mount point. In Figure 3.2 you can see the trace folder located in this filesystem. To interact with ftrace you simply write in these files with `echo your_value > file`: by doing this you can toggle options and set parameters before/during the trace.

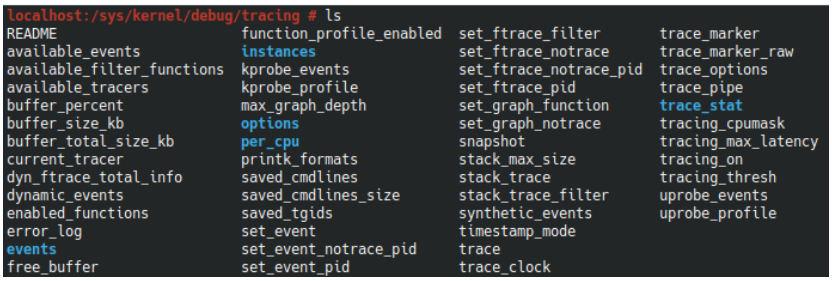


Figure 3.2: Tracing folder inside the debugfs special filesystem

The purpose of some of these files is not to set options. Rather, it is to list available options. For instance, in Figure 3.3, the available tracers are listed. These are essentially tracing modes: we activate one by doing `echo function > current_tracer`, which will immediately start to trace with the "function" tracer. We can then see the trace output by simply executing `cat trace`. Most of the other files are used for filtering what is being traced, which we will see in detail in the upcoming section.

Basically, we can interact with *ftrace* using the filesystem. Later, we also analyze other methods like `trace-cmd` 3.3.4 and `kernelshark`.

```
localhost:/sys/kernel/debug/tracing # cat available_tracers
blk function_graph wakeup_dl wakeup_rt wakeup function nop
```

Figure 3.3: Types of tracers on by distribution (OpenSUSE Tumbleweed)

```
$ cd /sys/kernel/debug/tracing
$ echo function > current_tracer
$ cat trace
```

3.3.2 Function tracing

Let us write a simple script that traces any input process.

```
1 #!/bin/bash
2 # traceprocess.sh
3 echo $$ > /sys/kernel/debug/tracing/set_ftrace_pid
4 echo function > /sys/kernel/debug/tracing/current_tracer
5 exec $1
```

`$$` is the variable that contains the pid of the script itself, and `$1` is the first argument of the script: in this case, the process to trace. The way it works is very simple:

1. Set this pid as the one that will be traced
2. Set the tracer to the function tracer
3. Execute the input program
4. The executed program will replace the process of the script itself, so the command passed as first argument to the script will be traced

Usually, we would see every kernel function that the input process calls, which is sometimes a big and uninformative output that needs filtering. The trace output can be found in the file `/sys/kernel/debug/tracing/trace`, or can be viewed as it gets written in `/sys/kernel/debug/tracing/trace_pipe`. The following is an output of `./traceprocess.sh ls`, which traces `ls`.

```
localhost:/sys/kernel/debug/tracing # head -n 20 trace
# tracer: function
#
# entries-in-buffer/entries-written: 204971/5796026   #P:4
#
#                      _------=> irqs-off
#                     / _-----=> need-resched
```

```
#                  | / _---=> hardirq/softirq
#                  || / _--=> preempt-depth
#                  ||| /      delay
# TASK-PID     CPU#  ||||   TIMESTAMP  FUNCTION
#    | |         |   ||||      |         |
    ls-12284   [000] ....  2755.250236: security_inode_permission <-link_path_walk.part.0
    ls-12284   [000] ....  2755.250236: open_last_lookups <-path_openat
    ls-12284   [000] ....  2755.250236: lookup_fast <-open_last_lookups
    ls-12284   [000] ....  2755.250236: __d_lookup_rcu <-lookup_fast
    ls-12284   [000] ....  2755.250236: step_into <-open_last_lookups
    ls-12284   [000] ....  2755.250236: __follow_mount_rcu <-step_into
    ls-12284   [000] ....  2755.250236: do_open <-path_openat
    ls-12284   [000] ....  2755.250236: complete_walk <-do_open
    ls-12284   [000] ....  2755.250236: unlazy_walk <-complete_walk
```

As expected, we only see function traced during the execution of `ls`. This information is not that useful by itself, but what is useful, instead, are the timestamps: with these, it is easy to detect latencies in the kernel. By using `kernelshark`the trace can be plotted to visualize the latencies; also, this may be used to estimate which actions cause most overhead. Another way of doing this just with ftrace is to use the `function_graph` tracer: it is similar to the `function` tracer, but it shows the entry and exit point of each function, creating a function call graph. Instead of timestamps it shows the duration of each function execution. The symbols `+`, `!` `#` are used whenever there is an execution time greater than 10, 100 and 1000 microseconds. As we know, scheduling and thread migration cause a lot of overhead, so we can try to use `function_graph` to see it.

```
localhost:/sys/kernel/debug/tracing # head -n 150 trace
# tracer: function_graph
#
# CPU   DURATION                  FUNCTION CALLS
#  |      |   |                     |   |   |   |
   2)   0.102 us    |    } /* text_poke_flush */
   2)               |    text_poke_loc_init() {
   2)   0.109 us    |      insn_init();
   2)               |      insn_get_length() {
   2)               |        insn_get_immediate.part.0() {
   2)               |          insn_get_displacement.part.0() {
   2)               |            insn_get_sib.part.0() {
   2)               |              insn_get_modrm.part.0() {
   2)               |                insn_get_opcode.part.0() {
   2)               |                  insn_get_prefixes.part.0() {
   2)   0.100 us    |                    inat_get_opcode_attribute();
   2)   0.101 us    |                    inat_get_opcode_attribute();
   2)   0.100 us    |                    inat_get_opcode_attribute();
   2)   0.704 us    |                  }
```

This is small piece of a trace using *function_graph*. Function duration is located at every leaf function and function exit point (}). Is important to keep always in mind that the buffer can be filled and some entries could be lost: this is very common if you trace everything without filtering. To mitigate this we can trace on a single CPU, instead of all 4. This approach has three advantages:

- The output has not function calls interleaved between the CPUs, which breaks the flow of function calls
- Since fewer entries are traced, the buffer is not filled and many will not be lost
- There is a performance gain: tracing every single function call generates significant overhead.

In general, it is better to narrow the filters as much as possible. For example, it would be good to trace only the function that we are interested in, and on one CPU only.

3.3.3 Event tracing

Function tracing is very useful and will come in handy to understand the code, but now we will focus on events. You may have noticed in Figure 3.2 that there is a directory called "events". It contains a folder for each *event subsystem* (as shown in Figure 3.4). Now let us focus on `kvm` events. Figure 3.5 shows its contents: there is a folder for each event, containing information about it and a switch to enable/disable it3.6.

```
localhost:/sys/kernel/debug/tracing/events # ls
alarmtimer        devfreq     gpu_scheduler  iomap        migrate         printk       scsi        tlb
amdgpu            devlink     hda            iommu        mmap            pwm          signal      udp
amdgpu_dm         dma_fence   hda_controller irq          module          qdisc        skb         v4l2
block             drm         hda_intel      irq_matrix   msr             random       smbus       vb2
bpf_test_run      enable      header_event   irq_vectors  napi            ras          snd_pcm     vmscan
bpf_trace         exceptions  header_page    jbd2         neigh           raw_syscalls sock        vsyscall
bridge            ext4        huge_memory    kmem         net             rcu          spi         wbt
btrfs             fib         hwmon          kvm          nmi             regmap       swiotlb     workqueue
cgroup            fib6        hyperv         kvmmmu       oom             regulator    sync_trace  writeback
clk               filelock    i2c            kyber        page_isolation  resctrl      syscalls    x86_fpu
compaction        filemap     initcall       libata       page_pool       rpm          task        xdp
context_tracking  fs_dax      intel_iommu    mce          pagemap         rseq         tcp         xen
cpuhp             ftrace      io_uring       mdio         percpu          rtc          thermal     xfs
cros_ec           gpio        iocost         mei          power           sched        timer       xhci-hcd
```

Figure 3.4: List of macro-event's section

```
localhost:/sys/kernel/debug/tracing/events/kvm # ls
enable                        kvm_eoi                       kvm_hv_synic_send_eoi       kvm_pi_irte_update
filter                        kvm_exit                      kvm_hv_synic_set_irq        kvm_pic_set_irq
kvm_ack_irq                   kvm_fast_mmio                 kvm_hv_synic_set_msr        kvm_pio
kvm_age_page                  kvm_fpu                       kvm_hv_timer_state          kvm_ple_window_update
kvm_apic                      kvm_halt_poll_ns              kvm_hypercall               kvm_pml_full
kvm_apic_accept_irq           kvm_hv_flush_tlb              kvm_inj_exception           kvm_pv_eoi
kvm_apic_ipi                  kvm_hv_flush_tlb_ex           kvm_inj_virq                kvm_pv_tlb_flush
kvm_apicv_update_request      kvm_hv_hypercall              kvm_invlpga                 kvm_pvclock_update
kvm_async_pf_completed        kvm_hv_notify_acked_sint      kvm_ioapic_delayed_eoi_inj  kvm_set_irq
kvm_async_pf_doublefault      kvm_hv_send_ipi               kvm_ioapic_set_irq          kvm_skinit
kvm_async_pf_not_present      kvm_hv_send_ipi_ex            kvm_mmio                    kvm_track_tsc
kvm_async_pf_ready            kvm_hv_stimer_callback        kvm_msi_set_irq             kvm_try_async_get_page
kvm_avic_ga_log               kvm_hv_stimer_cleanup         kvm_msr                     kvm_update_master_clock
kvm_avic_incomplete_ipi       kvm_hv_stimer_expiration      kvm_nested_intercepts       kvm_userspace_exit
kvm_avic_unaccelerated_access kvm_hv_stimer_set_config      kvm_nested_intr_vmexit      kvm_vcpu_wakeup
kvm_cpuid                     kvm_hv_stimer_set_count       kvm_nested_vmenter_failed   kvm_wait_lapic_expire
kvm_cr                        kvm_hv_stimer_start_one_shot  kvm_nested_vmexit           kvm_write_tsc_offset
kvm_emulate_insn              kvm_hv_stimer_start_periodic  kvm_nested_vmexit_inject    vcpu_match_mmio
kvm_enter_smm                 kvm_hv_syndbg_get_msr         kvm_nested_vmrun
kvm_entry                     kvm_hv_syndbg_set_msr         kvm_page_fault
```

Figure 3.5: Every event associated with kvm

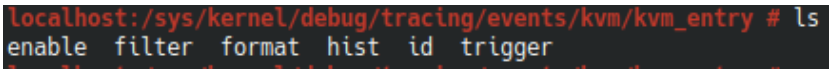


Figure 3.6: Control files for the `kvm_entry` event

Let us now see how event tracing is enabled and how to filter events. Events are not related to any tracer because tracers are used for dynamic tracing only. If we want to see just the events, then we must use the `nop` tracer (which does not trace anything), but we could also trace events while tracing functions by enabling any other tracer.

```
# enable kvm events
$ echo nop > /sys/kernel/debug/tracing/current_tracer
$ echo 1 > /sys/kernel/debug/tracing/events/kvm/enable
# enable just the kvm_entry events
$ echo nop > /sys/kernel/debug/tracing/current_tracer
$ echo 1 > /sys/kernel/debug/tracing/events/
↪  kvm/kvm_entry/enable
```

The "enable" file is located in every folder of the event directory tree. As you can see, the directory hierarchy is used to toggle single events, entire event subsystems, or all the existing events. Be aware that this filter does not stop the events from being written in the trace buffer, we are just ignoring them. "You have to recompile the whole kernel to disable specific events" can be paraphrased as "You have to recompile the whole kernel to prevent ftrace from writing specific events in its buffer, even when they are disabled from `debugfs`".

The following is a small piece of a trace of every `kvm entry` event:

```
# tracer: nop
#
# entries-in-buffer/entries-written: 360039/5452116   #P:4
#
#                                _-----=> irqs-off
#                               / _----=> need-resched
#                              | / _---=> hardirq/softirq
#                              || / _--=> preempt-depth
#                              ||| /     delay
#           TASK-PID     CPU#  ||||   TIMESTAMP  FUNCTION
#              | |         |   ||||      |         |
       CPU 0/KVM-10145   [002] d...   663.282125: kvm_entry: vcpu 0
       CPU 0/KVM-10145   [002] d...   663.282127: kvm_entry: vcpu 0
       CPU 0/KVM-10145   [002] d...   663.282129: kvm_entry: vcpu 0
       CPU 0/KVM-10145   [002] d...   663.282132: kvm_entry: vcpu 0
       CPU 0/KVM-10145   [002] d...   663.282135: kvm_entry: vcpu 0
       CPU 0/KVM-10145   [002] d...   663.282138: kvm_entry: vcpu 0
```

```
    CPU 0/KVM-10145   [002] d...   663.282142: kvm_entry: vcpu 0
# ... many more entries ...
```

In this trace the virtual cpu 0 (vCPU0) starts executing the virtual machine code on the host CPU2. So the virtual CPUs of the guests are treated exactly like all the other processes of the host. In other words, from the host's point of view, virtual CPUs are nothing more than processes, therefore they will be scheduled together with all the other processes on the machine. Obviously you can trace the `kvm_exit` event and its operation is similar to the previous one. In particular, after this event the host code starts running again on the cpu.

3.3.4 trace-cmd

After figuring out how to use ftrace using the filesystem directly, let us analyze how to do it more immediately. The `trace-cmd` command interacts with the *ftrace* tracer that is built inside the Linux kernel. It interfaces with the *ftrace* specific files found in the debugfs file system under the tracing directory. A command must be specified to tell `trace-cmd` what to do [10].

ftrace allows you to make and specify infinite options, trace-cmd cannot be outdone in fact there are many commands and each of them has many options. The two most important commands are `record` 3.7 and `report` 3.8. Let us see an example, this time by tracing the `kvm_exit` event.

```
# Interfacing through a command-line program
$ sudo trace-cmd record -e kvm:kvm_exit
$ sudo trace-cmd report
```

The `record` command starts capturing until we stop it with the usual `ctrl+c` (interrupting signal). A `trace.dat` file with all the traced information will then be created to the current folder (Figure 3.7 shows the output of `trace-cmd record`). After the recording phase, we can read the `trace.dat` file by the `report` command (Figure 3.8 shows the output of `trace-cmd report`).

```
lorenzo@localhost:/HDD/tesi/dat> sudo trace-cmd record -e kvm:kvm_exit
Hit Ctrl^C to stop recording
^CCPU0 data recorded at offset=0x7e0000
    8192 bytes in size
CPU1 data recorded at offset=0x7e2000
    8192 bytes in size
CPU2 data recorded at offset=0x7e4000
    4096 bytes in size
CPU3 data recorded at offset=0x7e5000
    4096 bytes in size
```

Figure 3.7: Output of the command *trace-cmd record*

```
lorenzo@localhost:/HDD/tesi/dat> sudo trace-cmd report | head -n 10
cpus=4
            CPU-10145 [003]  3729.120309: kvm_exit:             reason MSR_WRITE rip 0xffffffff9727b924 info 0 0
            CPU-10145 [003]  3729.120327: kvm_exit:             reason MSR_WRITE rip 0xffffffff9727b924 info 0 0
            CPU-10145 [003]  3729.120370: kvm_exit:             reason HLT rip 0xffffffff97c0ab6d info 0 0
            CPU-10146 [000]  3729.120450: kvm_exit:             reason MSR_WRITE rip 0xffffffff9727b924 info 0 0
            CPU-10146 [000]  3729.120465: kvm_exit:             reason MSR_WRITE rip 0xffffffff9727b924 info 0 0
            CPU-10146 [000]  3729.120494: kvm_exit:             reason MSR_WRITE rip 0xffffffff9727b924 info 0 0
            CPU-10146 [000]  3729.120501: kvm_exit:             reason HLT rip 0xffffffff97c0ab6d info 0 0
            CPU-10145 [003]  3729.120559: kvm_exit:             reason HLT rip 0xffffffff97c0ab6d info 0 0
            CPU-10145 [003]  3729.129280: kvm_exit:             reason MSR_WRITE rip 0xffffffff9727b924 info 0 0
```

Figure 3.8: Output of the command *trace-cmd report*

3.4 Tracing in KVM

Tracing while running a KVM hypervisor is not much different from tracing while running any other program. Every virtualized CPU of the guest machine corresponds to a process in the host.This process, despite being a special process with a couple of dedicated interrupts, is scheduled like any other by the host scheduler. So tracing in KVM is like tracing on a bare metal system, Section 3.3, but new events can be traced, in particular vm_entry and vm_exit when the guest system receives or gives back control of the cpu.

3.5 Tracing in Xen

Unlike in Linux, tracing in Xen is about scheduling domains with their respective virtual CPUs and in general all events that belong to the hypervisor. Therefore, it is not possible to have knowledge of the events occurring in the operating system kernel of a virtual machine except by tracing operations from within it. The tracing technique consists in dividing the events in buffers according to the number of CPUs present in the physical machine. Each of these buffers will keep in memory the events occurred only on the CPU to which they are related.

3.5.1 xentrace

xentrace [14] is the data collection tool developed by the **Xen Project** and provided along with the hypervisor. It can be executed only from the *control domain* and its main purpose is to retrieve and reassemble the tracing data from the various buffers into a single binary file.

Usage example We require an intial buffers cleanup (`-D`), a reading of CPUs from 0 to 7 (`-c 0-7`) and want to trace all events (`-e 0xffffffff`).

```
$ sudo xentrace -D -c 0-7 -e 0xffffffff trace_xen.bin
# Will continue to record until interrupted by CTRL+C
```

The output file will consist of a finite list of event records in binary form made as follows.

```
typedef struct {
    /* Header fields (32bit unsigned integer) */
    unsigned code:28;     // Identifier code
    unsigned n_extra:3;   // Extra arr. size
    unsigned in_tsc:1;    // Check bit for TSC
    /* Optional fields */
    unsigned long tsc;    // Timestamp counter
    unsigned extra[ 7 ];  // Extra informations
} xen_event;
```

The main features of which are:

- The presence of a "special event", called **TRACE_CPU_CHANGE**, indicating that the reading has passed from a buffer to another;
- A dummy domain, named *Default* (**Dom32768**), which is used to indicate that the given event cannot be bound back to any virtual machine in the hypervisor;
- A dummy domain, named *Idle* (**Dom32767**), which is used to indicate that the given event is related to some internal hypervisor operations and not to a virtual machine (There is one idle domain for each physical CPU).

3.5.2 xenalyze

xenalyze [13] is a CLI utility released in 2009 by **George Dunlap** as an external program and later included in the Xen Project. The xentrace analyzer offers the possibility to retrieve of information in different formats by mapping and rearranging events.

Dump mode Returns one line of text for each event read from the trace.

If you run the command:

```
$ xenalyze --dump-all trace_xen.bin
```

You get an output as follows (comments formatted as for `trace-cmd` are added to increase readability):

```
# Physical CPUs: 4
#
#               _----------=> pCPU0 ( buffer on idle dom )
#              / _----------=> pCPU1 ( reading from this buffer )
#             | / _---------=> pCPU2 ( buffer on default dom )
#             || / _-------=> pCPU3 ( buffer on idle dom )
#             ||| /           ...
#             ||||    _----=> DOMAIN
#             ||||   /  _--=> vCPU
```

```
#             ||||  |  /
#  TIMESTAMP  ||||  | |     EVENT            ADDITIONAL INFO
#      |      ||||  | |       |         |       |      |       |
  0.010334620 .x-. d5v1 sched_switch     prev d5v1 next d0v1
  0.010334845 .x-. d5v1 runstate_change  d5v1 running->runnable
  0.010334980 .x-. d?v? runstate_change  d0v1 runnable->running
  0.010376510 .x-. d0v1 vcpu_block  d0v1
  0.010377381 .x-. d0v1 csched2:schedule  cpu 1, rq# 1, ...
  0.010377546 .x-. d0v1 csched2:burn_credits  d0v1, credit = ...
  0.010377876 .x-. d0v1 csched2:runq_candidate  d32767v1 ...
  0.010378131 .x-. d0v1 csched2:update_load
# ... many more entries ...
```

Summary mode This output mode shows summary characteristics about each individual domain running during the tracing period. It reports, for example, how long a given domain ran and how long it was idle, or what physical CPUs it was primarily bound to.

```
$ xenalyze --summary trace_xen.bin

|-- Domain 0 --|
  Runstates:
    blocked: 5700 7.52s 3165824 {1075562|12707375|230355468}
   partial run: 29040 6.21s 513456 {134712|1887243|14283052}
   full run: 659 0.16s 567795 {560884|2474028|21593579}
   concurrency_hazard: 92181 12.73s 331312 {94330|804996|126539}
   full_contention: 1682 0.13s 187351 {32364|941060|7909034}
  Grant table ops:
   Done by:
   Done for:
  Populate-on-demand:
   Populated:
   Reclaim order:
   Reclaim contexts:
 -- v0 --
  Runstates:
    running: 12534 4.98s 954063 {237024|3239136|36007416}
   runnable: 11211 8.59s 1839312 {927052|7148439|72024376}
         wake: 8731 6.37s 1751990 {1044498|7211847|32301546}
      preempt: 10 0.00s 38239 {27144| 42912| 59076}
    blocked: 12389 78.54s 15214023 {3817083|57345697|372936012}
   cpu affinity: 1 220339879 {22033988339|2209878339|2339878339}
     [3]: 1 298778339 {220339878339|220339878339|220339878339}
 -- v1 --
  Runstates:
    running: 14708 86.23s 14070983 {3360155|195788661|351255873}
```

```
   runnable: 14686  5.72s 935515 {298548|5319216|110482002}
      preempt: 14686  5.72s 935515 {298548|5319216|110482002}
   cpu affinity: 1 220348050740 {2203480540|220340740|220350740}
     [1]: 1 22034805740 {220348050740|220348050740|220348050740}
 PV events:
   page_fault 45918
#  ...
   hypercall 683585
# ... many more entries ...
```

Scatterplot mode This output mode produces a chronological list showing the distribution of virtual CPUs to physical ones during the tracking period [1].

If you perform the command:

```
$ xenalyze --scatterplot-pcpu trace_xen.bin
```

You get an output as follows (comments formatted as for `trace-cmd` are added to increase readability):

```
#  vCPU               pCPU
#    |     TIMESTAMP    |
#    |         |        |
    0v11  0.018407919  11
    0v0   0.018413500   0
    0v1   0.024310861   1
    0v2   0.024325698   2
    0v3   0.024603944   3
# ... many more entries ...
```

The output can be transformed into a graph using an external tool such as "*gnuplot*".

3.5.3 xentrace_format

xentrace_format [15] is a CLI tool released by Mark Williamson in 2004, integrated in the Xen repository, it is a much simpler program than xenalyze as it just performs a *one-to-one translation* of events from their binary form to a more human readeable form. To use it, a text file that structures the output format is required (comments have been added to increase readability).

```
#                                HUMAN-READEABLE FORM
#                 ------------------------|-------------------------
#               /                                                   \
#  EVENT CODE  |    pCPU     TSC    RELATIVE TSC   EVENT NAME    EXTRA  |
#       |      |      |       |           |            |           |    |
   0x0001f001   CPU%(cpu)d %(tsc)d (+%(reltsc)8d) lost_records 0x%(1)08x
```

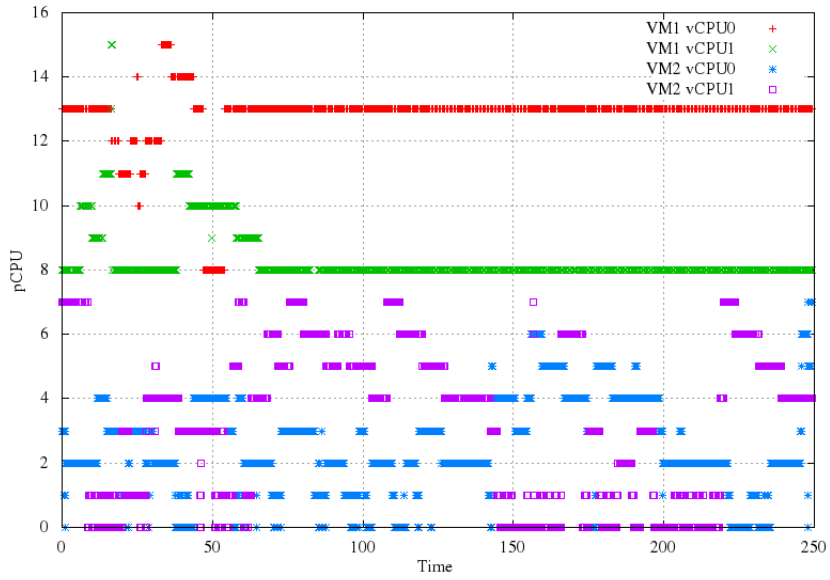


Figure 3.9: vCPU on pCPU execution of 2 VMs over time

```
   0x0001f002   CPU%(cpu)d %(tsc)d (+%(reltsc)8d) wrap_buffer  0x%(1)08x
   0x0001f003   CPU%(cpu)d %(tsc)d (+%(reltsc)8d) cpu_change   0x%(1)08x
# ... many more entries ...
```

Usage example

```
$ xentrace_format formats.txt < trace_xen.bin

CPU0  50475889362429  (+ ...)  csched2:schedule [ ... ]
CPU0  50475889363140  (+ ...)  switch_infprev [ ... ]
CPU0  50475889363455  (+ ...)  switch_infnext [ ... ]
CPU0  50475889363704  (+ ...)  __enter_scheduler [ ... ]
CPU0  50475889364070  (+ ...)  running_to_runnable [ ... ]
CPU6  50475889364427  (+ ...)  runnable_to_running [ ... ]
CPU6  50475889490340  (+ ...)  do_block [ ... ]
CPU6  50475889492062  (+ ...)  csched2:ratelimit [ ... ]
CPU6  50475889492443  (+ ...)  csched2:credit burn [ ... ]
CPU6  50475889492938  (+ ...)  csched2:schedule [ ... ]
CPU6  50475889493694  (+ ...)  csched2:update_load
CPU6  50475889494060  (+ ...)  csched2:updt_runq_load [ ... ]
CPU6  50475889494630  (+ ...)  switch_infprev [ ... ]
CPU6  50475889494975  (+ ...)  switch_infnext [ ... ]
CPU6  50475889495230  (+ ...)  __enter_scheduler [ ... ]
CPU6  50475889495821  (+ ...)  running_to_blocked [ ... ]
```

```
CPU2  50475889496157  (+ ...)  runnable_to_running [ ... ]
CPU2  50475889497954  (+ ...)  csched2:update_load
CPU2  50475889498236  (+ ...)  csched2:updt_runq_load [ ... ]
CPU2  50475889498545  (+ ...)  csched2:updt_vcpu_load [ ... ]
CPU2  50475892232310  (+ ...)  domain_wake [ ... ]
CPU2  50475892232781  (+ ...)  blocked_to_runnable [ ... ]
# ... many more entries ...
```

3.5.4 xentrace_parser

xentrace_parser is a C library for parsing Xen traces developed specifically for this project [16]. The analysis made by the tool allows deriving a list of records each independent of the previous/next ones, so that each record has all the information needed to be read on its own. This is made by performing three operations for each event read from the trace file:

1. It checks whether the event indicates a switch from a buffer to another (**TRC_TRACE_CPU_CHANGE** event).
2. It checks whether the event reports a change in the running domain/vCPU (**TRC_SCHED_MIN** events family);
3. Finally, for those events without any value of the TSC, such as value is set equal to the last TSC value read from previous events.

The type of the "*rec*" field is the same as that shown in section 3.5.1.

```
typedef struct {
    unsigned cpu:16;  // Host CPU value
    union {
        /* Merged as a single entity */
        unsigned u32;
        /* Values in detail */
        struct { unsigned vcpu:16, id:16 };
    } dom;          // Domain value
    xt_record rec;  // Record data
} xt_event;
```

After that, using stdlib's **QuickSort** algorithm, sorts all events in chronological order based on their "*Time Stamp Counter*".

```
int qsort_cmpr(const void *curr, const void *next)
{
    xt_event *curr_event = (xt_event *) curr,
            *next_event  = (xt_event *) next;

    xt_record curr_record = curr_event->rec,
              next_record  = next_event->rec;

    unsigned curr_tsc = curr_record.tsc,
```

```
10              next_tsc  = next_record.tsc;
11
12      /* Return values could be -1, 0, 1 */
13      return (curr_tsc > next_tsc) - (curr_tsc < next_tsc);
14  }
```

Usage example

Below is an example of library usage:

- At **line 8**, the parser instance is created on a hypothetical xentrace file, the path can be either absolute or relative.
- At **line 11**, the generation of the event list is done starting from the previously indicated trace file, the function returns the number of the events read.
- At **line 18**, a `while` loop retrieves each record in the list, until it finds the value `NULL` indicating the end of the list.
- At **line 46**, the number of *physical* CPUs read from the trace file is retrieved.
- At **line 54**, the memory allocated for the parser is freed.

```
1   #include <stdio.h>
2   #include "xentrace-parser.h"
3   #include "xentrace-event.h"
4
5   int main(int argc, char **argv)
6   {
7       // Init parser
8       xentrace_parser parser = xtp_init("trace_xen.bin");
9
10      // Analyze trace file (Returns readed events count)
11      unsigned n_events = xtp_execute(parser);
12
13      // Print events
14      xt_event *event;
15      while ( (event = xtp_next_event(parser)) )
16      {
17          // Print header
18          xt_domain dom = event->dom;
19          printf("[pCPU#%u][vCPU#%u][DOM#%u] ",
20                      event->cpu, dom.vcpu, dom.id);
21
22          // Print record
23          xt_record rec = event->rec;
24          printf("{HDR: {EVENT: 0x%08x, N_EXTRA: %u, IN_TSC: %u}",
25                      rec.id, rec.n_extra, rec.in_tsc);
26
27          // Print TSC
28          printf(", TSC: %lu", rec.tsc);
29
```

```
30          // Print extras (0 <= n_extra < 7)
31          if (rec.n_extra) {
32              printf(", ");
33
34              for (int i = 0; i < rec.n_extra; ++i) {
35                  printf("D%d: 0x%08x", i + 1, rec.extra[i]);
36
37                  if (i + 1 < rec.n_extra)
38                      printf(", ");
39              }
40          }
41
42          printf("}\n");
43      }
44
45      // Get pCPUs count
46      unsigned n_pcpus = xtp_cpus_count(parser);
47
48      // Print num of events and num of pCPUs
49      printf("\n\n");
50      printf("NUM# events: %u\n", n_events);
51      printf("NUM# pCPUs:  %u\n", n_pcpus);
52
53      // Free parser instance
54      xtp_free(parser);
55
56      return 0;
57  }
```

3.6 Tracing in guest

Tracing inside a VM is not that different from tracing inside a bare metal system if the hypervisor is KVM. The virtualized kernel doesn't know it's running inside a virtualized environment so it acts exactly as in Section 3.3. Every traced event is collected in the *trace* file 3.3.1 and can be analyzed. While it's actually possible to gather tracing information from host and guests virtual machines simultaneously, thanks to the *trace-cmd agent*, there was no need to. So we decided to use ftrace to trace separately the host and the guest systems as a normal tracing session. We used *mmetest* [5], a program that takes care of setting up the tracing environment 3.3.1, running the chosen benchmark and collecting the results.

- First *mmtest* sets the tracing environment in the host and in the guest, enabling the tracing events specified by the user

```
#!/bin/bash

if [ ! -e /sys/kernel/debug/tracing/ ]; then
        mount -t debugfs none /sys/kernel/debug 2>&1 || exit -1
fi
```

```
for OPTION in $MONITOR_FTRACE_OPTIONS; do
        if [ -e /sys/kernel/debug/tracing/options/$OPTION ]; then
                echo 1 > /sys/kernel/debug/tracing/options/$OPTION
        fi
done

for EVENT in $MONITOR_FTRACE_EVENTS; do
        SUBSYSTEM=`echo \$EVENT | awk -F / '{print $1}'`
        if [ "$SUBSYSTEM" = "probe\_func\_return" ]; then
                EVENT=`echo $EVENT | awk -F / '{print $2}'`
                perf probe "$EVENT%return"
                EVENT="probe/$EVENT"
        fi
        echo 1 > /sys/kernel/debug/tracing/events/$EVENT/enable
done
```

The events that the user wants to trace are specified in the **MONITOR_FTRACE_OPTION** environment variable

- Then it waits for ftrace to log events on the file *trace_pipe*

```
exec cat /sys/kernel/debug/tracing/trace_pipe
```

- *mmtest* starts all the benchmarks requested by the user

```
for TEST in $MMTESTS; do
        export CURRENT_TEST=$TEST
        # Configure transparent hugepage support as configured
        reset_transhuge

        # Run installation-only steps
        echo Installing test $TEST
        export INSTALL_ONLY=yes
        ./bin/run-single-test.sh $TEST
        if [ $? -ne 0 ]; then
             die "Installation step failed for $TEST"
        fi
              unset INSTALL_ONLY

        # Run the test
        echo Starting test $TEST
```

- When all the tests are done a *signal* is sent to the *cat* process that was reading the output of ftrace. Then the logs from ftrace are compressed and sent to the host if tracing is running on guest system.

Chapter 4

Tracing overhead in virtualized environment

4.1 Experimental setup

First thing to be done is to choose what aspect of virtualization we want to analyze. In our case we chose to focus on scheduling and cpu workload, because the introduced overhead is probably going to be really small and the CPU is the component less sensitive to external disturbances.For example if we focused on input/output to memory instead we could have had a process writing to memory while the benchmarks where running (graphical interface, network manager, etc ..). However, not every benchmark is suitable for testing the overhead, tracing has a small impact on performance so the tests needs to be as stable as possible. The same benckmark was run several times without tracing, if the differences between runs where smaller then the standard deviation of the single run the test was deemed stable. We settled on 7 types of micro-benchmarks:

- **cyclictest-none**: measures the latency of response to a stimulus, for example responding to an asyncronous call
- **cyclictest-hackbench**: same as above but the latency is measured while the system is under load, in this case the hackbench benchmark is used as workload
- **hackbench-process-pipes**: Hackbench is a benchmark and a stress test for the kernel scheduler, it also stresses parts of the memory subsystem through repeated setup and teardown of threads and pipes.
- **schbench**: schbench is a latency measurement benchmark that provide latency distribution statistics for scheduler wakeups. The worker threads wait for messages to come in which include a timestamp of when they were queued and records how long it took to receive the message.

- **sysbench-cpu**: calculates primes up to a certain value using varying numbers of thread.
- **sysbench-thread**: each worker thread will be allocated a mutex and will, for each execution, loop a number of times in which it takes the lock, yields and then, when it is scheduled again for execution, unlock. [8]
- **unixbench**: a complex suite benchmark that provides a basic indicator of the performance of a Unix-like operating system.
- **unixbench-io**: from the same family of unixbench but focused on I/O operation (read and write) on buffer or disk

The machine where the benchmark were run had 6 CPUs and 16GB of RAM, so we tested on 4 different configuration:

- Xen hypervisor, 1 VM with 4 CPU and 8GB of memory
- Xen hypervisor, 3 VM with 2 CPU and 4GB of memory each vm
- KVM hypervisor, 1 VM with 4 CPU and 8GB of memory
- KVM hypervisor, 3 VM with 2 CPU and 4GB of memory each vm

4.2 Cost of tracing with KVM

To help the reader understand the following graphs is better if every label is explained first.
BASELINE rappresents the the average of the 3 baseline runs,the ones without tracing. **FTRACE_HOST** is the run with ftrace enabled, and the following events traced: vm entry, vm exit, all scheduler events and irq events (*interrupts*). **FTRACEALL_HOST** rappresent the run with every possible event traced, **FTRACE_GUEST** is ftrace running only in the guest machine with the same events of FTRACE_HOST enabled and at last **FTRACE_BOTH** is the run with ftrace enabled on guest and host machine with the same events of FTRACE_HOST.

4.2.1 cyclictests

Cyclictest is a latency focused test, its results are the time elapsed from when an asyncronous call is made from a process to the moment the called process answersr.The graphs shows how well the benchamrks performed in comparison to the baseline.

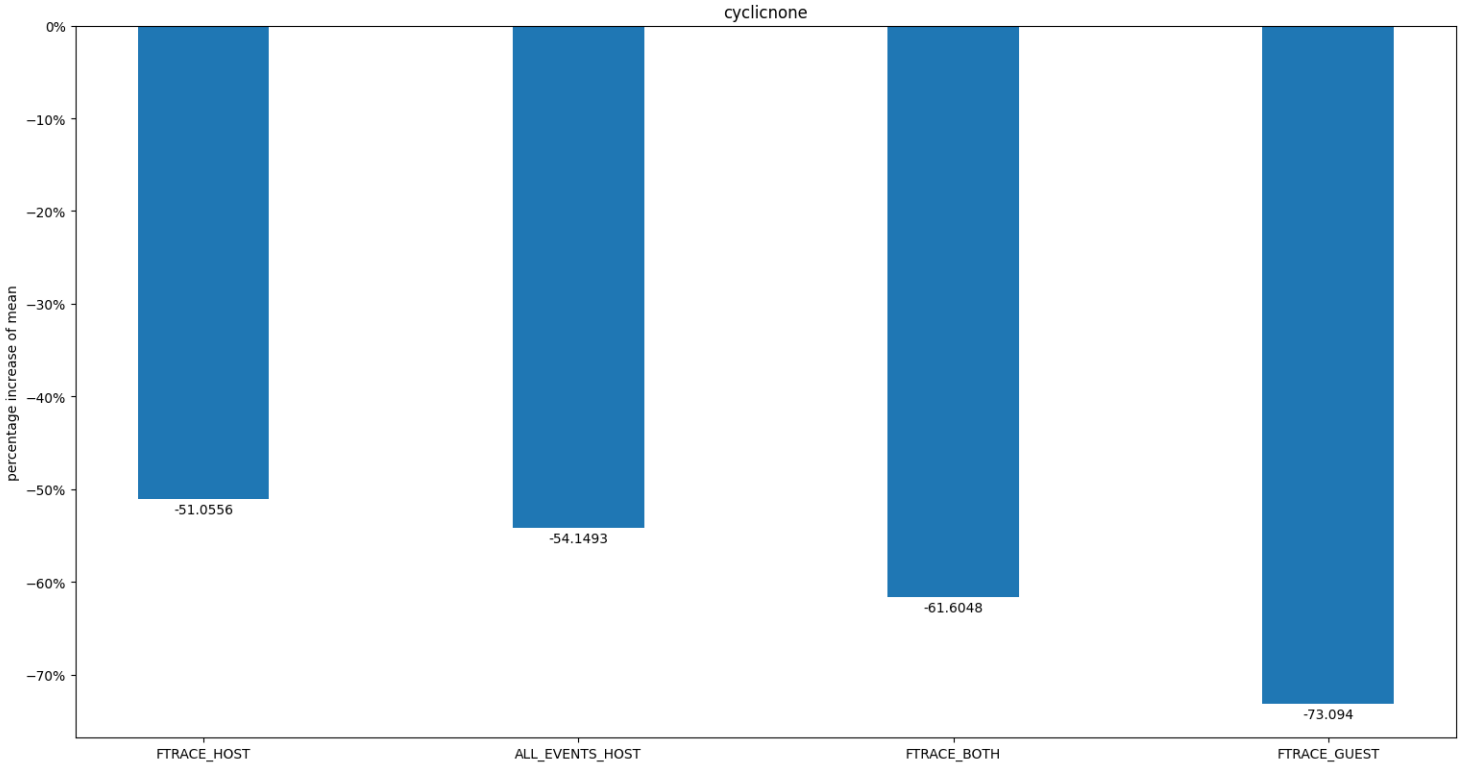


Figure 4.1: Mean comparison W.R.T. baseline run for cyclictest none

At a first analysis the results seem wrong (Figure 4.1), the benchmark runs perform better than the baseline run, but this is caused by the cpu frequency scaling. In fact in modern machines some monitors (both hardware and software) keep track of the workload and change the cpu frequency accordingly, scaling it down to reduce power consumption if the cpu has less work to do. In case of the cyclictestnone benchmark the cpu is completly unloaded during the baseline run, so the governor implemented in the CPU scales the frequency down slowing the machine, when tracing is enabled the cpu has some work to do so the governor reduces the frequency less and less as more tracing options are enabled.
This phenomenon is not present while running other benchmarks, like cyclictest-hackbench, because they combine the testing process and an intensive workload.In case of cyclictest-hackbench more processes are running on the machine, the cyclictest one and the hackbench stress test, so the CPU is never unloaded and the governor never scales down the frequency.

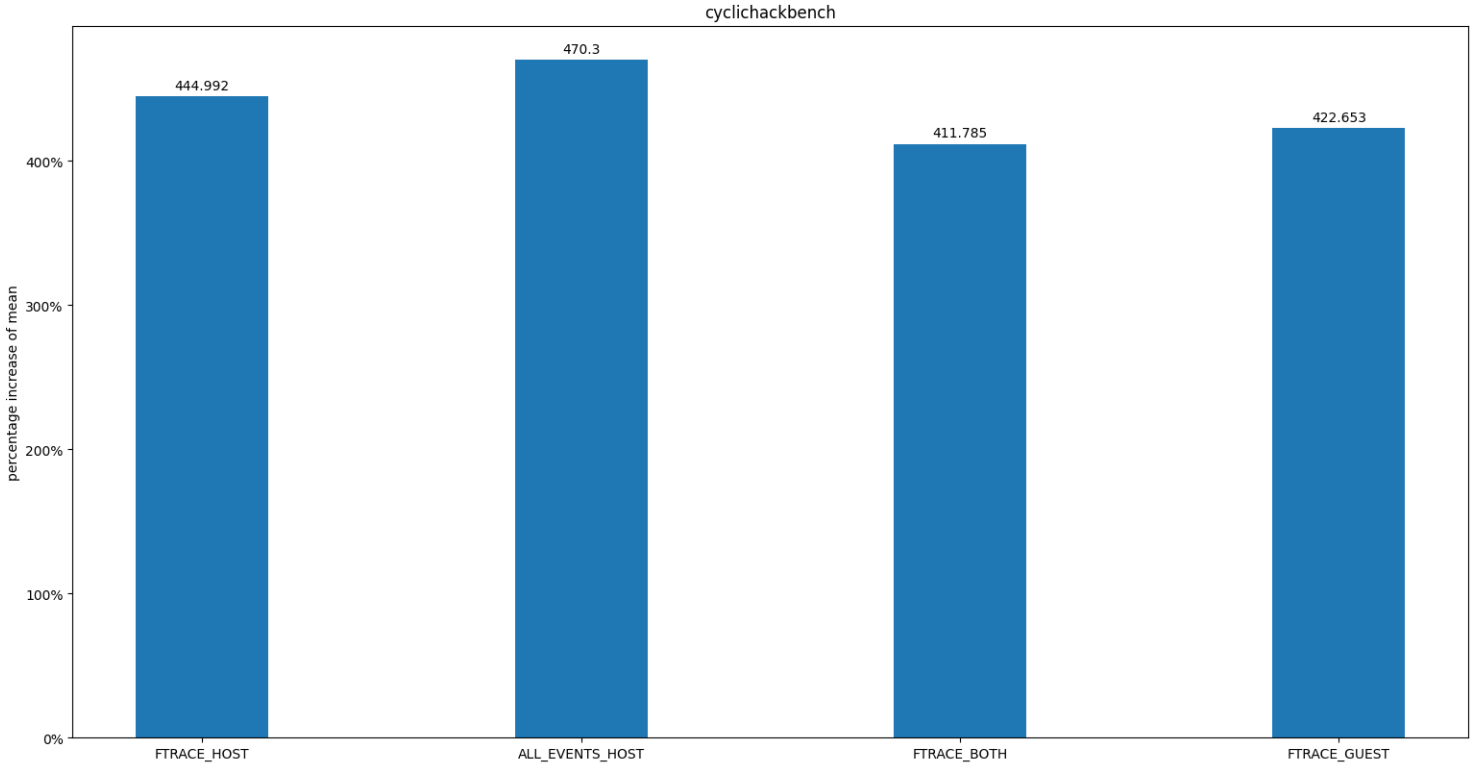
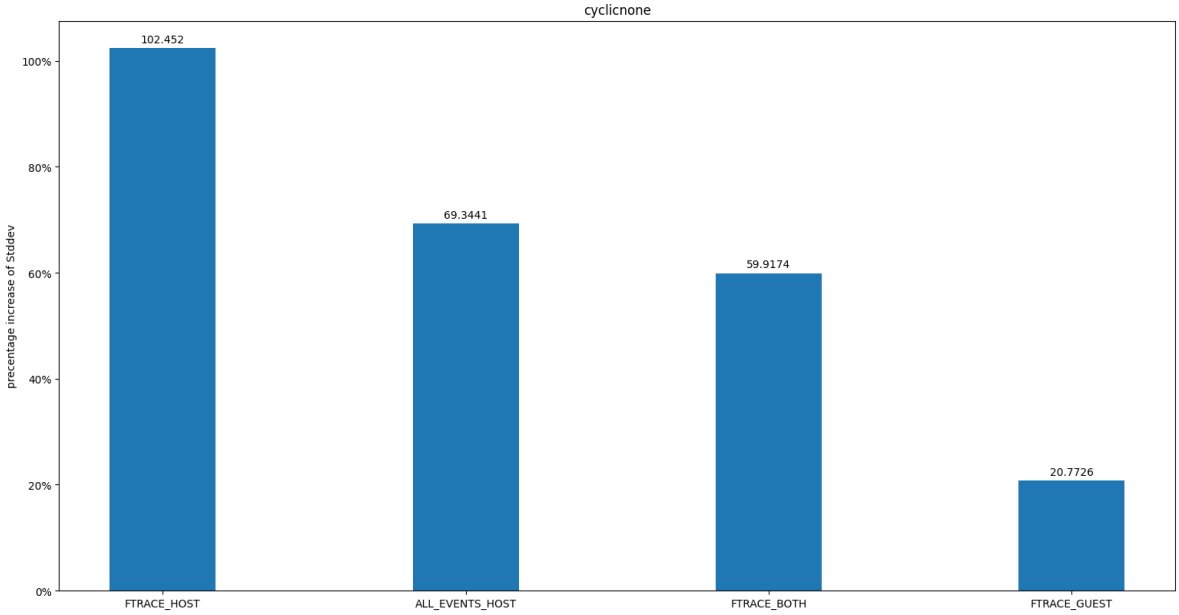
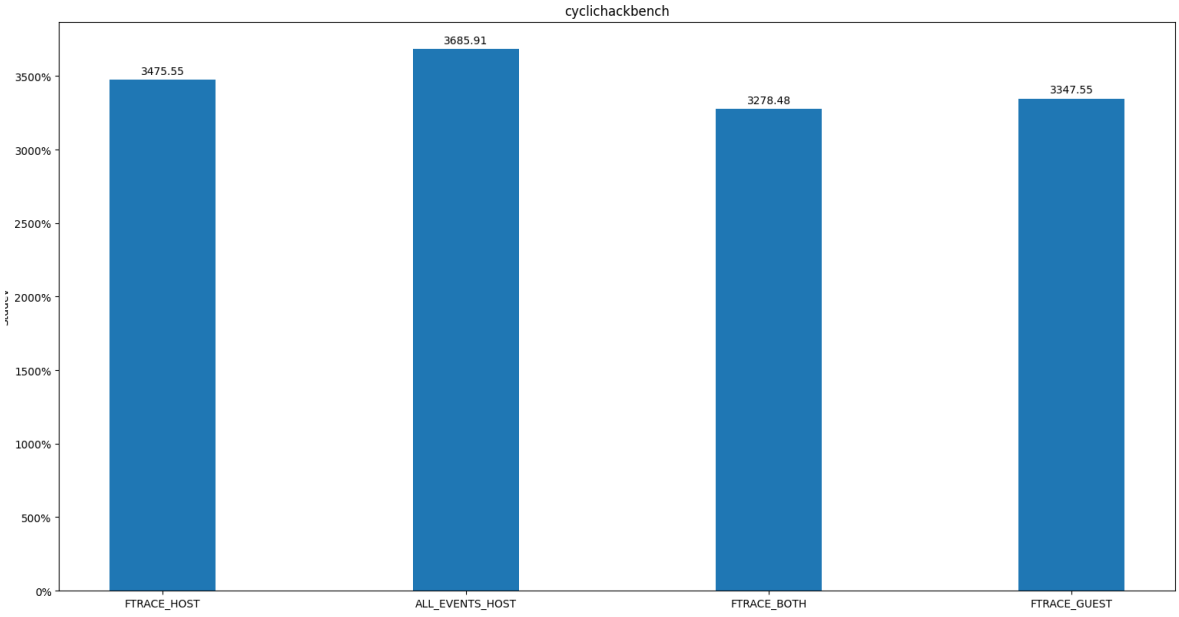


Figure 4.2: Mean comparison W.R.T. baseline for benchmark cyclictest hackbench

Looking at those results it seems likely that the hoverhead is directly related to the number of tracing options enabled, and this is generally the case. If more events are enabled the system will be less responsive.
Another interesting phenomenon that emerges from this data is how tracing increases fluctuation.The elapsed time from call to answer varies a lot when tracing is enabled, this can be seen studying the difference between standard deviation in the runs,Figure 4.3, but there is a benchmark specifically made for this,schedbench, analyzed in Section 4.2.3



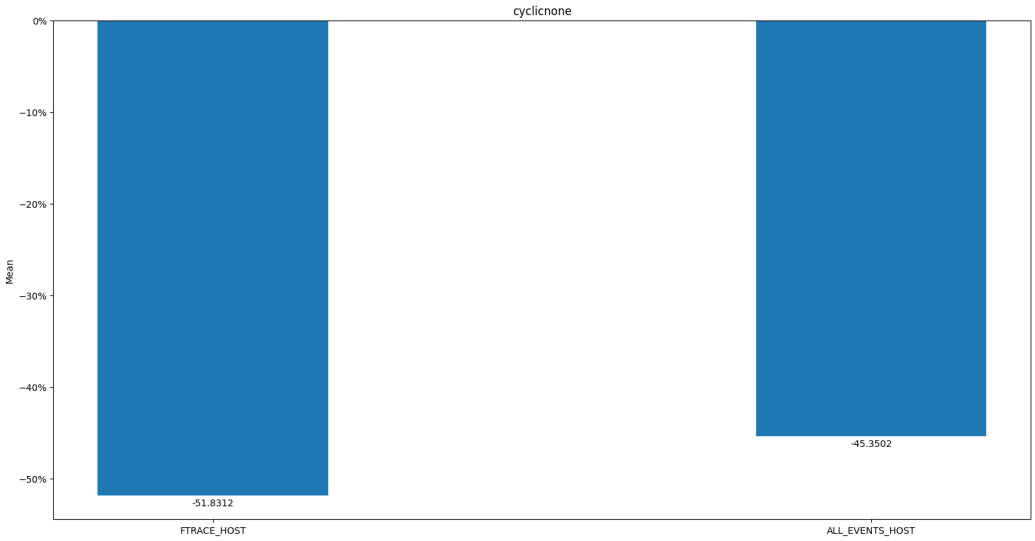
(a) standard deviation increase in cyclicnone



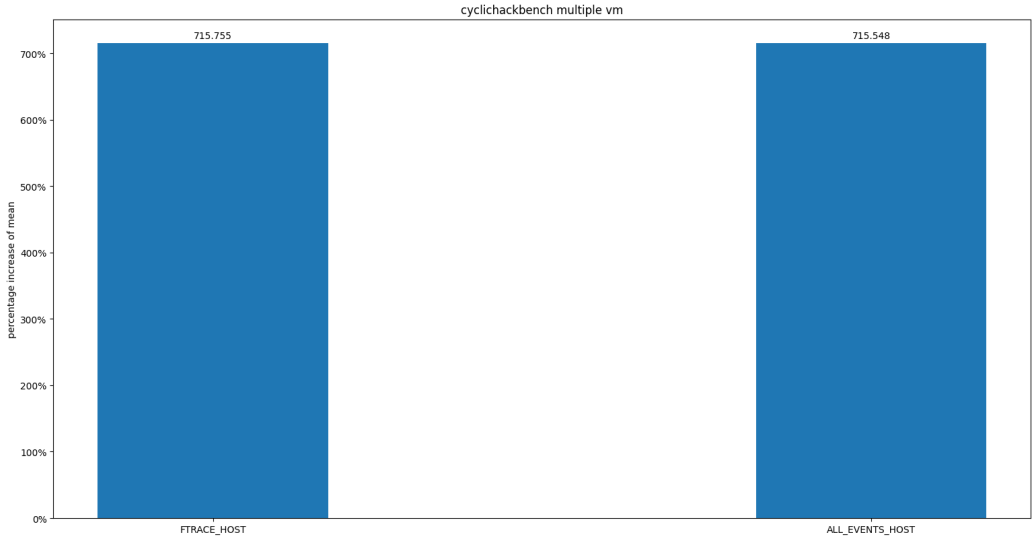
(b) standard deviation increase in cyclichackbench

Figure 4.3: stddev comparison for cyclictest's benchmarks

the same results can be seen in the configuration with multiple VMs (Figure 4.4), where an even stronger phenomenon of fluctuation can be observed (Figure 4.5)

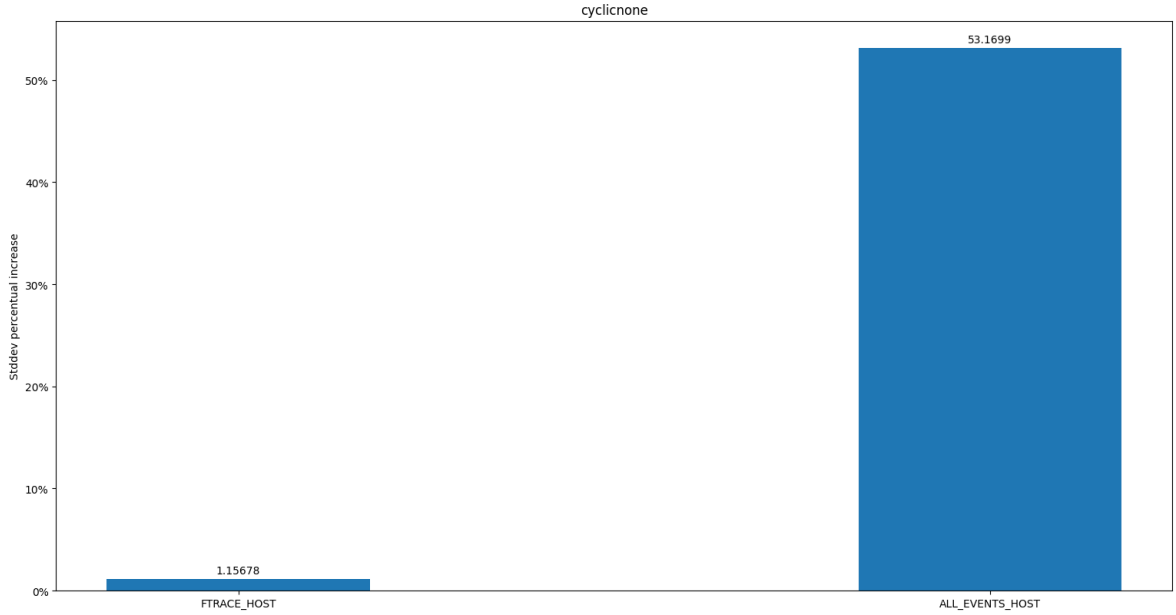


(a) performance decrease in cyclicnone on multiple VMs

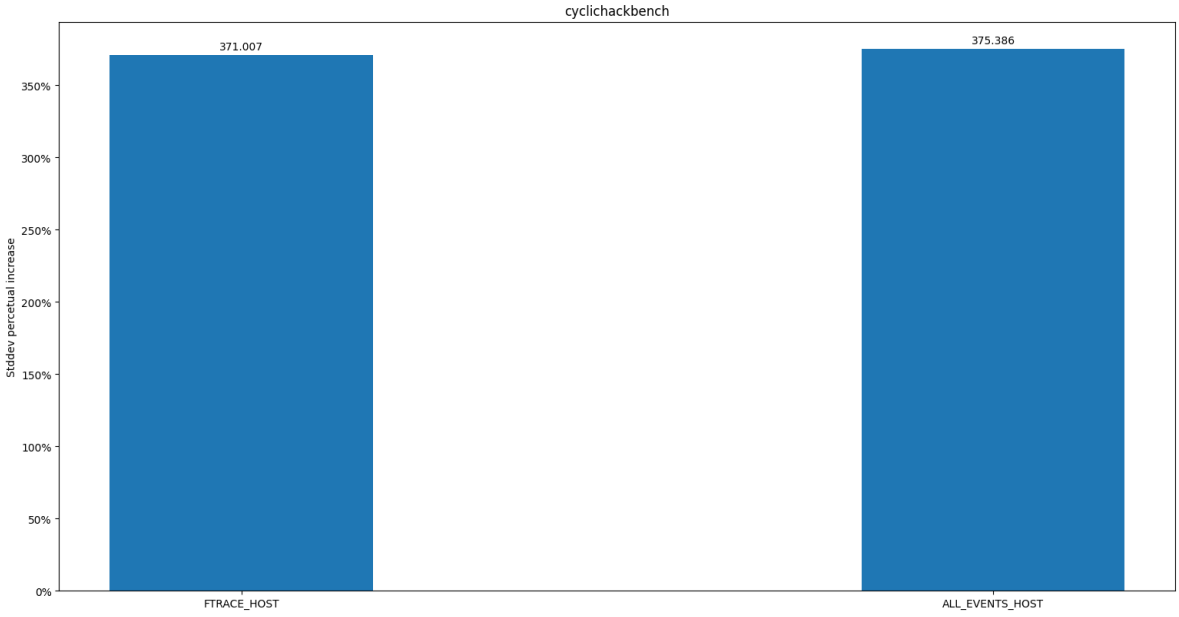


(b) performance decrease in cyclichackbench on multiple VMs

Figure 4.4: multiple vm performance comparison in kvm



(a) standard deviation increase in cyclicnone on multiple VMs



(b) standard deviation increase in cyclichackbench on multiple VMs

Figure 4.5: multiple vm standard deviation comparison in kvm

4.2.2 hackbench-process-pipes

Hackbench-process-pipes times how long it takes for pairs of process to send data back and forth using pipes, so it solely relies on CPU and a bit of RAM (data excanged is small). So it's the perfect candidate to analyze while looking for overhead in the CPU.In fact Figure4.6 shows that tracing has a negative impact on performance, but as we can see the performance drop is small.

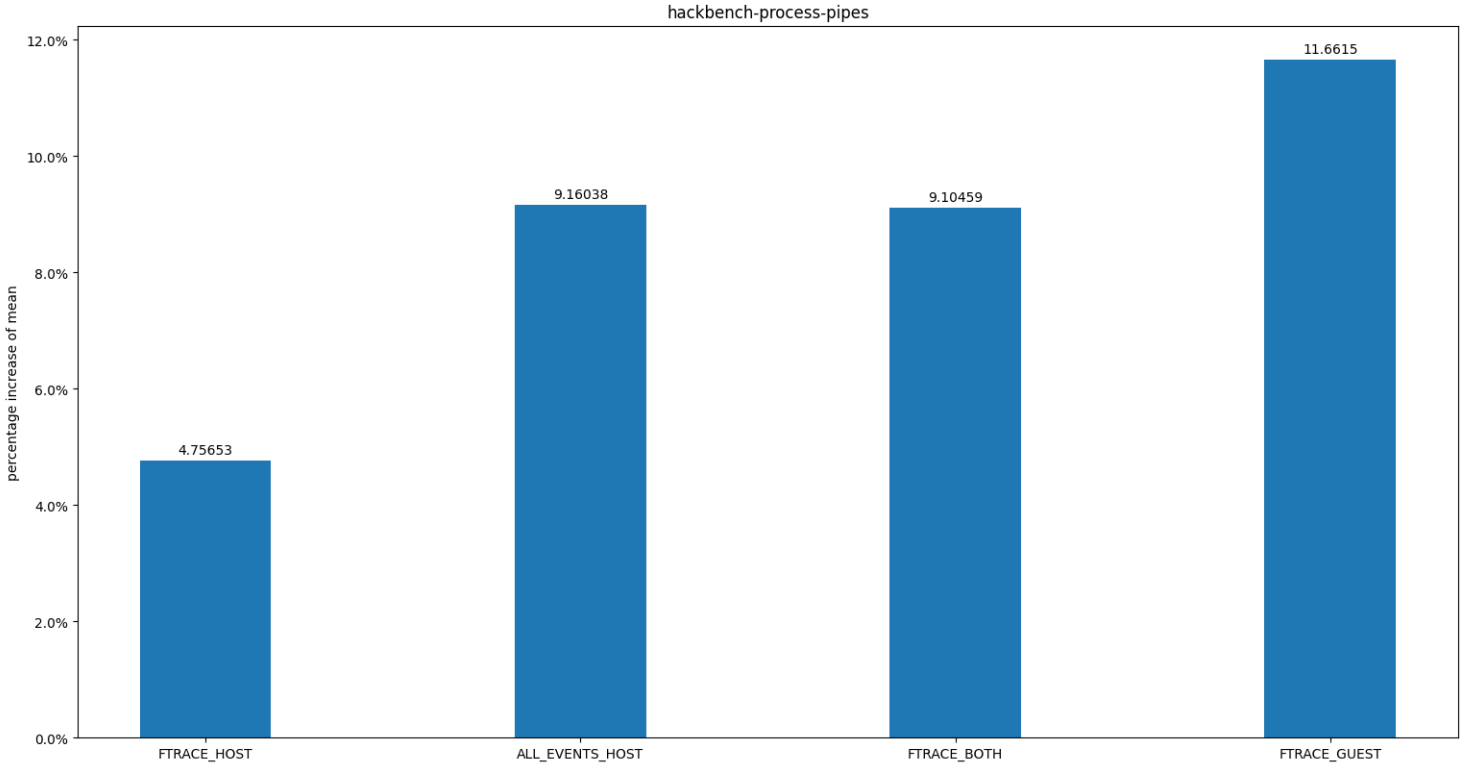


Figure 4.6: hackbench-process-pipes benchmark's results performance W.R.T baseline

Comparing the results from a single VM and the ones from multiple VMs (Figure 4.7) an interesting phenomenon emerges: in the runs with 3 VMs there is more overhead. It's expected that the 3 VMs shows worst performance but the difference in performance between benchmark and baseline should not change, why does it happens? This happens because in the case with multiple VMs every CPU in the system was busy, while in the case of a single VM at least 2 CPUs where unloaded.In Section 4.2.4 this behaviour will be better explained with a more relevant example.

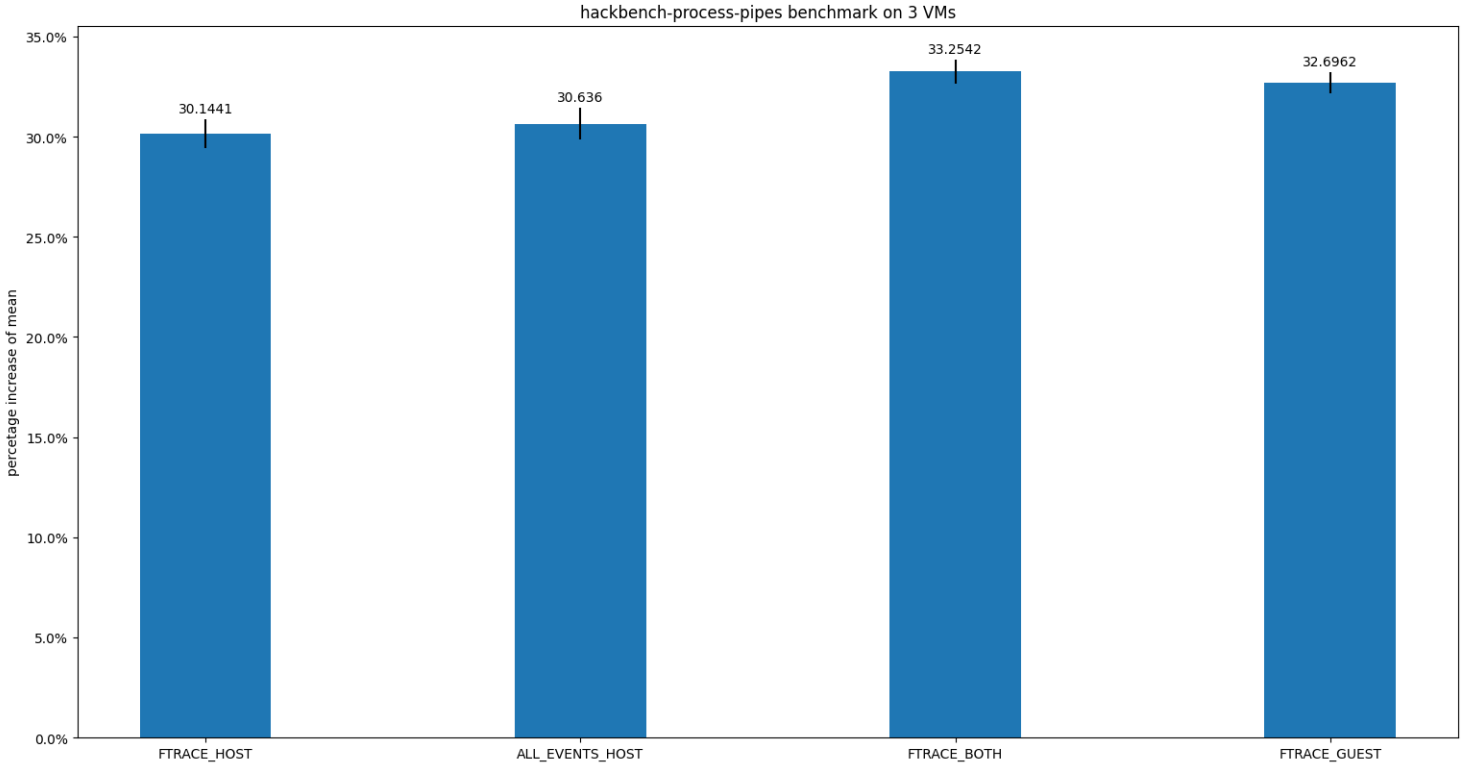


Figure 4.7: hackbench-process-pipes performance W.R.T baseline on multiple VMs

4.2.3 schbenc

Schbench is a latency measurement benchmark like the cyclictests, but it's results read a bit differently. The benchmark doesn't rely on avereage to show ,instead it uses percentiles.The results are split in different percentiles, meaning that if the 50-th percentiles shows a result of x milliseconds 50% of processes woke up before x ms, so bigger numbers mean worst response time.
As expected and anticipated while talking about the standard deviation of the latency tests (Section 4.2.1),tracing in **KVM** makes the response time less stable. Looking at Figure 4.8 as more events are traced an increase of outliers is expected.This is not the case for the 1-thread's run, for the same process of power management discussed in Section 4.2.1, but for more than 2 threads it's exaclty what is seen. In those benchmarks in particular the response time is variable between calls, this is because *ftrace* is not a consistent workload, the more monitors are triggered the greater the system responsiveness will be affected causing a chain reaction.In case of a spike in workload *ftrace* has to trace more events slowing the system even more. While the low percentiles, 50% and 70%, show just a bit of stability loss, the higher percentiles like 95% or 99%

show that tracing introduces a lot of delay in some cases causing outliers to have a big difference in value from the mean value.

0	thread	BASELINE	FTRACE_HOST	FTRACEALL_HOST	FTRACE_BOTH	FTRACE_GUEST
Lat	50.0th-qrtle-1	46.6667	33	39	6	49
Lat	75.0th-qrtle-1	47.6667	36	41	6	51
Lat	90.0th-qrtle-1	49.3333	39	46	12	52
Lat	95.0th-qrtle-1	50.3333	41	59	16	54
Lat	99.0th-qrtle-1	53.6667	55	66	29	56
Lat	99.5th-qrtle-1	53.6667	60	67	29	56
Lat	99.9th-qrtle-1	53.6667	60	82	29	56
Lat	50.0th-qrtle-2	52.3333	36	43	12	49
Lat	75.0th-qrtle-2	61.3333	42	49	35	52
Lat	90.0th-qrtle-2	62.6667	44	55	337	3452
Lat	95.0th-qrtle-2	63.3333	52	60	3820	4296
Lat	99.0th-qrtle-2	66	63	498	9488	9744
Lat	99.5th-qrtle-2	68.3333	65	1013	10192	11568
Lat	99.9th-qrtle-2	68.3333	65	1330	10192	11568
Lat	50.0th-qrtle-3	61.3333	41	49	15	45
Lat	75.0th-qrtle-3	63.6667	46	55	34	1794
Lat	90.0th-qrtle-3	65.6667	50	67	4456	8656
Lat	95.0th-qrtle-3	68	61	376	10416	10416
Lat	99.0th-qrtle-3	71.6667	3924	5848	12688	11440
Lat	99.5th-qrtle-3	72.6667	7656	7688	12720	12688
Lat	99.9th-qrtle-3	77.3333	10736	11856	12720	12688

Figure 4.8: schbench benchmark's results

4.2.4 unixbench

While analyzing **Cyclictest** and **hackbench** benchmark's results (Section 4.2.1 adn Section 4.2.2) we said that there is a correlation between overhead and the number of events traced, as clearly showed in Figure 4.2.But the **Unixbench** benchmarks showed that performance and tracing are not directly correalted. Every benckmark in the **Unixbench** suite measures how many times a program can run in a intervall of time.

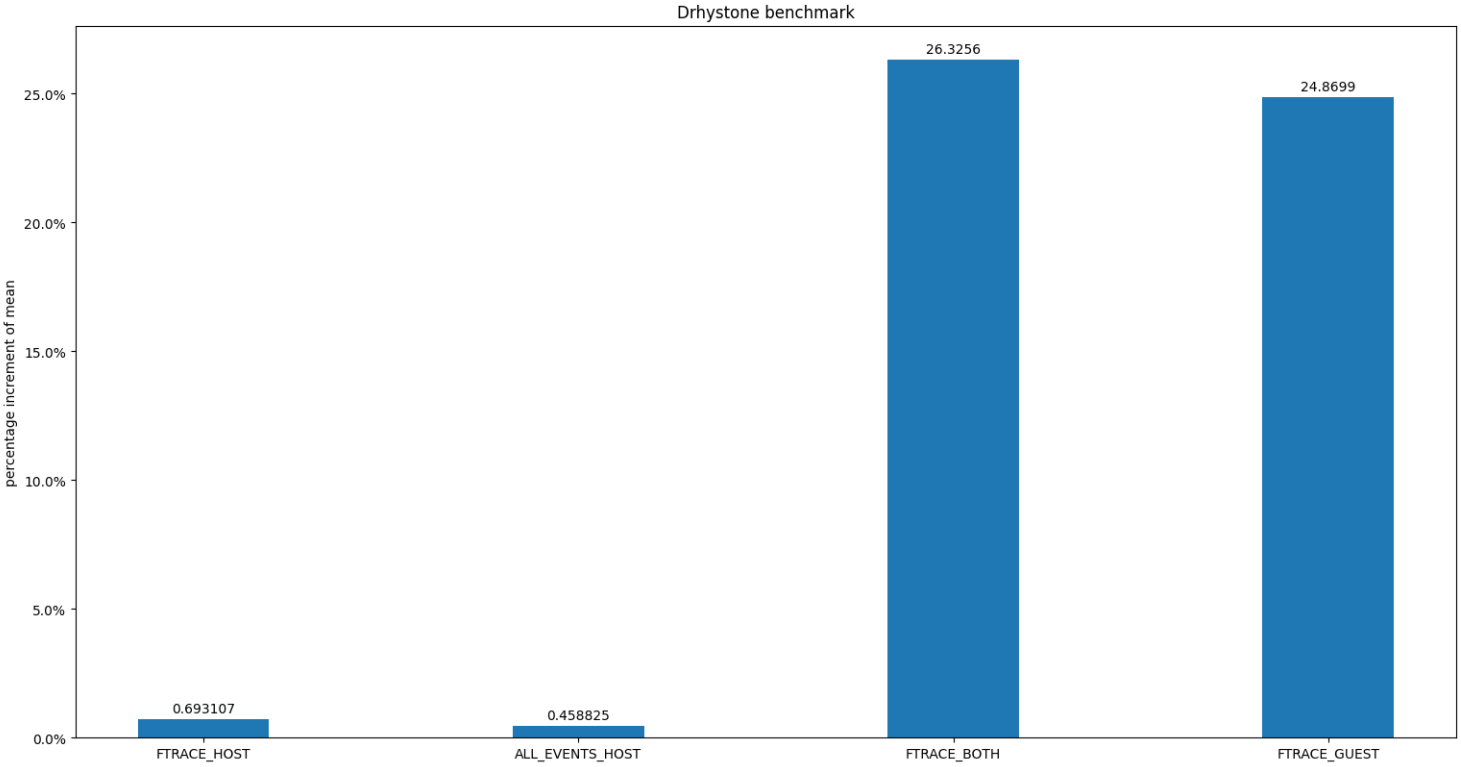


Figure 4.9: Dhrystone benchmark on single VM

As said in Section 4.2.2 it's interesting to note that there is a minimal difference between the runs with tracing enabled on host and the baseline.There is instead a noticeable drop in performance when tracing is enabled in the guest machine. This behaviour is not present when any of the Unixebenchmarks are run in the multiple VMs configuration(Figure 4.10).

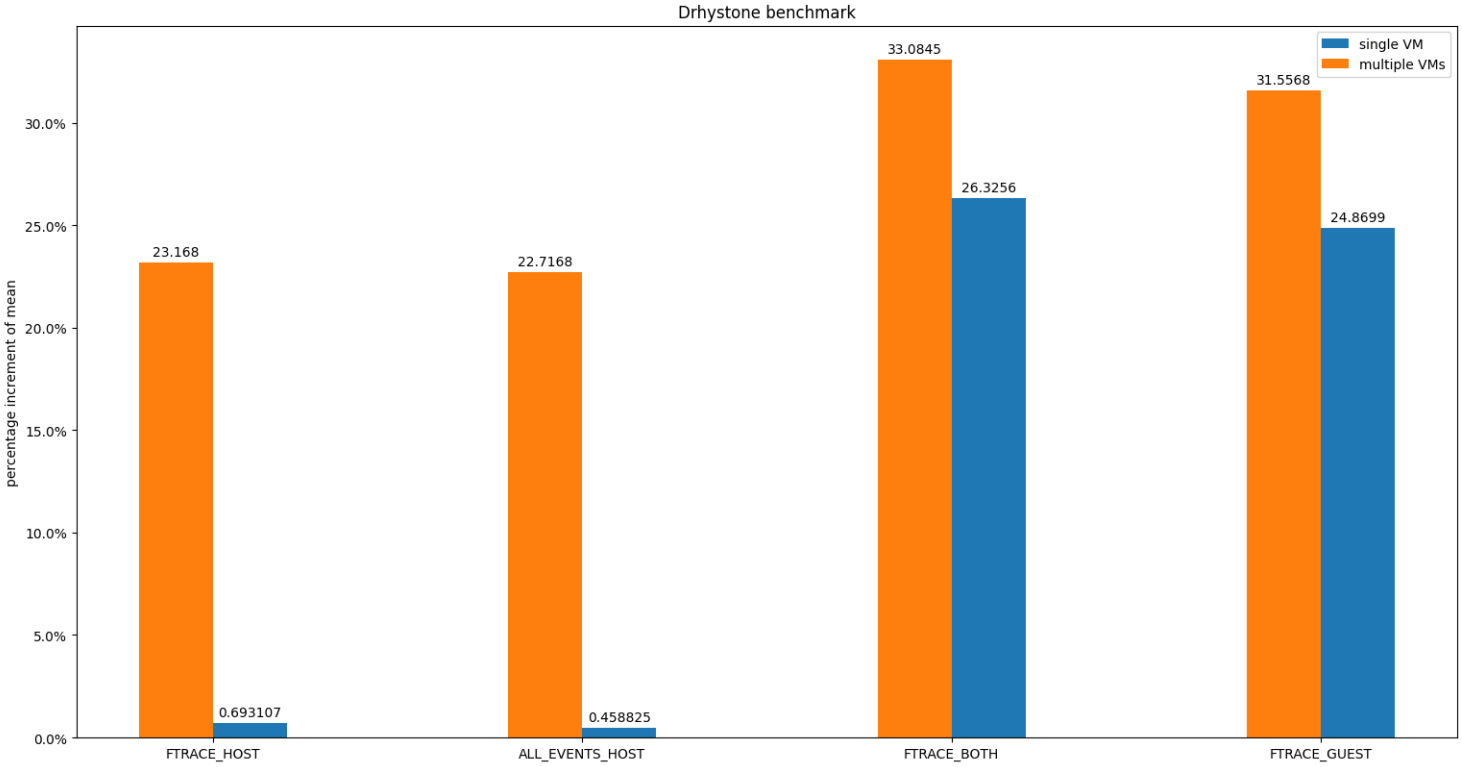
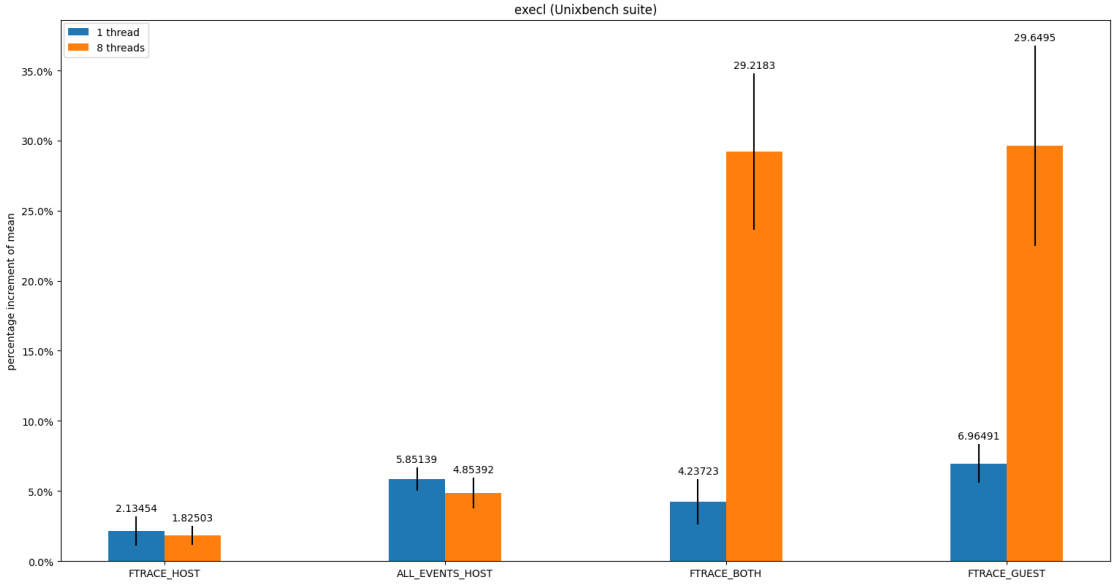


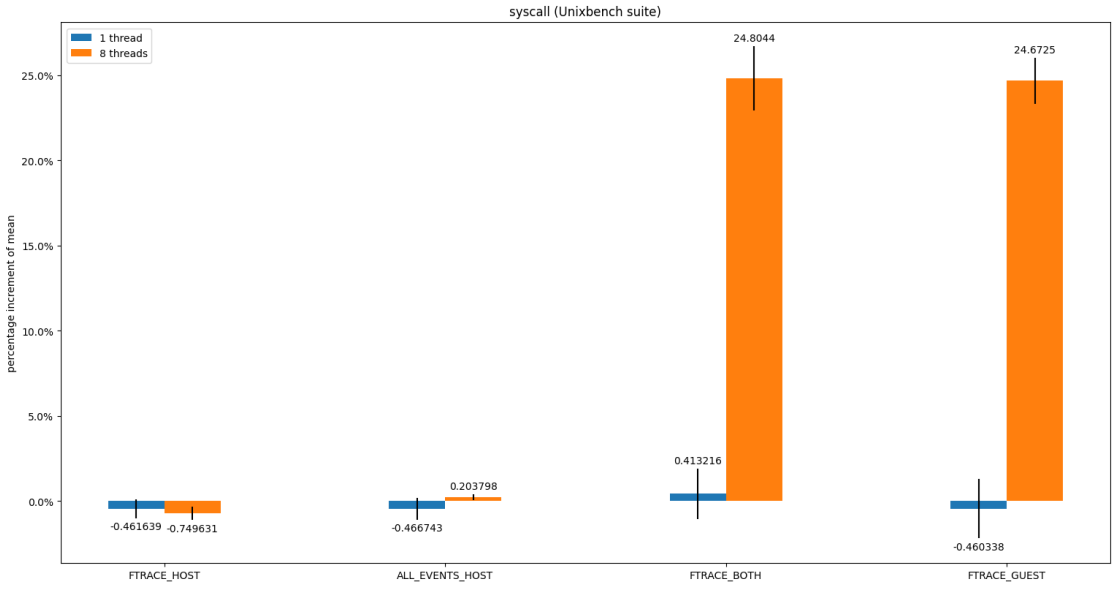
Figure 4.10: comparison between Dhrystone results in one VM and in 3 VMs

The main difference between the 2 runs is the number of CPUs that the host can use exclusively.Every virtualized CPU of the guest machine corresponds to a process in the host (Section 3.4), so if the VM has only 4 CPU only 4 processes will be created. In the worst case scenario every process will be running at the same time, occupying 4 CPUs but leaving 2 for the host to use. This means that the host always has at least 2 CPUs unloaded where the *ftrace*'s processes can run. *Ftrace*'s overhead is relevant only if the number of thread running in the host is bigger than the CPU's number.

We can onserve the same results in every Unixbench benchmark (Figure4.11), apart from the I/O ones.



(a) average of execl W.R.T. baseline run



(b) average of syscall W.R.T. baseline run

Figure 4.11: significative Unixbenchmarks ran on sigle VM

Tracing with *ftrace* has little impact on memory and on disk, but the process still has to write it's results on disk. Using a benchmark that stresses input output we can observe (Figure4.12) that indeed *ftrace* introduces overhead on disk and with more extensive research we quantified the overhead introduced in RAM. Section 4.4.2.

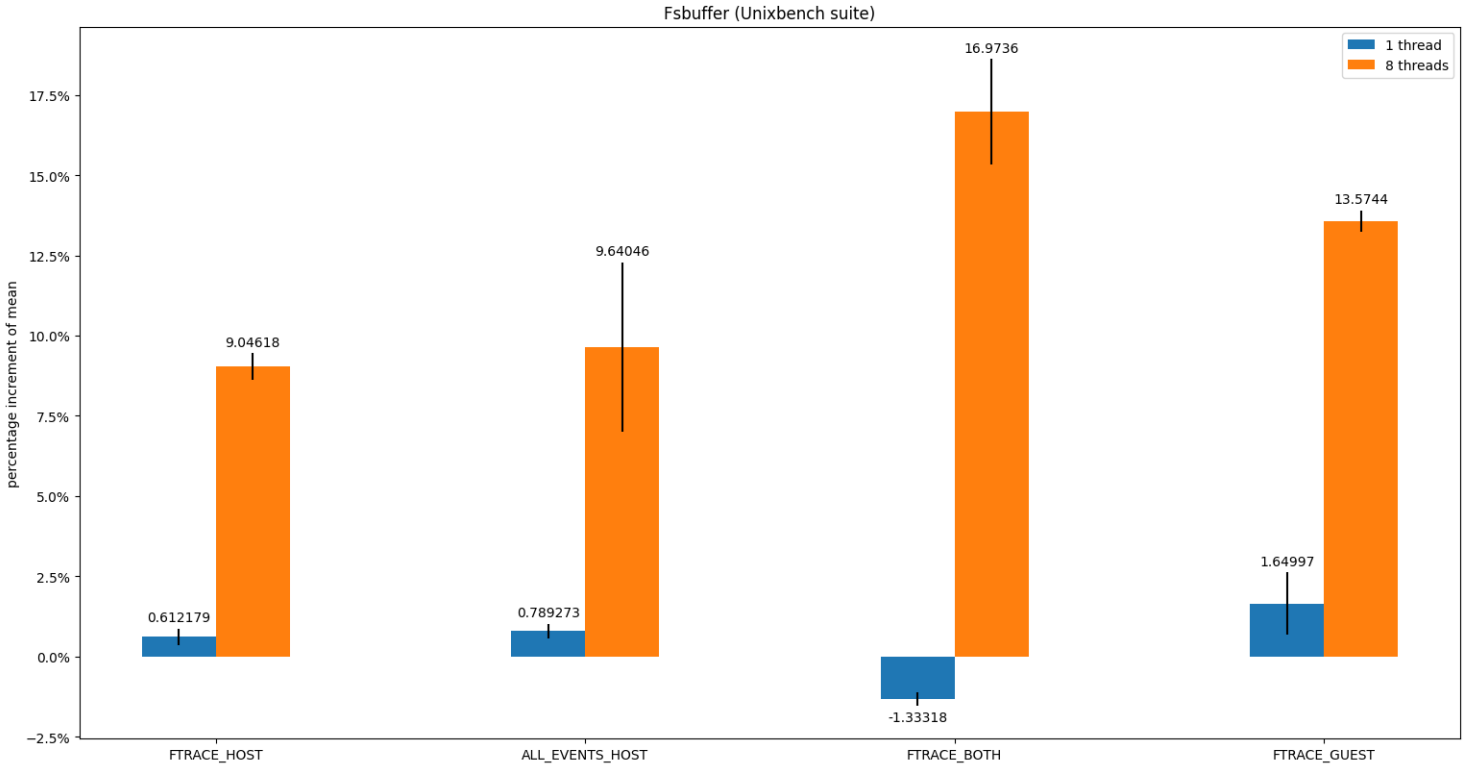


Figure 4.12: Fsbuffer on KVM, single VM

According to the results of this research *ftrace* has a big impact on CPU usage, not because of the interrupts, but because it's another process that has to compete to write data on memory and on disk.The overhead introduced by the interrupts is present, but the performance really starts to drop when not enough resources are available to run a program and *ftrace* at the same time. Comparing the run made with only 1 thread with the one made with 8 threads (Figure 4.13) we can even notice that tracing introduces more overhead in the 8 threads case.To show this we can observe the percentages that show by how the benchmark performed compared with the baseline, a minus sign shows worst performance. The 1-thread's runs are less affected by the overhead, let's study for example the **FTRACE_GUEST** run: in the 1 thread case the benchmark is 1.51% slower than the baseline, in the 8 threads it's 15.38% slower. In fact a

bigger workload means more events captured by tracing, so *ftrace*'s process has to be woken up more often introducing more overhead.

2	#threads	BASELINE	FTRACE_HOST	FTRACE_ALL	FTRACE_BOTH	FTRACE_GUEST
3 Hmean	1-thread	1020553.73 (0.00%)	1017521.40 (-0.30%)	1041017.67 (2.01%)	1022055.44 (-4.16%)	1005149.90 (-1.51%)
4 Hmean	8-thread	1227514.89 (0.00%)	1126302.06 (-8.25%)	1133053.61 (-7.70%)	1020002.90 (-16.91%)	1038769.29 (-15.38%)
5 Stddev	1-thread	4193.32 (0.00%)	1125.49 (73.16%)	4798.31 (-14.43%)	6752.96 (-61.04%)	2455.16 (41.45%)
6 Stddev	8-thread	6818.30 (0.00%)	16805.34 (-146.47%)	14454.34 (-111.99%)	5962.15 (12.56%)	2249.54 (67.01%)

Figure 4.13: difference in percentages between single thread and 8 threads on dhrystone

This is less noticable, though still present, in case of multiple VM's. Anlayzing for example the results of the Unixbench's *syscall* benchmark(Figure 4.14) we can notice that the difference between the runs is not as bis as the difference using only one VM.The explanation is the same given in the beginning of this section, while analyzing Figure 4.10, that is with CPU's to spare the overhead is less noticeable.

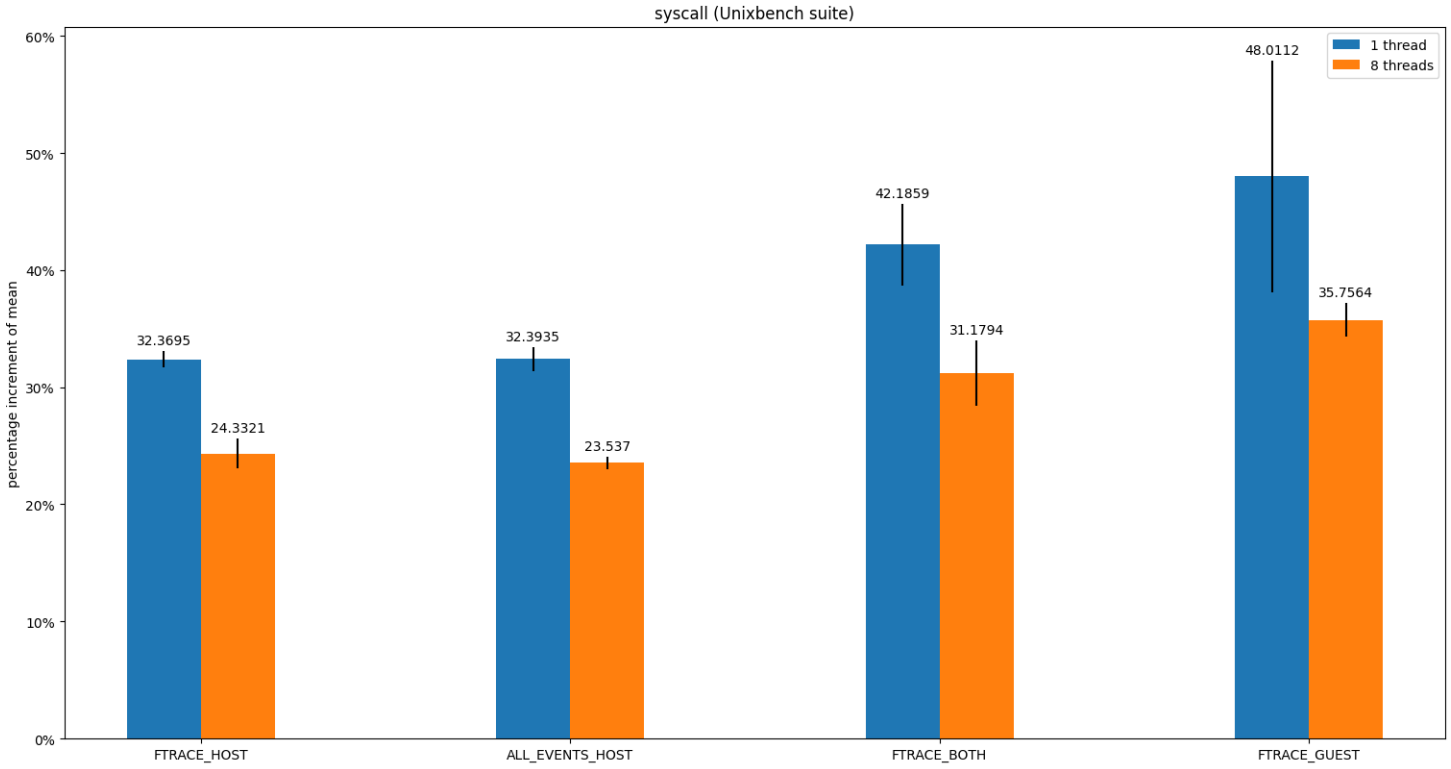


Figure 4.14: difference in percentages between single thread and 8 threads on syscall

4.2.5 sysbench-cpu

Unixbench benchmarks, in Section 4.2.4, and **Hackbench-process-pipes** in Section 4.2.2 showed a new peculiarity of *ftrace*, if the workload is low not much overhead is present.To confirm this hypotesys we ran 2 more test, **sysbench-cpu** and **sysbench-thread**, both very focused on CPU usage and threads performance. The results confirmed what we tought, as can be seen in Figure 4.15.In this case the data rappresents how long does the benchmark takes before finishing its job, so a higher percentage indicates worst performance.

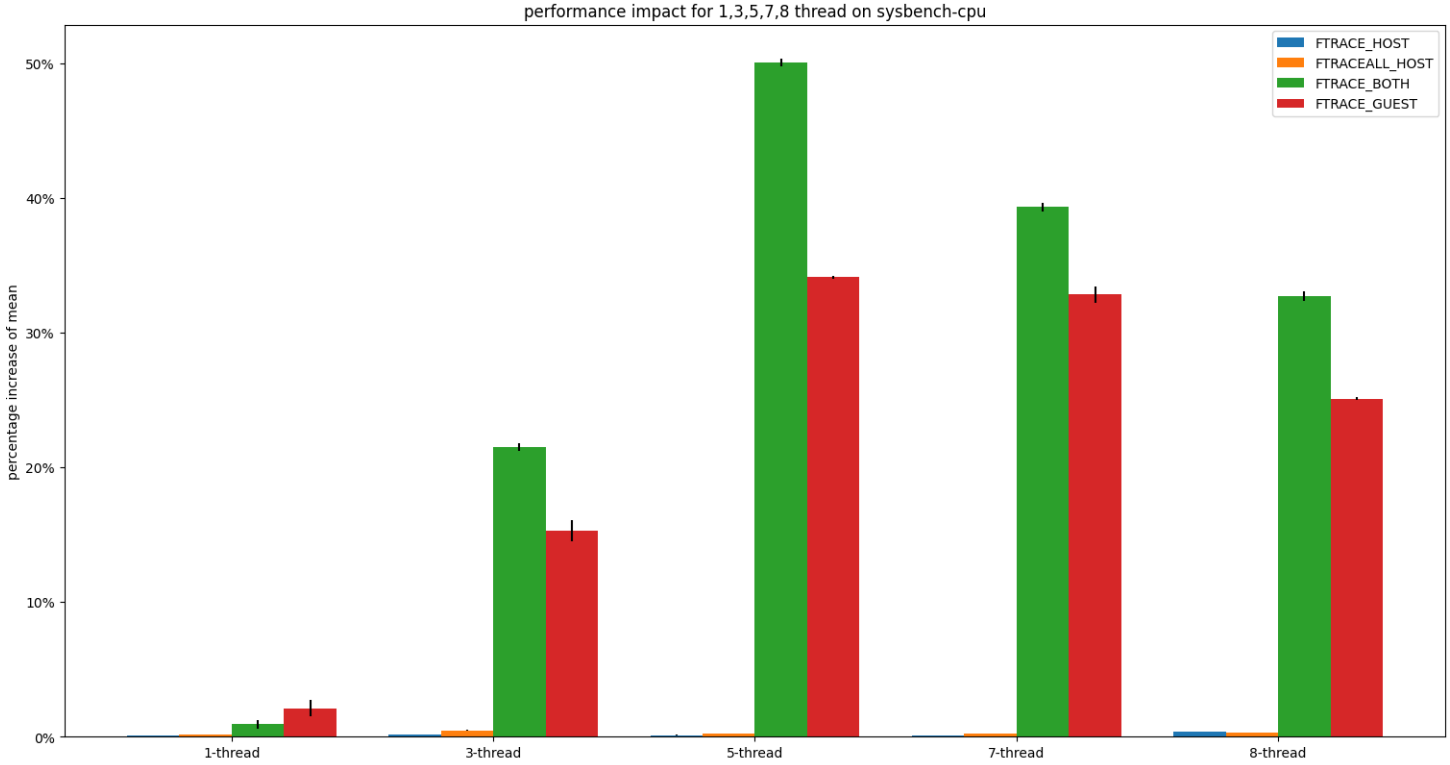


Figure 4.15: sysbnech-cpu comparison between 3,5,7,8 thread on single VM

The first thing to notice is the same behaviour found in Section 4.2.1,with only one thread the governor scales down the cpu frequency and the test performs much worst .Ignoring the first column we can see that there isn't an observable overhead unitl the number of threads gets bigger than the available CPUs (4 CPUs), even at that point the overhead is really noticeable only when tracing is enabled on guest, because as said in Section 4.2.4 while analyzing Figure 4.10 the overhead is influent only if the number of thread running in the host is bigger than the CPU's number. If we run the same benchmark in the

multi-VMs configuration we obtain the same results, but the overhead is present even when tracing on host (Figure 4.16).

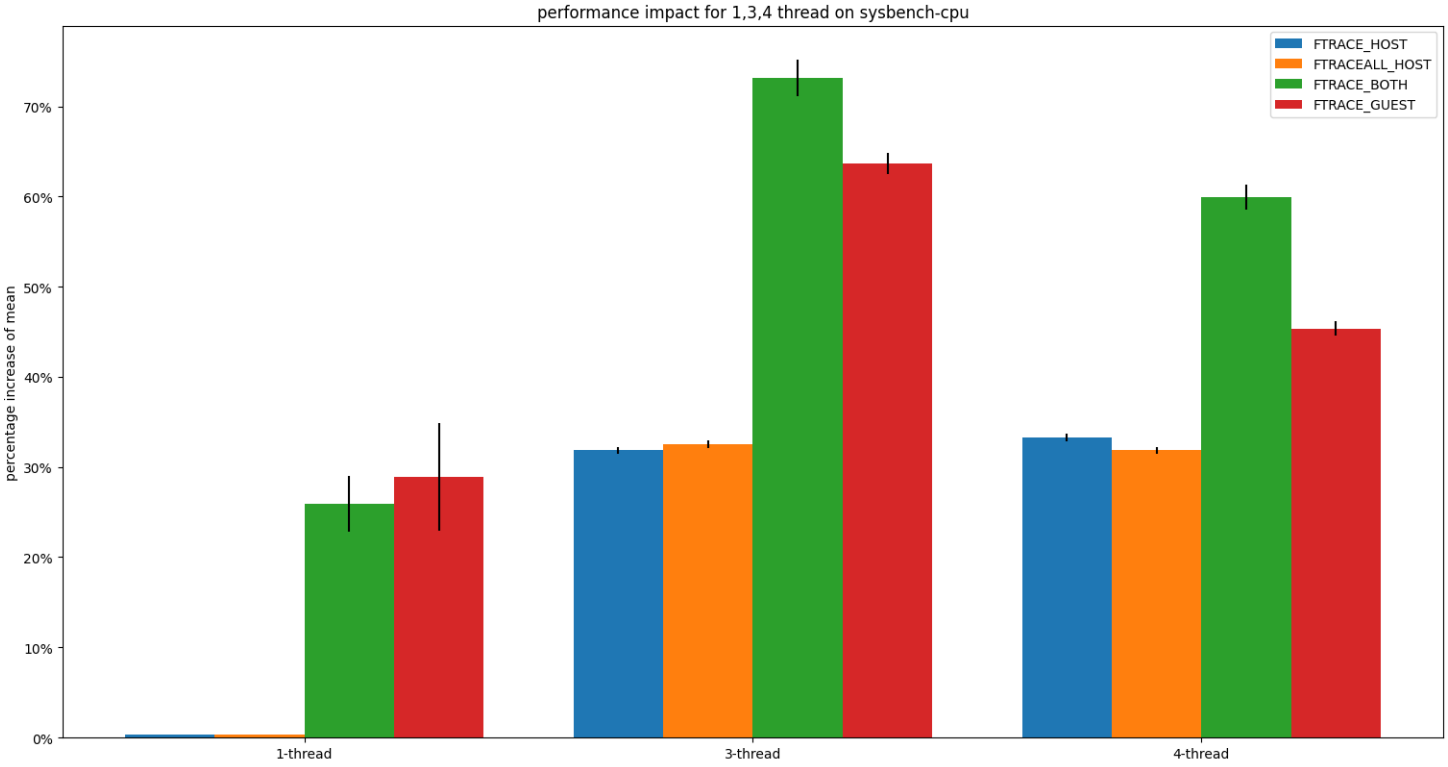


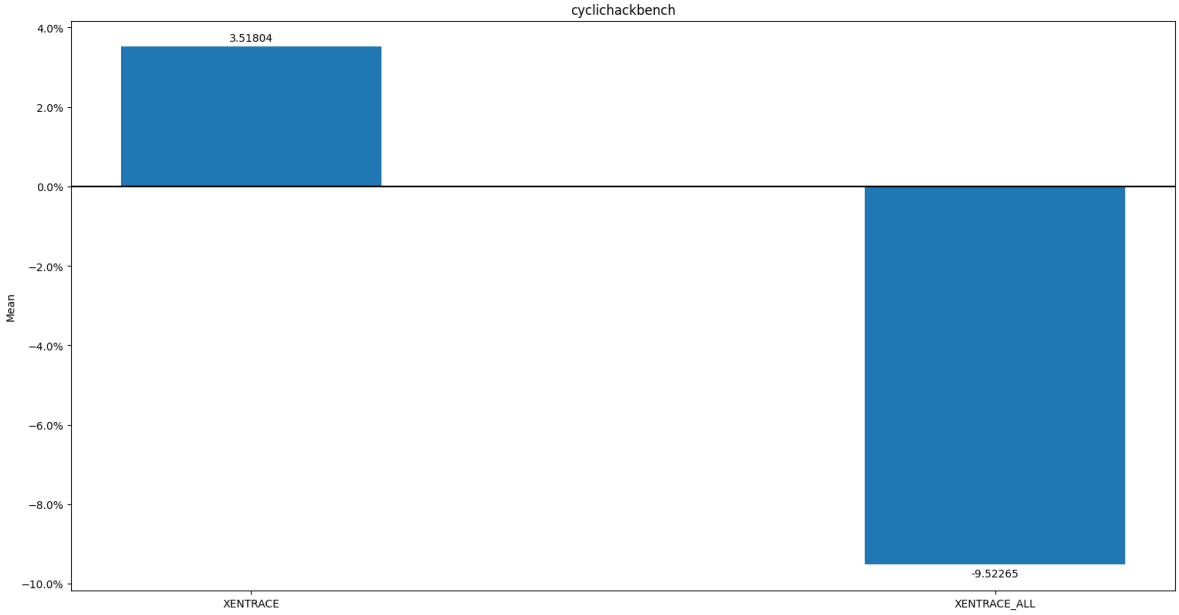
Figure 4.16: sysbnech-cpu comparison between 1,3,4 thread on multi VM

4.3 Cost of tracing with Xen

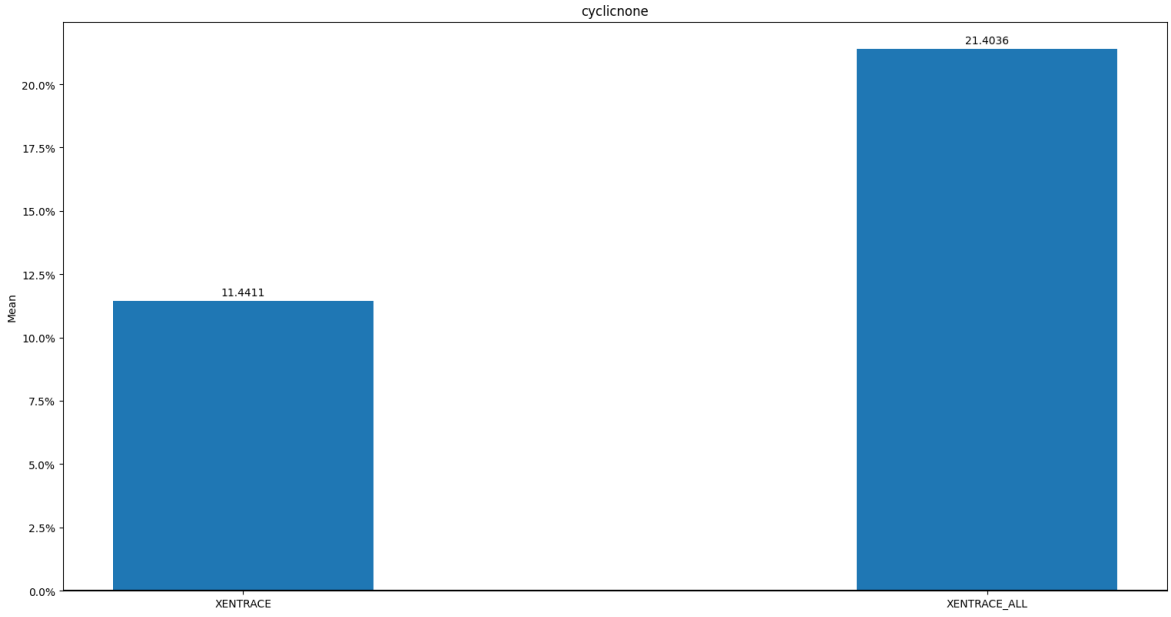
Differently from kvm, that uses *ftrace*, the tracing program for Xen is *xentrace* (Section 3.5.1).We expected *xentrace* to have a similar effect on the CPU as *ftrace* but *Xentrace* tries to prevent overheading by making more use of the memory ad less use of CPU.In particular *xentrace* saves his raw data in memory and writes them on disk only when the tracing is done and the program is closing. This caused a lot of problem during benckmarking beacause often xentrace's raw data exceded the available RAM of the machine, causing an **Out Of Memory** error on dom0 and a subsequent kernel panic.
This is not a problem in real life scenarios, because tracing is meant to be done for short periods of time and with few specific events captured, but in this research we needed longer and intensive workload, so many events were captured.The problem was bypassed using a lot of swap, but some tests could not be done at all, specifically tests with tracing enabled on guest and the tests using Hackbench-process-pipes on multiple VMs.
So for xen we have traced only on host on 2 different configurations **XENTRACE_ALL**, command *xentrace* with all the possible events enabled, and **XENTRACE**, command *xentrace* with the same events traced of **FTRACE_HOST** (so vm entry, vm exit, all scheduler events and irq events).

4.3.1 cyclictests

The first thing that can be noticed is the minimum difference between the baseline and the other runs.In fact, as said on Sec 4.3, xentrace doesn't use much cpu.This, in combination with the fact that the main source of overhead is the cpu usage and not the massive use of interrupts (Section 4.2.4), means that *xentrace* causes a very low CPU overhead. This can be seen in Figure 4.17a and Figure 4.17b



(a) Mean comparison W.R.T. baseline run in cyclicnone



(b) Mean comparison W.R.T. baseline run in cyclichackbench

Figure 4.17: cyclictests single vm

It's interesting to notice that the issue of the power management is not present in xen (Figure 4.17b), not because the hypervisor changes governor settings but because xen relies less on the CPU so the changes in frequency affect it only partially. Also in **Xen** is minimum the increase of the standard deviation found in **KVM** as can be seen in Figure 4.18, because having less loads on the cpu means a stabler response time to wakeups.

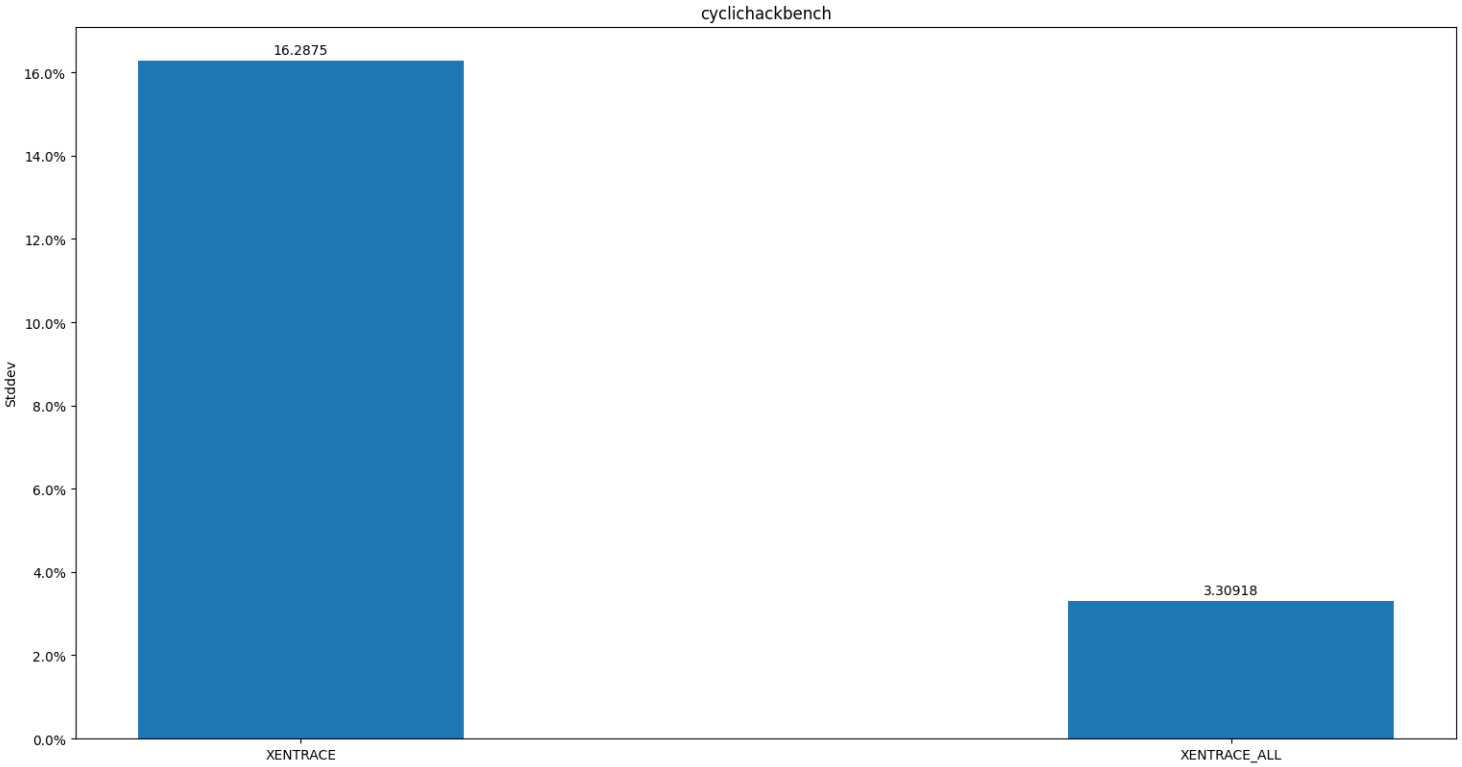


Figure 4.18: cyclictest-hackbench on single Xen vm

Results of benchmarks ran on multiple VMs can be observed in Figure 4.19, this time we observe no difference between the single-vm and the multiple-vm runs because *xentrace* has a minimal need of CPU, so it doesn't have to compete with the benkmark's processes to get CPU time.

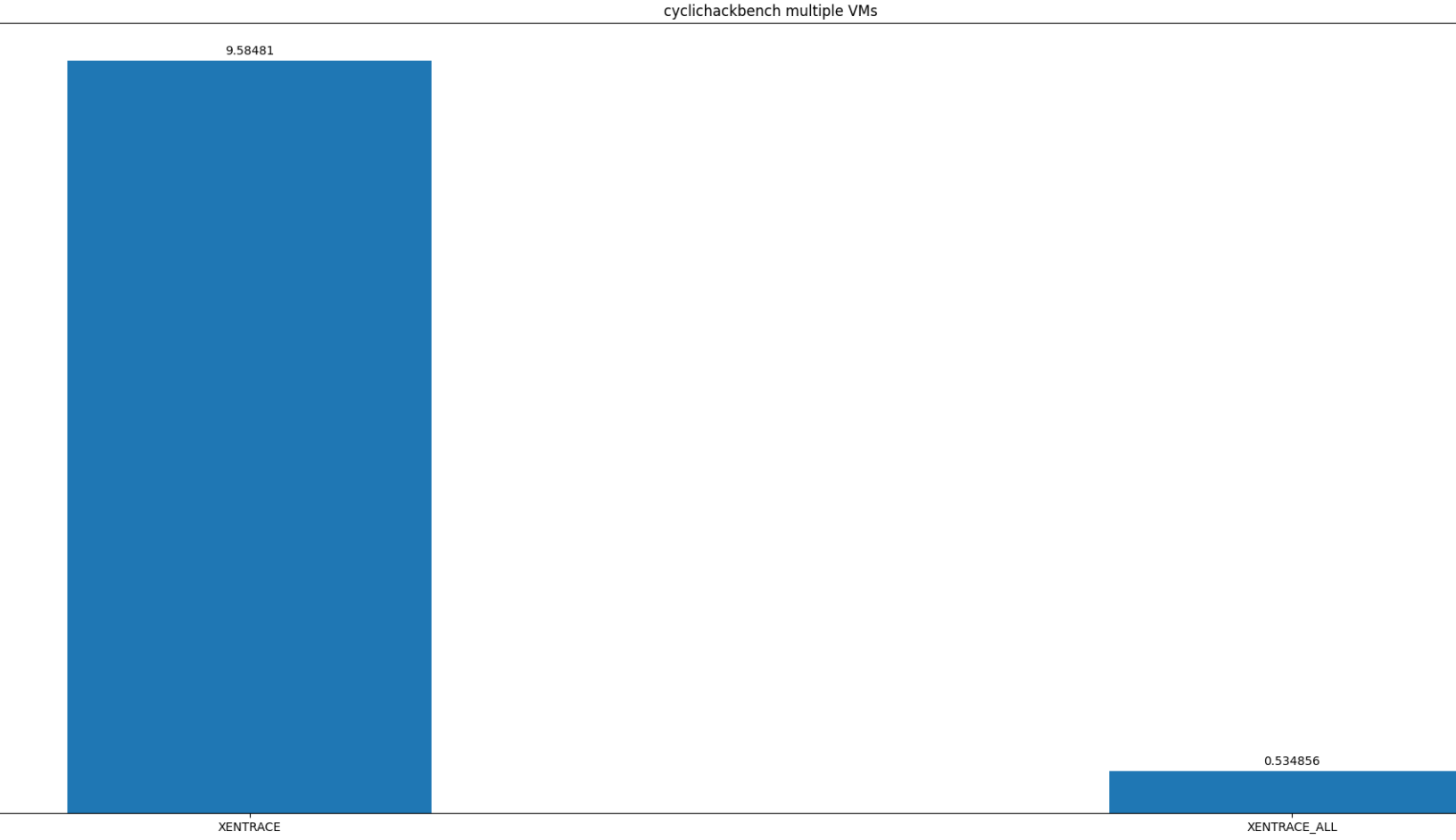
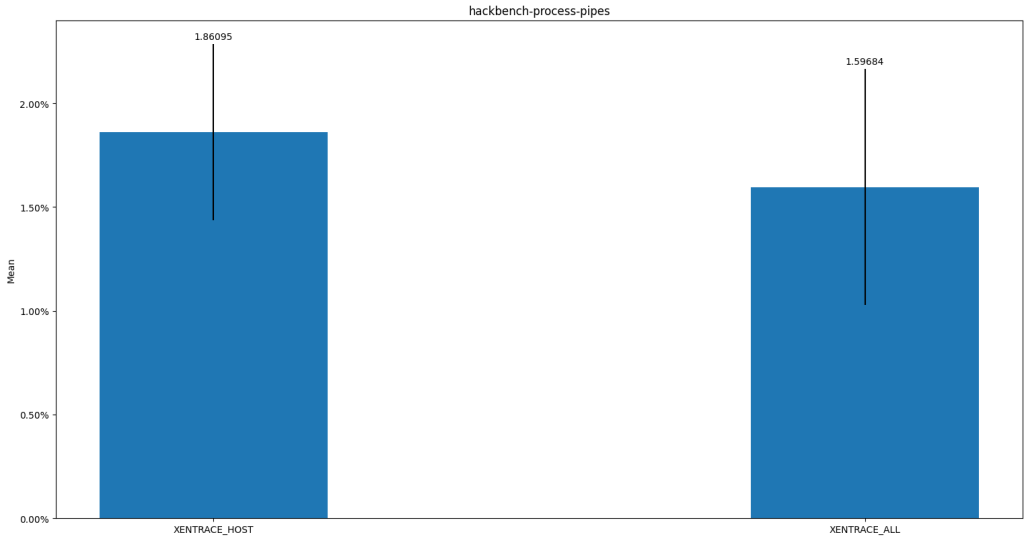


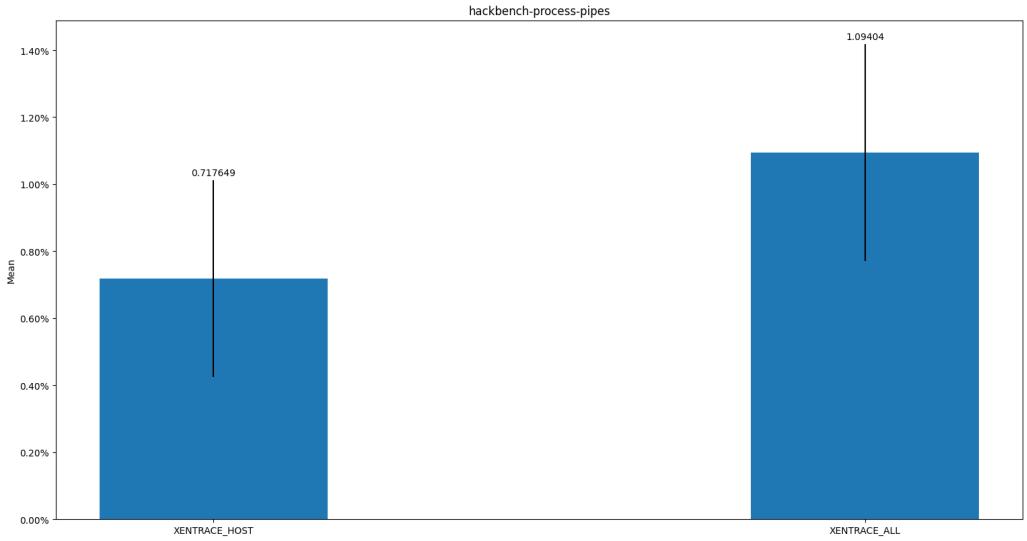
Figure 4.19: cyclictest-hackbench on multiple Xen VMs

4.3.2 hackbench-process-pipes

As said in the introduction (Section 4.3) we suspected that xen CPU usage was very low, **hackbench-process-pipes** confirmed our suspicions. As can be seen in Figure 4.20 benckmark's runs are almost identical to baseline, both on single and multiple VMs



(a) hackbench benchmark in a single VM



(b) hackbench benchmark in 3 VMs

Figure 4.20: hackbench-process-pipes benchmarks on xen

4.3.3 schbench

xentrace introduces less overhead, but it still uses interrupts to capture events and it still needs CPU time to write results to memory. So tracing with *xentrace* can still cause a chain reaction like the one observed in Section 4.3.3 so even on **Xen** tracing makes the processes response time less stable.This can be seen using **schbench** benkmark as in Figure 4.21

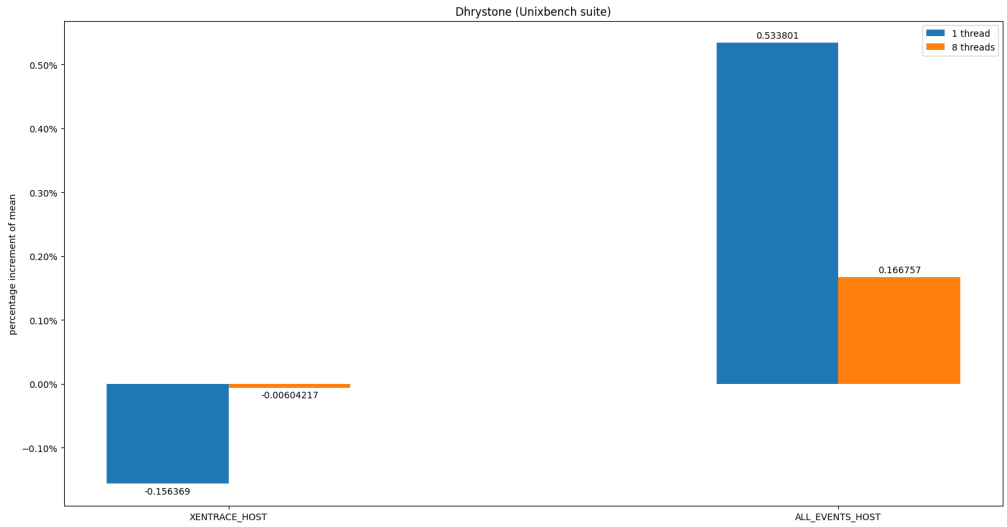
0	thread	BASELINE	XENTRACE	XENTRACE_ALL
Lat	50.0th-qrtle-1	108.333	106	120
Lat	75.0th-qrtle-1	138	128	152
Lat	90.0th-qrtle-1	165.333	154	175
Lat	95.0th-qrtle-1	173.667	175	181
Lat	99.0th-qrtle-1	187	192	188
Lat	99.5th-qrtle-1	198.333	192	188
Lat	99.9th-qrtle-1	198.667	192	188
Lat	50.0th-qrtle-2	108.333	104	110
Lat	75.0th-qrtle-2	136.333	136	142
Lat	90.0th-qrtle-2	170	163	179
Lat	95.0th-qrtle-2	181.667	173	195
Lat	99.0th-qrtle-2	209	211	205
Lat	99.5th-qrtle-2	216	237	210
Lat	99.9th-qrtle-2	223.667	264	210
Lat	50.0th-qrtle-3	115.333	115	125
Lat	75.0th-qrtle-3	143.667	141	153
Lat	90.0th-qrtle-3	170.667	169	183
Lat	95.0th-qrtle-3	188.333	187	207
Lat	99.0th-qrtle-3	215	222	234
Lat	99.5th-qrtle-3	222.333	232	272
Lat	99.9th-qrtle-3	229.333	232	463

Figure 4.21: schbench benckmark on Xen

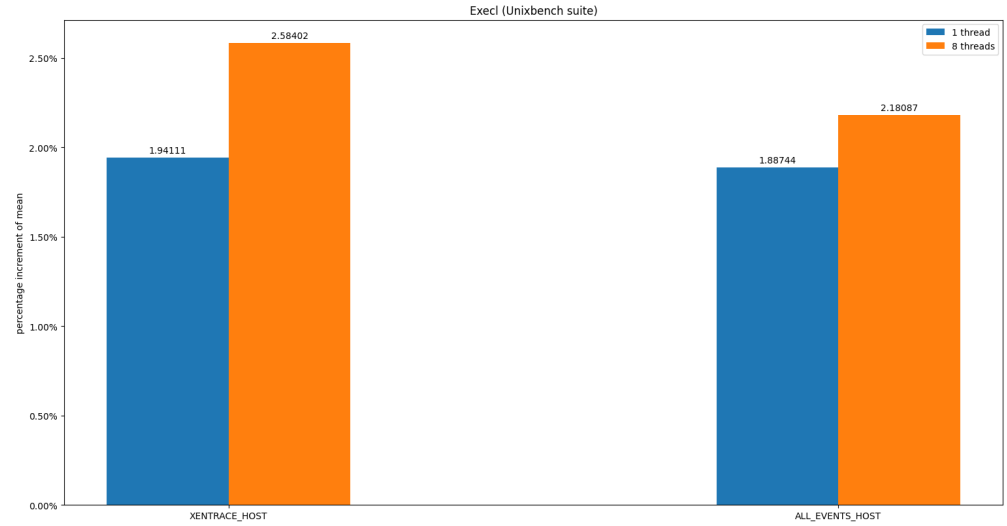
While less than the counterpart in KVM (Figure 4.8) the tables show that when considering an high percentile the response time is increased, dubled if we consider the 99,9-th percentile.

4.3.4 unixbench

Analyzing **Unixbench**'s results shows nothing more than what's been already discussed (Figure 4.22),but if we focus only to **Unixbench-io** it's clear that *xentrace* introduces overhead (Figure 4.23)and it's probably a consequence of the heavy swap usage.



(a) dhrystone benchmark in Xen



(b) execl benchmark in Xen

Figure 4.22: Unixbench on xen

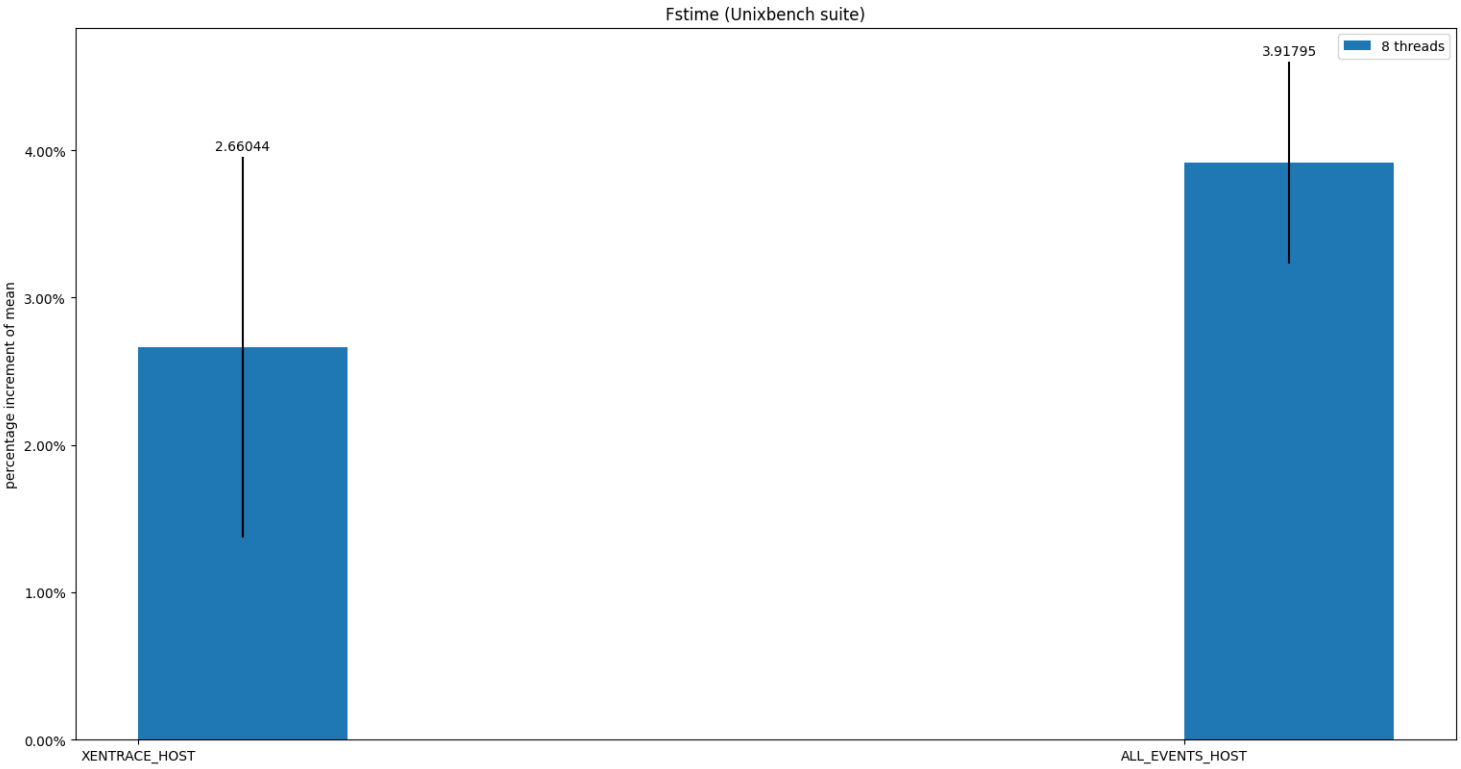


Figure 4.23: fstime benckmark on Xen

In a perfect situation heavy memory usage on dom0 shouldn't affect memory performances on the virtual machines, every virtual machine ran by the hypervisor (dom0 included) should have a fixed ammount of RAM (Section 2.3.2).But because our system doesn't have enough RAM we are forced to rely on swap, basicaly involving the hard disk in the process. The swap partition, beening accessed by all the virtual machines becomes a bottlened reducing the performance in the **Unixbench-io** benchmark when tracing is enabled (Figure 4.24).It's interesting to notice that the drop in performance is directly correlated to the number of events traced.

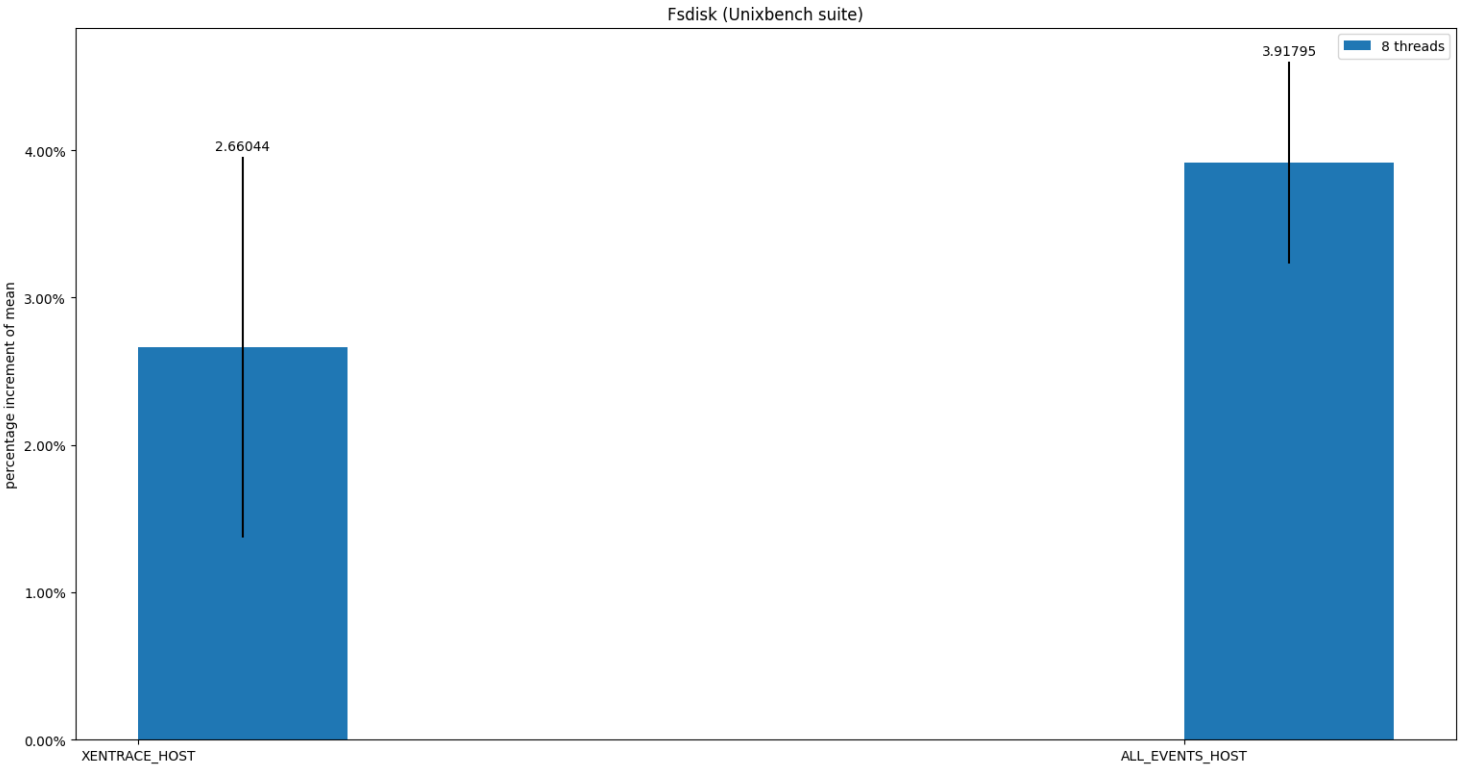
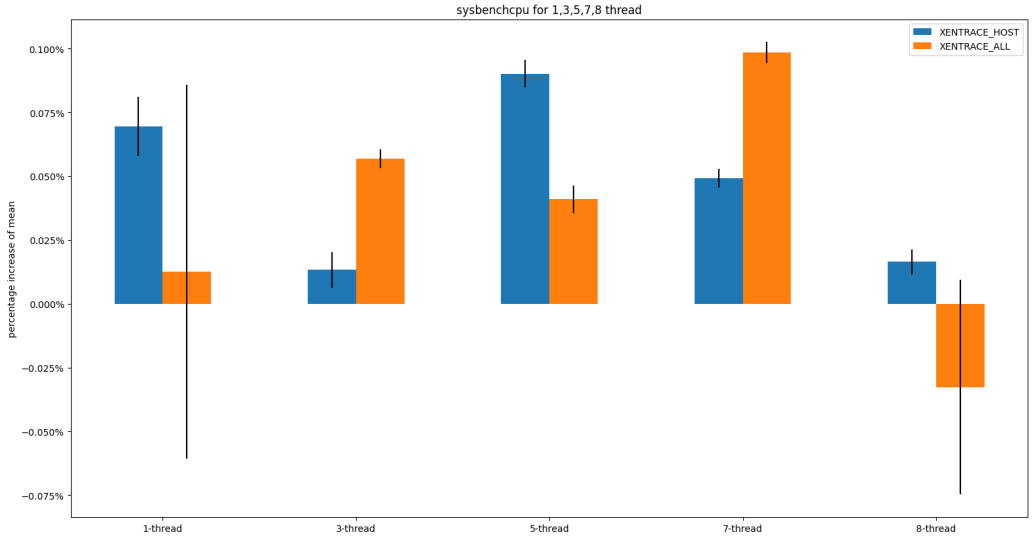


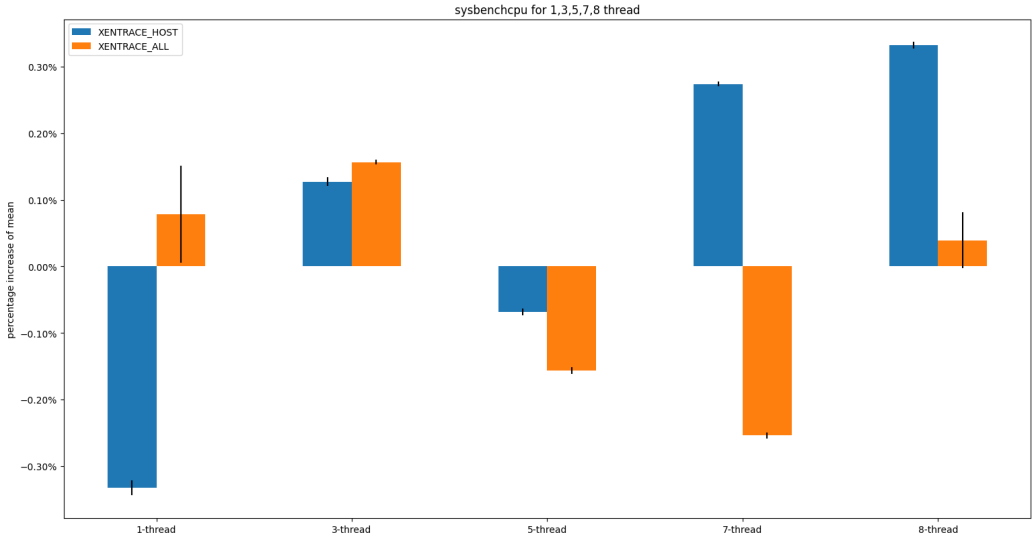
Figure 4.24: fsdisk on Xen

4.3.5 sysbench

sysbench's benckmarks are really focused on CPU and multi-threading so it' expected that, as shown in Figure 4.25, the difference between baseline runs and benckmarks is irrelevant, even using a lot of thread and stressing the CPU.



(a) sysbench-cpu benckmark on Xen



(b) sysbench-thread benchmark on Xen

Figure 4.25: sysbench benchmarks on xen

Naturally the difference between single thread runs and multi thread runs is still present(FIgure 4.25a), even if tracing doesn't affect the CPU a multi-threaded program is faster than a single-threaded one.

4.4 Comparison of the two hypervisors

All the result point to a conlcusion: tracing in **KVM** generates overhead on CPU, tracing in **Xen** generates overhead on RAM.
To prove this hypothesis more test where made this time keeping track of RAM as well as CPU.**Cyclichackbench** benckmark was used, because it's a quick benchmark that involes memory as little as possible, unlike **Sysbench** or **Unixbench**. The tests were ran in the multi-VMs configuration: 3 VMs with 2CPUs and 4GB of RAM each.

4.4.1 KVM CPU overhead

We expected a heavy CPU usage from *ftrace* and Figure 4.26 confirms that, but the figure tops out at 100, meaning that every CPU in the system is used at 100%, and that doesn't give much information.

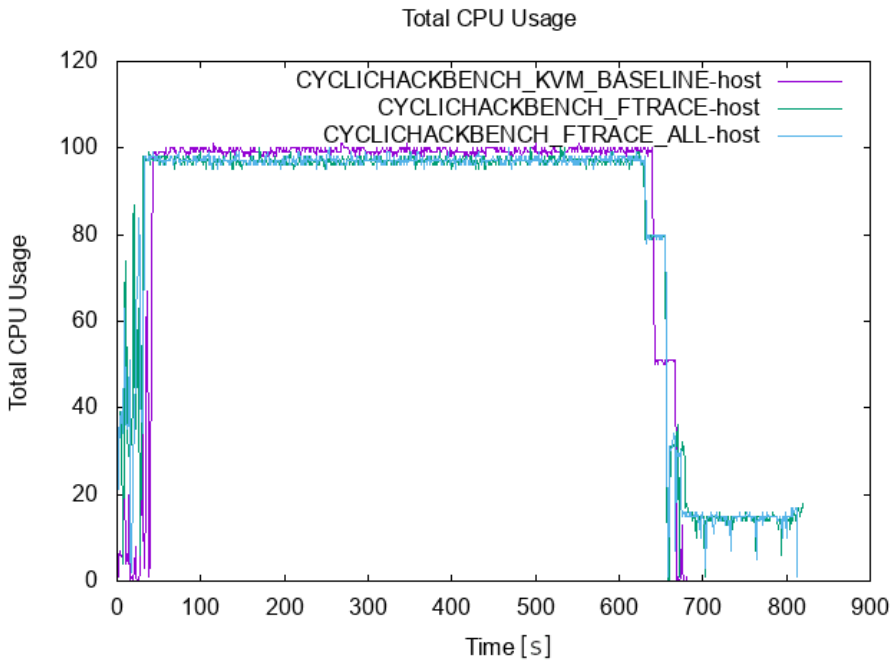
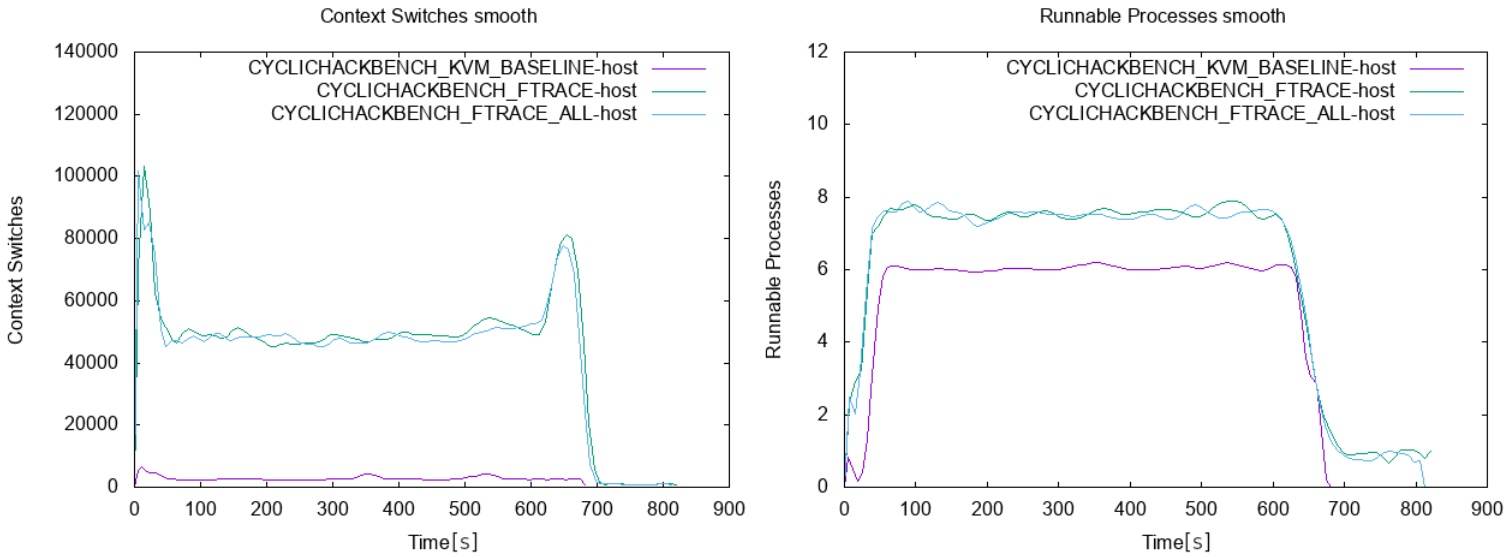


Figure 4.26: CPU usage in Xen

More important than the CPU usage in the study is the number of runnable processes and the context switches (Figure 4.27). As can be seen in Figure 4.27b even if all the CPUs are running at 100% there still are a lot of processess runnable, so processes that are waiting in the ready queue of the scheduler to receive cpu time.The difference between baseline and benckmark higlight the overhead introduced by tracing.



(a) context switches every second in kvm (b) processes runnable every second in kvm

Figure 4.27: processes running in kvm

As said in Section 3.4 every virtualized CPU of the guest machine corresponds to a process in the host, so is quite expected to see an average of 6 processes running during the baseline run, in fact there are 3 VMs with 2CPU each, so 6 virtual CPU, so 6 processes in the host. During the benchmark's runs the average is 8 process, the two processes extra are *ftrace*'s tasks, this confirms that tracing in **KVM** has a noticeable impact if the system doesn't have enough resources to run it

4.4.2 KVM memory overhead

In KVM we don't expect to see heavy RAM usage, and Figure 4.28 confirms this prediction, the available memory never goes down more than 2GB and, more importantly, there isn't a noticeable difference between baseline run and benkmarks.The decrease in available memory has to be beacuse of the test, *ftrace* doesn't seem to use much memory.

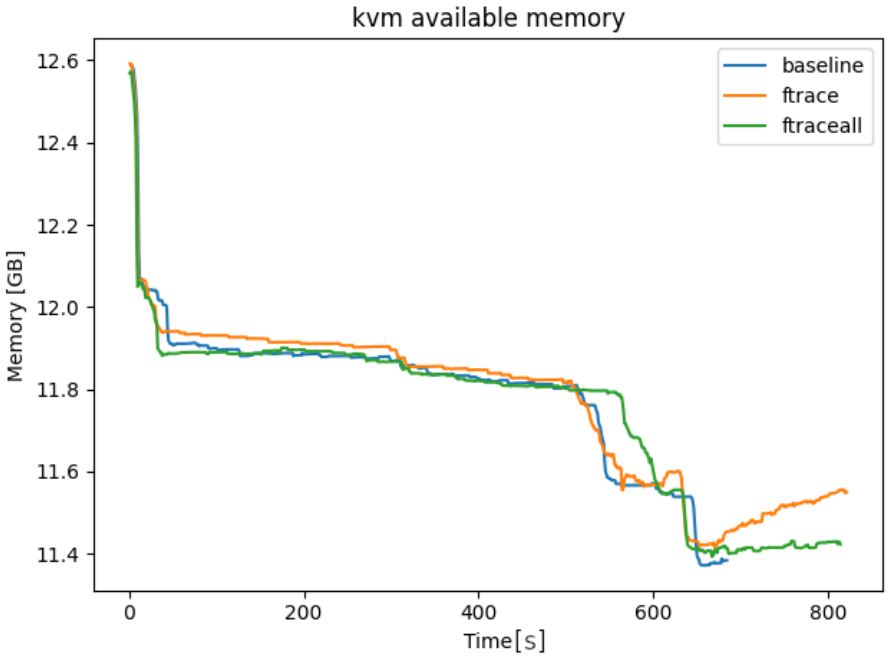


Figure 4.28: RAM usage every second in KVM

Regarding the swap we collected some results, but there was enough memory to run both the VMs and *ftrace* so the swap partition was almost unused.

4.4.3 Xen CPU overhead

Differently from **KVM** a processes analysis on **Xen** doesn't show what is really happening on the system, the only processes that we can keep track of are the ones running in dom0, for this study it's better because we can observe how tracing impacts dom0 without the influences of the running processes inside the VMs. While it could be possible to obtain the same result we got on KVM (Section 4.4.1), for example using tools like *xentop* or *xl*, this was out of the scope of this thesis.

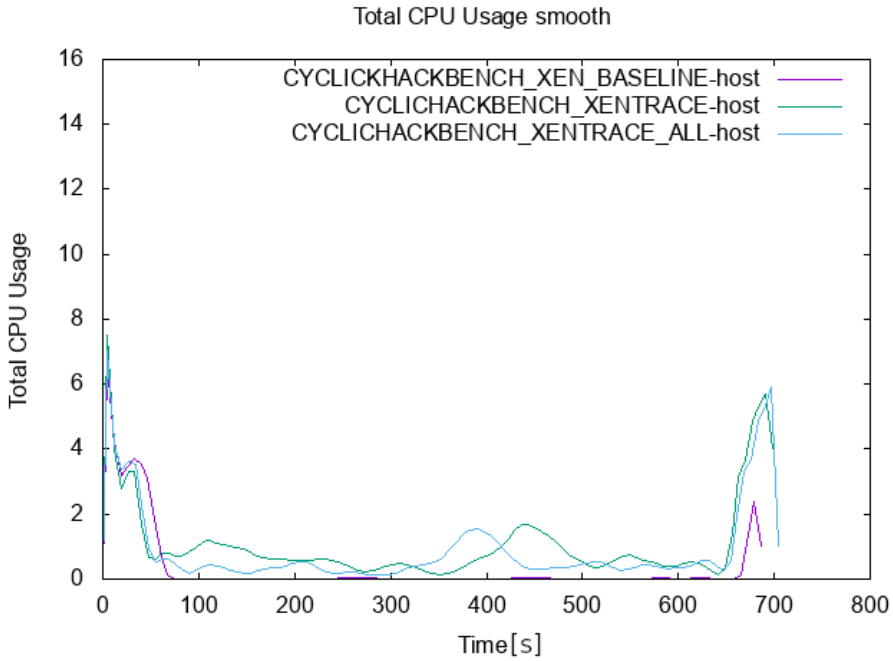


Figure 4.29: CPU usage in xen

Figure 4.29 shows dom0 CPU usage.Baseline run doesn't use any CPU, only wen starting and finishing, those are the times when the VMs tries to access to disk and to the network.During Benchmark's runs *xentrace* is running and it uses a bit of processing power, but it never saturates the system.It's not like in **KVM** (Section 4.4.1) where the tracing tools don't have the resources to run, in this case *xentrace* doesn't want to run. This can be prooven by observing the Figure 4.30 containing the runnable processes, those are few, meaning that no appl ication requested CPU time.

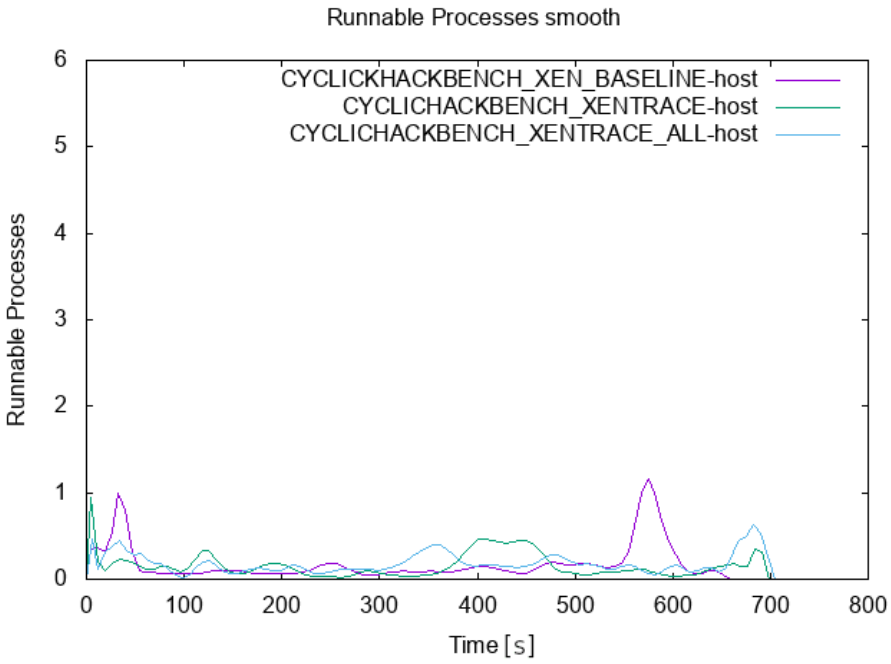


Figure 4.30: runnable processes in xen

4.4.4 Xen memory overhead

The heavy cpu usage by **Xen** was already known, but it was never studied or documented, becuase in no real situation tracing needs to be done for more of a cou ple of seconds. Figure 4.31 shows memory availability.

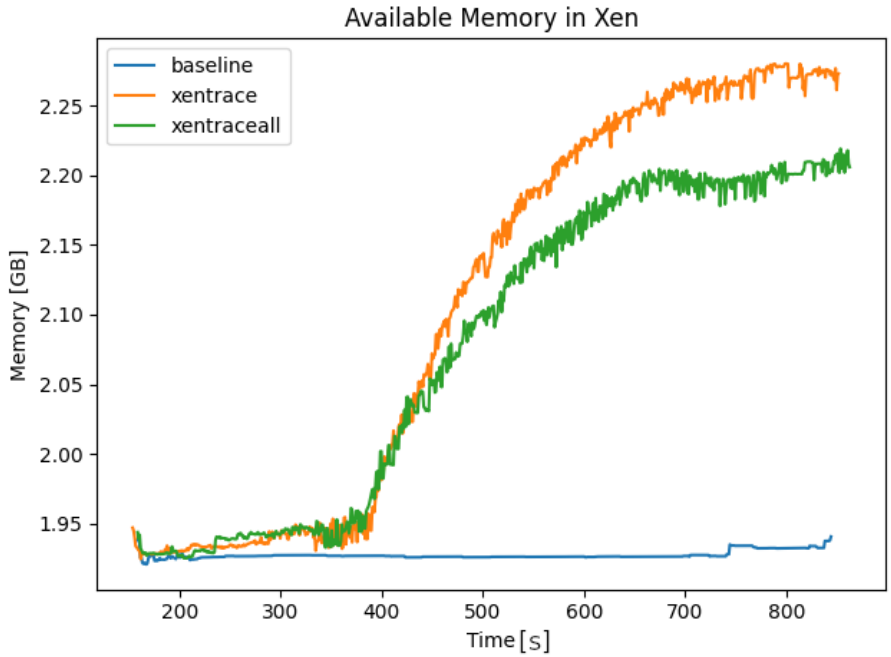


Figure 4.31: RAM usage in Xen

The small memory (3GB) runs out pretty fast, when the monitor starts registering it's already down to 1.95GB but observing the baseline run we can observe a different behaviour from the benckmark's run. The baseline run seems stable, it runs smootly with a few MB to spare; the memory is used to run dom0 and probably some processes that controls VMs status, network connection and VMs access to disk.Instead the benckmarks starts needing memory but soon seems to release it.
In reality the memory is not freed,the system actually moves the *xentrace*'s raw data to the swap partition and keeps the RAM as cache, prioritising process pages to be stored in RAM, this means a slover access to the data but avoids a kernel panic.We can see that the memory is been swapped by observing the fluctuating memory avalability, not present in the baseline graph, or by analyzing swap usage in Figure 4.32

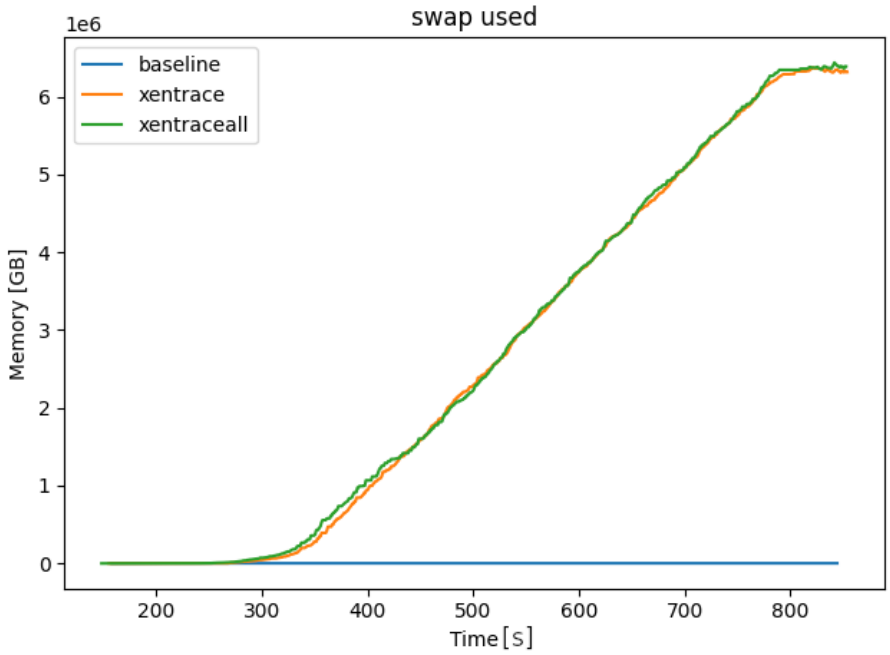


Figure 4.32: RAM usage in Xen

This graph makes clear how *xentrace* makes heavy uses of the swap partition.The baseline run shows almost not swap usage (1MB at maximum), instead whith *xentrace* running dom0 keeps asking for more space until the end of the test, reaching more than 6.5 GB.

Bibliography

[1] Dario Faggioli. https://xenproject.org/2012/09/27/tracing-with-xentrace-and-xenalyze. Tracing with Xentrace and Xenalyze.

[2] eBPF. http://www.brendangregg.com/ebpf.html. Linux Extended BPF (eBPF) Tracing Tools.

[3] How Does Xen Work? http://www-archive.xenproject.org/files/Marketing/HowDoesXenWork.pdf. A Xen virtual environment consist of several items that work together.

[4] LTTng. https://lttng.org. LTTng is an open source tracing framework for Linux.

[5] MMTest. https://github.com/gormanm/mmtests. MMTests is a configurable test suite that runs performance tests against arbitrary workloads [...] Support exists for gathering additional telemetry while tests are running and hooks exist for more detailed tracing using ftrace or perf.

[6] perf_events. http://www.brendangregg.com/perf.html. perf_events is an event-oriented observability tool, which can help you solve advanced performance and troubleshooting functions.

[7] strace. https://strace.io. strace linux syscall tracer.

[8] Sysbench. https://wiki.gentoo.org/wiki/Sysbench.

[9] Sysdig. https://sysdig.com/blog/sysdig-tracers. Sysdig tracers track and measure spans of execution in a distributed software system.

[10] trace-cmd. https://linux.die.net/man/1/trace-cmd. trace-cmd - interacts with Ftrace Linux kernel internal tracer.

[11] Xen Hypervisor. https://en.wikipedia.org/wiki/Xen. Is a type-1 hypervisor that allow multiple operating systems to execute on the same computer concurrently.

[12] Xen Project Software Overview. https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview.

[13] xenalyze. `https://github.com/mirage/xen/blob/master/tools/xentrace/xenalyze.c`.

[14] xentrace. `https://github.com/mirage/xen/blob/master/tools/xentrace/xentrace.c`.

[15] xentrace_format. `https://github.com/mirage/xen/blob/master/tools/xentrace/xentrace_format`.

[16] xentrace_parser v0.1.0. `https://github.com/giuseppe998e/xentrace-parser`.

[17] Pariseau Beth. Kvm reignites type 1 vs. type 2 hypervisor debate. `https://searchservervirtualization.techtarget.com/news/2240034817/KVM-reignites-Type-1-vs-Type-2-hypervisor-debate`, 2011.

[18] Lorenzo Brescia. Coupling tracing over host and guest machines. Master's thesis, University of Turin, December 2020.

[19] Oracle Corporation. VirtualBox. `https://www.virtualbox.org`. A free and open-source hypervisor for x86 virtualization.

[20] Mathieu Desnoyers. Ph.d. dissertation: Low-impact operating system tracing. `https://lwn.net/Articles/370992`.

[21] Docker. `https://www.docker.com`. A software that use operating system virtualization to deliver software packages via containers.

[22] Giuseppe Eletto. Development of a kernelshark plugin for xen traces analysis. Master's thesis, University of Turin, June 2021.

[23] Cristian Monticone. Impact of virtualization layers onto scheduling policies. Master's thesis, University of Turin, October 2019.

[24] Marco Perronet. Linux kernel: monitoring the scheduler by trace sched* events. Master's thesis, University of Turin, July 2019.

[25] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, July 1974.

[26] QEMU. `https://www.qemu.org`. A free and open-source hypervisor that performs hardware virtualization.

[27] Diomidis Spinellis. Trace: A tool for logging operating system call transactions. `https://www2.dmst.aueb.gr/dds/pubs/jrnl/1994-SIGOS-Trace/html/article.html`.

[28] OpenVZ. `https://openvz.org`. An operating system level virtualization software for Linux.