Università degli Studi di Torino

DIPARTIMENTO DI INFORMATICA
Corso di Laurea Triennale in Informatica

TESI DI LAUREA

A study of Intel's Speed Shift power management technology

Candidato:
Federico Serra
Matricola 898925

Relatore:
Enrico Bini
University of Turin

Correlatore:
Giovanni Gherdovich
SUSE

Anno Accademico 2020/2021

Contents

Acknowledgments **iv**

Abstract **v**

1 Basics of Linux kernel **1**
1.1 Operating Systems 1
1.2 A general overview 2
1.2.1 System calls 3
1.2.2 A different kind of software 5
1.2.3 User and kernel stacks 6
1.2.4 A monolithic design 7
1.3 Process management 10
1.3.1 Processes and threads 11
1.3.2 List implementation 14
1.3.3 Scheduling 17
1.3.4 Tasks lifetime 20

2 Tracing events in Linux **23**
2.1 Introduction 23
2.2 Tracing tools 24
2.2.1 strace 24
2.2.2 ltrace 25
2.3 Tracing with ftrace 26
2.3.1 Interfacing with ftrace 26
2.3.2 Function tracing 28
2.3.3 Event tracing 30
2.3.4 trace-cmd 32

3 The Intel Skylake family of processors **34**
3.1 Introduction 34
3.2 Performance states 34
3.3 MSR 35
3.4 Hardware-Controlled Performance State 37
3.4.1 HWP software interface 38

4 The SystemTap tool **40**
4.1 Introduction 40
4.2 Language overview 41
4.2.1 Types of scripts 41
4.2.2 Built-in functions 41
4.2.3 Probe points 42
4.2.4 Target variables 43
4.2.5 Embedded C and guru mode 44
4.2.6 Statistics and aggregates 44
4.3 Command line flags and arguments 45
4.4 Real-word example and operating principle 46

5 Extracting processor internal features **48**
5.1 Introduction 48
5.1.1 A simple overview 48
5.2 The tool chain 49
5.2.1 Systemtap script 49
5.2.2 rt-app workload 53
5.2.3 bash script 55
5.2.4 Data analysis in Jupyter 58
5.3 Testing machine setup 60
5.4 Experiments 61
5.4.1 Steps of frequency growth 61
5.4.2 Frequency switch overhead 67
5.4.3 Turbo always on 70
5.4.4 Temperature variations 73

List of Figures

1.1 Kernel space (in red) and user space (in green and blue) 3
1.2 The Linux kernel stack 7
1.3 The most popular kernel designs and their differences 8
1.4 A portion of the kernel subsystems map 10
1.5 Pid hash table, pids 199 and 29384 are both hashed to 199 . . . 14
1.6 Hash table for the TGID pid type 15
1.7 A generic doubly linked circular list 16
1.8 State machine of task states 22

2.1 The procfs special filesystem 27
2.2 Tracing folder inside the debugfs special filesystem 27

2.3 Types of tracers on by distribution (OpenSUSE Tumbleweed) . . 28
2.4 List of macro-event's section 30
2.5 Every event associated with kvm 30
2.6 Control files for the `kvm_entry` event 31
2.7 Output of the command *trace-cmd record* 32
2.8 Output of the command *trace-cmd report* 33

3.1 P-states schema. Image borrowed from IDF15 [32] 37
3.2 Most efficient P-state. Image borrowed from IDF15 [32] 38

5.1 rt-app code 54
5.2 Example ipywidgets 60
5.3 Vertical alignment of three sample periods. Dashed squares are zoomed on the right for better understanding. Lines of the same color identify the same phenomenon. 62
5.4 Regular, step-like, APERF tendency 63
5.5 Plot with frequency 63
5.6 Drop example 64
5.7 Plot of collected steps, zoom of one period 67
5.8 Distribution plot of collected steps 68
5.9 Intel's data on HWP 68
5.10 Plot with MPERF 69
5.11 Frequency overhead parameters illustrated on sample period. . . 70
5.12 APERF stays "high" 71
5.13 Light benchmark 72
5.14 Intensive benchmark 72
5.15 $\frac{APERF}{MPERF}$ ratio 73
5.16 Temperature variations registered simultaneously on all cores . . 76
5.17 APERF values traced on CPU 0 77

List of Tables

3.1 MSRs involved in DVFS, performance counters 35
3.2 IA32_HWP_REQUEST register, address 0x774 36
3.3 Auxiliary MSRs involved in DVFS 36

5.1 Environment specifications 60
5.2 Registered samples of nomimal and drop frequencies. Column *result* shows the overhead in microseconds computed using formula 5.2. 70

Acknowledgments

Part of Chapter 1 is borrowed from the thesis by Marco Perronet [31] and Stefano Chiavazza [24]. Part of Chapter 2 is borrowed from the thesis by Marco Perronet [31] and Lorenzo Brescia [22]. Part of the content of Chapter 3 is borrowed from Energy Efficient Server [25], Inside 6th-Generation Intel Core [27], and two Intel's Software Developers Manual: System Programming Guide [29] and Model Specific Registers [30]. Part of Chapter 4 is borrowed directly from `stap` man pages [15], the SystemTap Language Reference [37], the SystemTap Beginners Guide [35] and the SystemTap Tapsets Reference Manual [36].

I would like to thank my family for the love, support and never-ending patience that made me who I am now. A special thanks to Chiara, who has been the best girlfriend I could ever dream of. Thanks also to my friends, the ones always there and the ones that took different paths. Last but not least, a thanks to Professor Enrico Bini and Giovanni Gherdovich, that guided me through this journey, transmitting me their passion and knowledge, always with enthusiasm, empathy and good spirits.

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

In modern processor design, lowering the energy consumption is of paramount importance. In PCs as well as in data centers, the CPU is the component consuming the most energy. Intel, from the Skylake processor generation and on, developed a hardware micro-controller, which is responsible for keeping the operating frequency at its optimal value, maximizing the performance per Watt. The logic behind such a microcontroller, however, is not disclosed. Hence the operating system and the HW controller may set contradictory operating points, leading then to performance degradation. In this work, we investigate non-documented features of the frequency scaling policy of Intel's Skylake processors, known as Intel Speed Shift or Hardware-Controlled Performance State (HWP). In our experiments, we investigated the response of the frequency controller in response to known workload patterns. A few characteristics were detected, analyzed and explained. Others are left as future work.

Chapter 1 outlines the main characteristics of the Linux Kernel, focusing on processes. Chapter 2 contains an overview of tracing, ftrace and other elements to make the tracing. Chapter 3 presents the technology of interest, HWP, and its components. Chapter 4 introduces the SystemTap tool, its syntax and usage. Chapter 5 explains the experiments conducted in this research and the various tools that made it possible.

Chapter 1

Basics of Linux kernel

1.1 Operating Systems

The operating system (OS) comprises the software intended to manage the hardware resources and the *application software*, which performs specific, high-level tasks. The application software, which is the larger part of the OS, is made of utility programs and any other software with which the user interacts directly. These programs are not part of the core OS. Rather, they are necessary to do anything useful. The operating system acts as an intermediary between the user and the machine by abstracting away the hardware, which makes interaction easier: this is why almost every computer runs an operating system.

It can be argued that the OS is not strictly necessary because it is possible to execute a program without loading an OS: this is referred as *bare metal programming* and it is common in small size embedded systems. Because there is no operating system (which means no file system, memory management or any useful application such as compilers), programs cannot be written on the system itself. Instead, the program is written on another machine with an operating system, then compiled with a cross-compiler, which compiles for a target architecture different from the one it is running on. Finally, the compiled binaries are loaded at boot time on the target embedded system. This is the opposite of what we are used to doing on our laptops/desktops: to be able to reprogram the machine as it is running, by writing and compiling our program with *application software* designed to edit text and compile code. Thus, an operating system greatly simplifies interaction with the machine by offering a platform for the user and, at a higher level, by making general-purpose computing possible.

Windows, MacOS, iOS, Android... Most of us are familiar with these operating systems. Besides the platform on which they run, they are all general-purpose and their goal is the same. What really changes among them is the architecture and philosophy in their design. At a macroscopic level they differ in kernel design approach (monolithic kernel vs. hybrid kernel). This is explained later in Section 1.3. At a microscopic level, there is literally not much to see

because the code of most OSes is closed, so it is impossible to see the implementation differences with Linux. This leads us to one of the peculiarities of Linux: it is completely open source and community developed. Besides ethical matters, which are not discussed here, this means that it is possible to study the code and get a full understanding of operating systems. In fact, before Linux, there was no way to see how operating systems work in practice. The only option was to study them from textbooks in order to implement your own kernel, which is exactly what Linus Torvalds did.

As stated earlier, a key component of an OS is "the software intended to manage the hardware resources": this is what we refer as the *kernel*. Dennis Ritchie, among the inventors of Unix and C, also called it the "Operating system proper"[38], which most likely means "The component that is the actual operating system". On the one hand this definition makes sense, because the low level tasks performed by the kernel are essential (and also because it is the most difficult component to develop). But on the other, without application software the kernel is useless. In such scenario, the kernel is loaded at boot, then it initializes and starts running, and then there is nothing but a black screen because there is no other program to start. it is clear that the kernel is not an operating system by itself, but what Dennis meant is that when we think about the core architecture of an OS, we think about the kernel. An engine is indeed useless without the rest of the car, but does that make the other components as important as the engine, where all the complexity resides? Despite the application software being the largest part of the OS; it is within the kernel that the hardest engineering challenges are found, which makes it the most interesting—and difficult—part to understand and analyze.

1.2 A general overview

The kernel's job is to manage hardware resources, which means handling all interactions with the CPUs, the memory hierarchy and the I/O devices. More specifically, the kernel needs to respond to I/O requests, manage memory allocation and decide how the CPU time is shared among the demanding processes. To achieve this, it has access to all resources in the system, which is needed to make the most out of the hardware. Its performance is what makes the difference between a fast or a slow operating system. This critical role requires a protection mechanism to ensure the stability and the security of the whole system. This is achieved by separating kernel code and user application code. In practice, depending on the configuration settings at compile time, what happens is:

1. The kernel binary image is loaded in RAM in a memory area which can start from a low or high address.
2. A pre-defined slice of RAM next to that memory area is reserved to the kernel.
3. The remaining part of the memory is accessible to the user.

These two portions of the address space are called kernel space and user space. The former is a reserved area dedicated to critical system tasks and it is protected from user access, the latter is the area where system utilities and user programs run. This memory partitioning makes sure that kernel and user data do not interfere with each other. Also, it is a security measure to prevent that a malfunctioning or malicious user program may affect the entire system.

1.2.1 System calls

By extension of this design, the interaction with the user space is regulated with a privilege system. Each process can run either in user mode or kernel mode. Processes running in user mode can access privileged kernel functionalities through special gates in a pre-defined and controlled manner. These gates are implemented as functions called *system calls*, which serve as APIs between user and kernel space. When a user process performs a system call

1. it temporarily executes in kernel mode,
2. it performs tasks that require a high privilege, and finally
3. it switches back to low privilege.

This mechanism exploits the availability of hardware functionalities. For example, in the x86 architecture 2 bits in the code selector (`cs`) register indicate the current privilege level (CPL) of the program that is running on the CPU. This value is 0 or 3, respectively, for kernel and user mode and each system call changes this value temporarily.

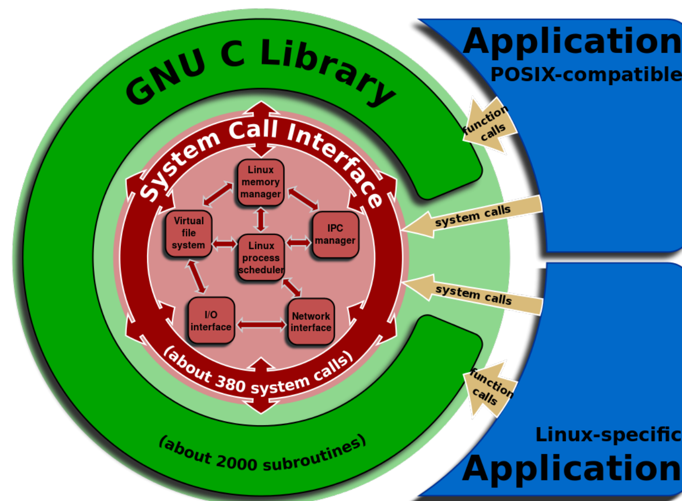

Figure 1.1: Kernel space (in red) and user space (in green and blue)

Very often, useful operations in the system require privileged services provided by the kernel. For example, even an extremely simple shell command such as `echo` performs dozens of system calls, which are reported below as listed by the command `strace -wc echo`

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 29.57    0.000514         514         1           execve
 17.03    0.000296          33         9           mmap
 11.62    0.000202          67         3           brk
  8.52    0.000148          37         4           mprotect
  7.19    0.000125          25         5           close
  6.10    0.000106          35         3           open
  5.93    0.000103          26         4           fstat
  5.58    0.000097          32         3         3 access
  2.82    0.000049          49         1           munmap
  2.24    0.000039          39         1           write
  1.84    0.000032          32         1           read
  1.55    0.000027          27         1           arch_prctl
------ ----------- ----------- --------- --------- ----------------
100.00    0.001738                    36         3 total
```

System calls can be called in user space applications directly, through assembly, or indirectly, by calling wrapper functions from the C standard library (`glibc`), as shown in Figure 1.1.

```
1  // Two different ways of calling open/close through glibc wrapper functions
2  // SYS_open and SYS_close correspond to the syscall numbers
3  int fd = syscall(SYS_open, "example.txt", O_WRONLY);
4  syscall(SYS_close, fd);
5  fd = open("example.txt", O_WRONLY);
6  close(fd);
```

Calling through assembly means filling the right CPU register with the syscall arguments and then using a special assembly instruction. On x86 machines it is required to fill the `EAX` register with the system call number (by a `mov` assembly instruction) and then invoke the interrupt 128 (by the instruction `int 0x80`). Modern processors may use a different one. This is what will happen upon its execution:

1. Interrupt number 128 (=`0x80`) is released. In Linux, it corresponds to a system call interrupt.

2. The process execution is suspended and the control passes to the kernel (kernel mode), which will look up the entry 128 in the *interrupt vector table*. This table simply associates interrupt numbers with their handler: a function that gets executed when the interrupt happens.

3. The corresponding handler is executed: this function copies the syscall number and arguments from the registers onto the kernel stack. It will then look up in the *system call dispatch table* the handler corresponding to the syscall number and call it with the correct arguments like any normal C function, because the arguments are now located on the kernel stack.

4. The system call is finally executed and the return value is stored in a general-purpose data register.

Registers are used to pass the parameters because this way it is easier to get them from user to kernel space. It is intuitive that such a procedure to invoke system calls is architecture dependent. For this reason, `glibc` wrappers are always used: they internally execute the assembly code that we just illustrated and do it differently for each architecture. Calling wrappers is also very safe since it avoids to accidentally fill the wrong registers or miss the right number of arguments. it is important to note that the kernel can protect itself against invalid syscall arguments in registers. This is crucial since, as we saw, syscalls are easily called from user space directly by executing the proper assembly instructions.

1.2.2 A different kind of software

The separation between kernel/user space and the fact that we are working at such low level makes the kernel a very peculiar piece of software. One of its properties is that there is no error checking, this is because the kernel trusts itself: all kernel functions are assumed to be error-free, so the kernel does not need to insert any protection against programming errors[23]. Instead, what the kernel does is to use assertions to check hardware and software consistency; if they fail then the system goes into *kernel panic* and halts. The choice of checking assertion (and possibly going to kernel panic if something went wrong) is that since the kernel controls the system itself, error recovery and error correction is very hard and would take a huge part of the code. Another way of thinking about it, is that there is no meta-kernel that handles kernel errors. Of course programming or hardware errors can (and will) still occur: when this happens the offending process is killed and a memory dump called "*oops*" is created. A typical example of this is when the kernel dereferences a NULL pointer: in user space this would cause a *segmentation fault*, while in the kernel it will generate an oops or in the worst case go directly into panic. After this kind of event, the kernel can no longer be trusted and the best thing to do would be to reboot, because the kernel is in a semi-usable state and it could potentially corrupt memory. Furthermore, a panic in this state is more likely to happen. Possibly, the user experiencing the kernel panic may also inform the kernel maintainers.

Another peculiarity of the kernel is that it uses its own implementation of the functions in the standard C library. For example `printf()` and `malloc()` are implemented as `printk()` and `kmalloc()`. There are different reasons for this choice, one of those is that the C standard library is too big and inefficient for the kernel. Another reason is that implementing your own functions gives

more freedom because they can be customized for their purpose in the kernel. Memory allocation in user or kernel space is very different, so the `kmalloc()` implementation is very specific. For instance, kernel data structures need a contiguous physical memory segment to be allocated, while regular user space allocation does not have this restriction. Furthermore, `printk()` writes its output into the kernel log buffer (that you can read by using the `dmesg` command in user space); this is very different from `printf()` that writes on standard output.

1.2.3 User and kernel stacks

As stated earlier, the memory management is different in kernel/user space. The same applies to the execution. Every process in the system has two stacks, located respectively in user and kernel space, and it will use one of the two while executing in the corresponding privilege mode. x86 CPUs automatically switch stack pointers when privilege mode switches occur, which usually happens for syscalls. The user space stack can potentially be very big, with a very high limit (8MB on my machine, but it can be increased), and even though it is initially small it can allocate more physical memory as it needs it: this mechanism is called "*on-demand paging*". The kernel stack, unlike the user stack, cannot expand itself and it has a fixed size of two pages. Since, 32-bit and 64-bit systems have 4KB and 8KB sized pages, then the kernel stack size is of size 8KB or 16KB, respectively. These two pages must be allocated contiguously, which can cause memory fragmentation for long system uptimes as stacks get deallocated. In other words, it becomes increasingly hard to find two physically contiguous available pages as the OS runs for a long time. For this reason, in the past efforts were made to reduce the stack size to one page, which would eliminate fragmentation, but after many stack overflows the standard settled on two pages.

This leads us to an interesting example of the kernel trusting itself: it makes the strong assumption that the stack will never overflow: **no protection against it is in place**. So what happens if it overflows? First, it will corrupt the `thread_info` data structure, which is the first data that the stack encounters along its path (Figure 1.2). This will make the process nonexistent for the kernel and cause a memory leak. Next, the stack can overflow outside of the address space and silently corrupt whatever kernel data is stored; the best case scenario here would be a kernel panic to prevent any further memory corruption. Another natural question might be "why are kernel stacks so small?" and the answer is simple: first, to use a small amount of kernel memory, and secondly, because of fragmentation. The bigger is your data structure in contiguous physical memory, the more it is hard to allocate. It is expected that any process stays in kernel mode for a small amount of time, so it should use a very small portion of the stack. A consequence of small stacks is that very few recursive functions are used to avoid long call chains and minimize stack usage; the same is true for big static allocations on the stack.

It is important to note that there are special processes called *kernel threads*

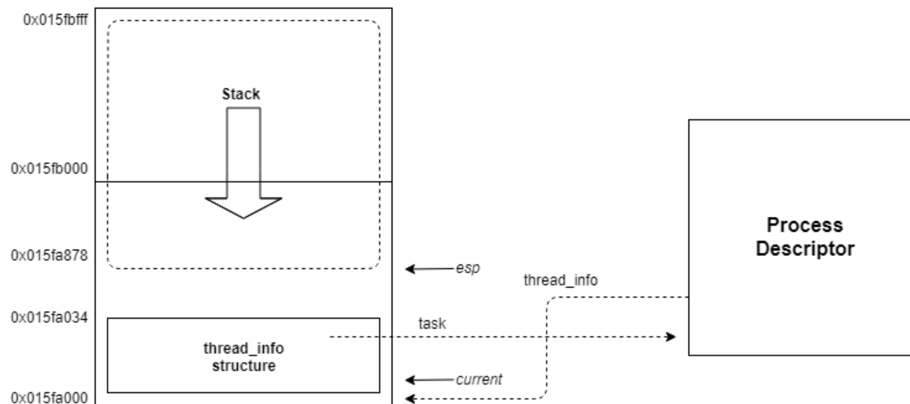

Figure 1.2: The kernel stack inside its small address space of two pages, it grows downward towards low memory.

that do not follow this pattern of kernel/user stack. Kernel threads perform a specific system task, they are created by the kernel and they live exclusively in kernel space, never switching to user mode. Their address space is the whole kernel space and they can use it however they want. Besides this, they are normal and fully schedulable tasks just like the others. An example of a kernel thread is `ksoftirqd`: there is always one for each CPU and their job is to dispatch interrupt requests. As a side note, the name stands for "Kernel Software Interrupt ReQuest Daemon", many kernel threads follow a similar naming convention.

1.2.4 A monolithic design

There are fundamentally different design approaches in kernel development. We can see these as a spectrum, where on one end there is the *monolithic kernel*, and on the other one the *microkernel* (or *µkernel*). The choice depends on how many services are located in kernel space: while in monolithic design every service is in the kernel itself, microkernels strive to reduce as much as possible the code running in kernel space. This is done by moving most services in user space, while keeping only essential primitives in the kernel (Figure 1.3).

These services are implemented as *servers*, and communication between the servers, applications, and the kernel is based on message passing. As in classic client/server approach, applications send requests to the servers, which can in turn request services to the kernel or satisfy the request directly. Because of this design choice, the system relies heavily on *Inter-Process Communication* (IPC), which can be achieved in different ways: in this case, there are actual messages being passed between processes. Even if they are part of the core architecture, the servers are user processes and run in user space just like the other user processes, though they get higher privileges.

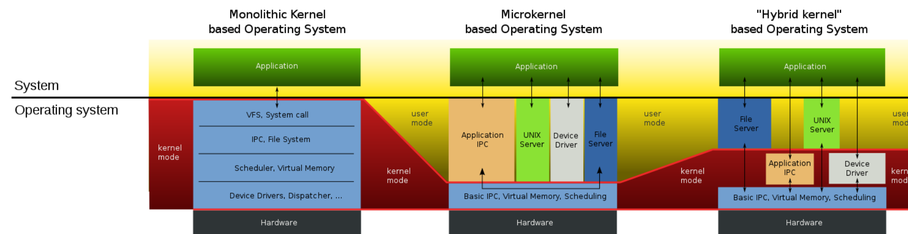

Figure 1.3: The most popular kernel designs and their differences

By reducing the code running in kernel space, there is less risk for bugs. Because the trusted codebase is very small, there is no need to make big assumptions like in monolithic kernels. As stated before, a bug in the kernel can bring down the entire system, but in microkernels bugs are contained. For example, if the networking service crashes, then we can just restart it since it is just a user process; in a monolithic environment, this problem would have crashed the entire system: this is one of the biggest flaws of the monolithic design. A small trusted codebase also means more portability because all the architecture dependent code is concentrated in the small kernel. The actual operating system is built on top of it, so it would be possible to implement it in a more high level language, while only the primitives in the kernel must be ported. Conversely, in a monolithic kernel, many functions must be rewritten for each architecture: in Linux, the folder for architecture dependent code (`arch`) is the second biggest folder and it represents 8% of the code.

Another direct consequence of shifting the code in user space is that microkernels are more easily maintainable. Development is easier because most of the code runs in user space, so the usual restrictions for kernel code are not present: for example, it would be possible to make use of `glibc`. Furthermore, testing can be done without rebooting the system: just stop the service, recompile the code and then start it again. On a monolithic system, not only it is needed to recompile the whole kernel, we must also reboot in order to load the image again. And if this new image does not work, then we must reboot again with the working image. In practice, this is always done in a virtual machine in order to test more efficiently, but it is still a tedious process.

Given all these advantages, why are not microkernels always used? It is mostly because of one deadly flaw: the performance penalty. It is easy to see this if we think that monolithic kernels communicate directly with hardware, while in microkernels most of the operating system does not; essentially, microkernels add an additional layer of abstraction through heavy use of IPC. More precisely, the task of moving in and out of the kernel to move data between applications and servers creates significant overhead. This process results in two major problems:

- A large number of system calls, caused by services frequently needing to use the primitives.

- Many context switches, because each service must be scheduled as a process. In order to pass a message between two services, a full context switch is needed to send and receive.

This last problem is not an issue in a monolithic setting, because kernel functions are executed when any currently running process enters kernel space. Of course, calling a plain function is much less costly than doing a system call or context switch. Furthermore, IPC in monolithic kernels is implemented through shared memory, which is more efficient than IPC with message passing. In Linux, because every functionality is in the kernel, it is a single, big program running in his dedicated address space: this means that every subsystem (scheduling, IPC, networking, memory management. . .) shares the same memory. Paradoxically, all the auxiliary code needed for interfacing and communication can make microkernel-based operating systems larger than monolithic kernels, even though all this code is not in kernel space.

Linux is a monolithic kernel, and because of this design choice, even the device drivers are located in kernel space: in fact, more than 65% of the kernel code is just drivers (in the `driver` directory, the largest folder). This means that while the system is running a huge part of the code is not being used. For this reason, many miniaturized versions of Linux have been distributed: a fully functional—and still monolithic—kernel can fit on a single floppy disk. If we wanted to create just a reduced version, it would not be too hard to remove drivers that are not needed and then recompile the kernel.

A problem of the monolithic design is the natural lack of modularity; microkernels do not have this problem because it is very easy to start/stop drivers running in user space. Monolithic kernels try to achieve the modularity of microkernels by using *kernel modules*: they are simply code that can be inserted/removed from the kernel at runtime. A module can be linked to the running kernel when its functionality is required and unlinked when it is no longer useful: this is quite useful for small embedded systems, to keep running code to a minimum. Modules are often used to add/remove drivers and this approach is much faster than having drivers in user space: since the code runs in kernel space, there is no need to do message passing or communicate with user space at all. it is just like in a microkernel and without performance penalty, but then again, now we are programming in kernel space, which is harder. In the end, it is a choice between ease of development/fault tolerance or performance. Furthermore, modules, unlike the external layers of microkernel operating systems, do not run as a specific process. Instead, they are executed in kernel mode on behalf of the current process, like any other kernel function: this means less switching between processes, so again, better performance. Because of the big flaw of monolithic kernels mentioned earlier, if a driver module does not behave correctly, the system can crash upon module insertion.

Modules are powerful, but cannot always accomplish what a microkernel can. As an example: on Linux, it is not possible to replace the scheduler at runtime. In order to do that, it is needed to have the two different schedulers directly in the core code and switch between them at runtime (This is how it

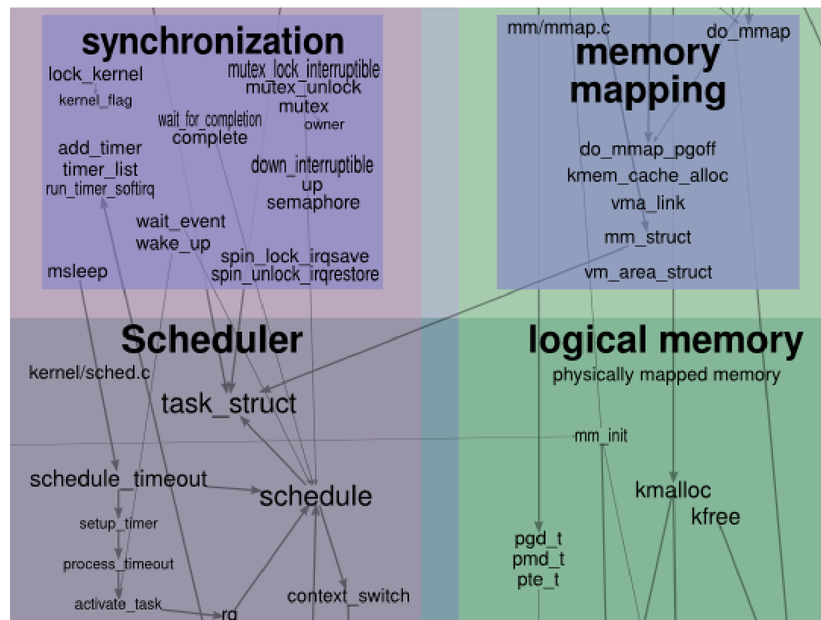

Figure 1.4: A portion of the kernel subsystems map

is actually done in Linux: there are different schedulers already implemented). Modules usually are not used to implement core functions, but are rather seen as extensions of the kernel. This means that it is very difficult to modify policies decided by the kernel through modules, and users must adapt to these policies or modify the code and recompile the whole kernel. Conversely, in microkernels it is easy to change core implementation, since it runs as a service.

Finally, the hybrid design is halfway between monolithic and microkernel, as it tries to take the best side of both approaches: having good performance, but also, to some extent, flexibility and maintainability. In practice, its philosophy is very similar to a monolithic kernel and hybrid kernels have been dismissed by Linus Torvalds as "just marketing"[39]. Notable OSes that use a hybrid kernel are Windows and MacOS.

1.3 Process management

The kernel is divided into subsystems that interact with each other. Figure 1.4 is a zoom into the kernel mechanisms inside the red part of Figure 1.1. The image represents the part of the kernel that will be covered, for the most part we will swing between the scheduling and memory mapping subsystems. The names in the picture are structs, functions or source code files; we will get familiar with most of these as we go on.

1.3.1 Processes and threads

A process is an instance of a running program. Each process has resources associated with it, such as an address space, open files, global variables and code. Each process must have its own address space that only he can access: when a process tries to access a memory location that does not belong to it, a segmentation fault interrupt is generated. A thread is defined as a single flow of execution, it has associated a stack of execution and the set of CPU registers that it uses, most notably the stack pointer and program counter. Each process can have multiple threads, in which case it is a *multi-threaded process*; threads belonging to a process will share resources between each other. The execution aspect of a process is always represented by threads, which means that a process cannot exist without at least one thread associated.

The kernel does not distinguish between processes and threads, so they are treated as the same entity. Because of this, a problem in terminology arises. Next, it is shown how processes and threads are distinguished.

Each process has its own PID (Process IDentifier) and groups of processes are identified by the TGID (Thread Group ID). If a process only has a single thread then its PID is equal to its TGID. If a process is multi-threaded then each *thread* has a different PID, but they will all have the same TGID. Furthermore, there will be a thread in this group called *thread group leader* that will have its PID equal to the TGID, so the TGID field in each thread is just the PID of their leader. Just to add some more confusion, when you call `getpid()` you are actually getting the TGID (the group leader PID, identifying the whole process), and when you call `gettid()` you are getting the PID (which identifies a single thread, not a group). Hence, the PID resembles more a thread identifier. This confusing way of using IDs was implemented to comply with POSIX standards, which require that each thread of a multi-threaded process must have the same id: this is why `getpid()` returns the TGID.

The real difference between threads and processes is that threads share the same address space, while processes do not. By saying that some threads are associated to a same process just means that they are sharing an address space. This enables concurrent programming, enables communication among threads via shared memory, and requires then synchronization methods. As shown in Section 1.3.3, using threads in a program instead of spawning new processes results in much better performance, which is why threads are sometimes called *lightweight processes* or LWP.

```
1  //stack size for cloned child
2  const int STACK_SIZE = 65536;
3  //start and end of stack buffer area
4  char *stack, *stackTop;
5  // ... define child startup function "do_something" ...
6  stack = malloc(STACK_SIZE);
7  stackTop = stack + STACK_SIZE; //stack grows downward
8
9  //spawns a new thread
10 clone(&do_something, stack + STACK_SIZE, CLONE_VM | CLONE_FS | CLONE_FILES |
   ↪  CLONE_SIGHAND, 0);
```

```
11 //spawns a new process, this is the same as using fork()
12 clone(&do_something, stack + STACK_SIZE, SIGCHLD, 0);
```

The system call `clone()` spawns a new child process. it is very similar to `fork()` but it is more versatile because flags can be used to decide how many resources are shared with the new process. `CLONE_VM` (where vm stands for virtual memory) makes the child process run in the same address space as the father, while the other flags clone filesystem information (such as working directory), open files and signal handlers. The flag `SIGCHLD` at line 12 requires that the parent process receives a `SIGCHLD` signal upon the termination of the created child process. Ultimately, the reason why threads and processes are treated as the same entity in Linux, is that processes are just threads that share nothing. In fact, the word *task* is always used inside the kernel instead of process/thread and we will do the same, especially when discussing implementation.

Each task is represented in the kernel with the struct `task_struct`, this is a fairly big structure that can be almost 2KB in size, depending on configuration at compile time. `task_struct` is what is often referred as the *process descriptor* or PCB (*process control block*): every information about a task is stored in here.

```
1  // Code from ./include/linux/sched.h
2  struct task_struct {
3      /* -1 unrunnable, 0 runnable, >0 stopped: */
4      volatile long       state;
5      void                *stack;
6      /* Current CPU: */
7      unsigned int        cpu;
8      // A boolean, "on_runqueue"
9      int                 on_rq;
10     int                 prio;
11     int                 static_prio;
12     int                 normal_prio;
13     int                 exit_state;
14     int                 exit_code;
15     int                 exit_signal;
16     /* The signal sent when the parent dies: */
17     int                 pdeath_signal;
18     pid_t               pid;
19     pid_t               tgid;
20     /* Real parent process: */
21     // The original parent that forked this task
22     struct task_struct __rcu    *real_parent;
23     /* Recipient of SIGCHLD, wait4() reports: */
24     // The current parent, maybe the original one exited
25     struct task_struct __rcu    *parent;
26     // Executable name, usually the command that spawned this task
27     char                 comm[TASK_COMM_LEN];
28     /* Filesystem information: */
29     struct fs_struct           *fs;
30     /* Open file information: */
31     struct files_struct        *files;
32     /*
33      * Children/sibling form the list of natural children:
34      */
```

```
35      struct list_head        children;
36      struct list_head        sibling;
37      struct task_struct      *group_leader;
38      /* PID/PID hash table linkage. */
39      struct pid              *thread_pid;
40      struct hlist_node       pid_links[PIDTYPE_MAX];
41      struct list_head        thread_group;
42      struct list_head        thread_node;
43  };
```

These are some of the most basic fields the struct, most of them are self-explanatory.

The `volatile` keyword asks the compiler not to optimize by caching the storage of this variable. This indicates that the value may change even if the variable does not appear to have been modified. Hence, every time a `volatile` variable is accessed, it needs to be read from the main memory. The opposite of `volatile` is the compiler hint `register`. The fact that the task state is volatile makes sense because it could be unpredictably modified by interrupts: it could be possible that an old value of the variable is read from the cache instead of the actual value.

Let us now focus on the `pid` fields to show how Linux uses pids to find any information and resources of a task. Given a pid, searching linearly through the pids to find the task we are looking for would be very inefficient. Instead, a hash table known as *pid hash table* is used for this purpose. The identifiers in this table are simply the result of hashing a given pid, you can see in figure 1.5 that conflicting entries are simply stored in a list associated with the same id. Because it is a hash table, the kernel can quickly look up the pid and find in $O(1)$ time the corresponding process descriptor. This procedure is, for example, applied when the command `kill [PID]` is launched.

```
1  // Code from ./include/linux/pid.h
2  enum pid_type {
3      PIDTYPE_PID,  //process PID
4      PIDTYPE_TGID, //thread group leader PID
5      PIDTYPE_PGID, //process group leader PID
6      PIDTYPE_SID,  //session leader process PID
7      PIDTYPE_MAX
8  };
```

There are actually four tables, one for each PID type. Each of these tables is an array of `hlist_head`, the head of the chain list, which points to a list of `hlist_node` (see Figure 1.6). These structures are used for non-circular lists. These lists are populated by `struct pid`, and a pointer to this struct is stored inside each process descriptor in the `thread_pid` field. Figure 1.6 shows an example for the TGID class that we discussed earlier. PIDs in the chain list are colliding and are different processes, PIDs in the `pid_list` are threads in the same group, where the leftmost thread in the image is the group leader. Despite the name "`list_head`" inside the pid structure, such a field points to a circular

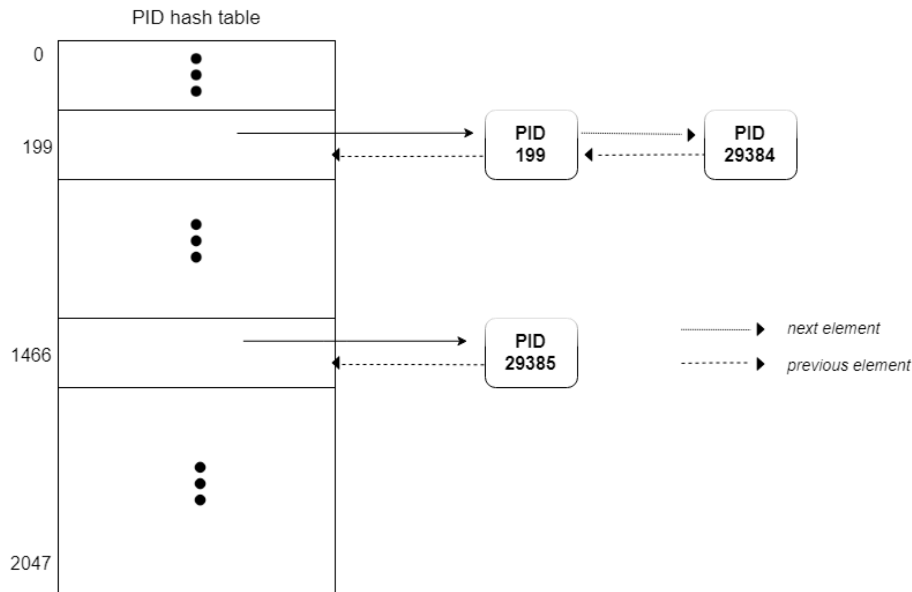

Figure 1.5: Pid hash table, pids 199 and 29384 are both hashed to 199

list, and since it is circular there is really no head structure that points to the first element.

```
// Code from ./include/linux/pid.h
struct pid {
    atomic_t count; // number of references to this PID
    int nr; // PID number
    struct hlist_node pid_chain; // Link to next and previous conflicting
 ↪  entries
    struct list_head pid_list;  // per-PID list
};
```

The implementation of `struct pid` is slightly different from what was presented, with other nested structures and a different linkage to the hash table. In this section, only a small portion of the process descriptor is described.

1.3.2 List implementation

In a classic circular list, the `struct` of the node contains the data and pointers to the next and previous nodes. This implementation is naive and would lead to have a different structure for each data type, or using a void pointer to our data for no reason. Let's see how lists are used in the kernel.

```
struct list_head {
    struct list_head *next, *prev;
};
```

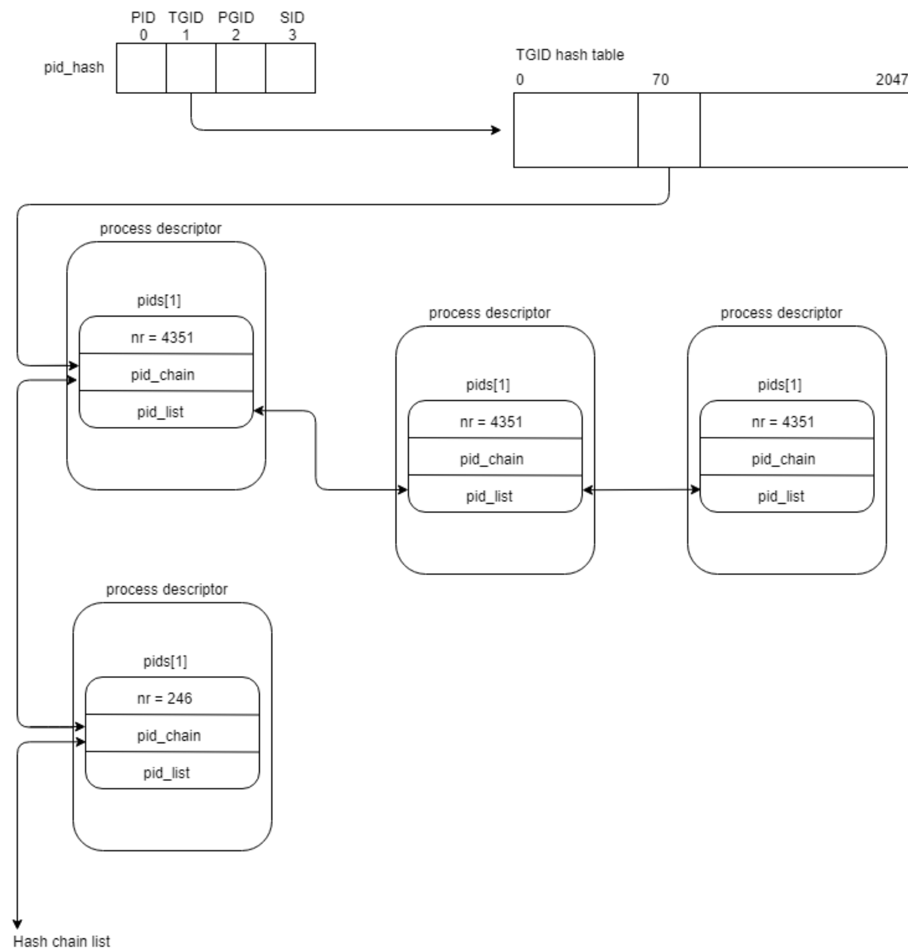

Figure 1.6: Hash table for the TGID pid type

The data is not contained in the list itself, but in another structure that contains the list node (Figure 1.7). For example, Linux keeps a big circular list of every `task_struct` in the system: this is done by embedding `struct list_head tasks;` into `task_struct`. Notice how this is not a pointer to a node: the node is embedded directly into the structure. So how can we get the data we want in the structure without a pointer in the node? The answer is the `container_of()` macro. This macro works with anything, but let's assume that we have a list embedded in the container structure.

```
// ./include/linux/kernel.h
#define container_of(ptr, type, member) ({ \
    void *__mptr = (void *)(ptr); \
    ((type *)(__mptr - offsetof(type, member))); })
// An alias that is used everywhere
```

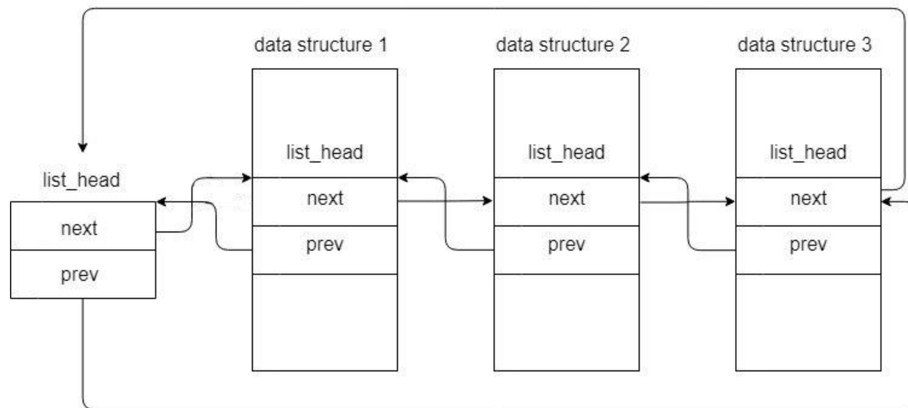

Figure 1.7: A generic doubly linked circular list

```
6 #define list_entry(ptr, type, member) \
7     container_of(ptr, type, member)
```

ptr is the pointer to the list node, type is the container struct, member is the field name of the list node in the container struct. We first cast ptr to a void pointer, then we subtract the offset from the beginning of the container struct of the field we want to get. When we allocate a struct, its field are allocated contiguously in virtual memory in the order that we declared them: this means that by moving the pointer backwards from a field by the right amount, we can end up at the beginning of the container structure. This is how we can get the offset of the specified field in any struct:

```
1 // ./include/linux/stddef.h
2 #define offsetof(TYPE, MEMBER) ((size_t)&((TYPE *)0)->MEMBER)
```

TYPE is the struct we are considering, MEMBER is the name of the field, what it does is:

1. Take the address 0, the first in the address space of the process
2. Cast it to a TYPE pointer
3. Dereference the pointer and take the MEMBER field
4. Take the address of the field and cast it to a size, now it is no longer an address

Essentially, we are pretending that there is the container structure allocated just at the beginning of the address space. This is arguably a bit of a hack, but it is perfectly safe since we are just playing with pointers and never touching actual memory. Indeed, it would be very dangerous to dereference and modify data

from a random pointer in memory. This approach has many advantages, such as being able to have multiple lists associated with the same data. `task_struct`, for instance, contains also the `children` and `sibling` lists among many others. This implementation is also very easy to use and it is oblivious about types.

1.3.3 Scheduling

A system with a single CPU can execute only one process at a time. For this reason, a scheduler for processes is needed. Process scheduling consists in choosing which processes should run in what order, essentially deciding how CPU time is shared among processes. To achieve this, there are many scheduling algorithms such as FCFS (*first come first served*), RR (*round-robin*), EDF (*earliest deadline first*) and SJF (*shortest job first*). Most of the scheduling policies are *preemptive*, which means that at any time the scheduler can arbitrarily decide to interrupt the currently running task and assign the CPU to another process. The use of preemption implies that processes have assigned *timeslices*: they are periods of time in which the process is allowed to run and after which it will be preempted.

FCFS, which is the most basic scheduling algorithm, does not have preemption nor timeslices: every process runs as much as it wants before voluntarily giving up the CPU to the next task in the queue. Round-robin is similar to FCFS because it has a FIFO runqueue; the difference is that it uses a constant timeslice, called *quantum*, assigned to each process: when the quantum expires the process gets preempted and the next task is scheduled.

In a UP (*uniprocessor*) system it is not possible to achieve true parallelism among processes. The only way to do it is to have multiple processors that share a common bus and the central memory: this is known as SMP (*symmetric multiprocessing*). A single processor can also have multiple cores, but each one is treated as a separate processor, so the SMP architecture applies to cores as well. Even on SMP systems, which represent most systems today, there often are more processes than cores. Hence, scheduling is necessary for each processor/core. There are also new problems that arise in SMP, such as *load balancing*: the problem of balancing processes between CPUs so that no CPU goes idle or has an unfair amount of workload. This kind of related problems must also be taken into account by the scheduler.

Every job carried out by the scheduler will eventually lead to a process switch on a given CPU. The kernel has a mechanism to suspend the execution of a process, save its status, and resume another process. This procedure is called *context switch*. Each process has an *execution context*, which includes everything needed for a task to execute, from its stack to the code. While every process can have its own process descriptor, the registers on the CPU must be shared between every process in the system. Every value in any register that a process is using is a subset of the execution context and it is called the *hardware context*. At every context switch the hardware context must be saved and restored, respectively, for the old and the new process. The content of the registers are saved in part in the process descriptor of the preempted process,

and in part on its kernel stack.

The routine that performs a context switch is called—not surprisingly—`context_switch()`, and it is called only in one well-defined point in the kernel: inside the `schedule()` routine, which triggers the scheduler and chooses the next task to schedule. `context_switch()` basically switches the address spaces of the two processes and then calls `__switch_to()`. This last function operates on registers and kernel stacks, so it is one of the most architecture dependent in the whole kernel. This is why, like many other similar routines, there is one version for each architecture supported by Linux in the `arch` folder. Next, the x86 version of the context switch is described.

There are 6 *segmentation registers* that hold *segment selector*, basically the starting address of memory segments in the process address space.

- `cs` *Code segment*, this points to the segment containing instructions of the loaded program, also known as the `.text` section. We mentioned in Section 1.2 that this register also holds 2 bits that describe the current privilege level of the CPU.
- `ss` *Stack segment*, points to the segment containing the stack of execution.
- `ds` *Data segment*, points to the segment containing global variables and constants, also known as the `.data` section.

The other 3, `es`, `fs` and `gs` are general-purpose and do not hold a specific address. There are also general-purpose data registers that hold data used in operations (`ax`, `bx`, `cx`, `dx`) and pointer registers, that hold offsets:

- `ip` *Instruction pointer*, offset to the next instruction. If added to `cs` will be the address of the next instruction to fetch (`cs:ip`).
- `sp` *Stack pointer*, offset to the top of stack. If added to `ss` will be the address of the top of stack (`ss:sp`).
- `bp` *Base pointer*, offset to subroutine parameters on the stack (`ss:bp`).

Let's now see which part of the process descriptor is involved in context switching.

```
1  struct task_struct {
2      // ...
3      /* CPU-specific state of this task: */
4      struct thread_struct    thread;
5  };
```

```
1  struct thread_struct {
2  #ifdef CONFIG_X86_32
3      unsigned long    sp0;
4  #endif
5      unsigned long    sp;
6  #ifdef CONFIG_X86_32
```

```
7       unsigned long    sysenter_cs;
8   #else
9       unsigned short   es;
10      unsigned short   ds;
11      unsigned short   fsindex;
12      unsigned short   gsindex;
13  #endif
14      // ...
15      /* Floating point and extended processor state */
16      struct fpu       fpu;
17  };
```

This struct is obviously very architecture dependent, its purpose is to save the hardware context before the context switch. You can see that even if it is specific to x86 it can still change depending on whether the architecture is 32 or 64 bits. You can also notice that only a small part of the hardware context gets saved in the process descriptor: the kernel stack pointer, general-purpose segmentation registers, data segment and the floating point registers. In older versions of the kernel most of the registers were stored here. Let's see in detail what happens when the kernel switches from process A to process B. There are actually two different mechanisms in this procedure: the entry/exit mechanism (user/kernel stack switch) and the context switch.

1. Process A enters kernel mode, so it will switch from its user stack to its kernel stack, in other words: it saves its **user** hardware context in the kernel stack. It does so by pushing its **user mode** stack (`ss:sp`), instruction pointer (`cs:ip`) and data registers onto the kernel stack, then all CPU registers are switched to use the kernel stack.

2. When in kernel context, process A invokes `schedule()` which will eventually do `context_switch()`.

3. Process A saves its hardware context:

   (a) It pushes most of its register values onto the kernel stack by a series of `mov` assembly operations.

   (b) It saves the value of the stack pointer (which is pointing to the **kernel** stack) into its `task_struct->thread.sp`.

   (c) Other registers such as the floating point registers are saved in the `thread` field of `task_struct`.

4. Process A loads a previously saved stack pointer from process B's `task_struct->thread.sp`, also loads the other saved registers

5. Address spaces are switched.

6. Using the loaded stack pointer, process B moves its previously saved registers from its kernel stack into the registers. This is done by a series of `pop [register]` assembly operations. Process B's state is now completely restored.

7. process B exits kernel mode and restores its **user** context. This is accomplished by loading previously saved registers from the kernel stack: its **user mode** stack (`ss:sp`), instruction pointer (`cs:ip`) and data registers. Process B is now in user context.

To understand scheduling mechanisms in the next sections it is important to highlight something in step 2, when the scheduler gets called by a process running in kernel mode. It may be intuitive to think of the scheduler as some kernel thread that is permanently running in kernel mode, but that is not the case. **The scheduler does not run as a separate thread, it always runs in the context of the current thread.** This means that any process in the system that goes from/to kernel mode can potentially execute the scheduler himself, using its own kernel stack. The simplest case is when a process voluntarily gives up the CPU by going into a sleep state, in which case it subsequently executes `schedule()` in kernel mode (it would have switched already to kernel mode when sleeping). Another thing to highlight is how the user hardware context has nothing to do with context switch, this is because it always gets saved/restored on the kernel stack when entering/leaving the kernel. An implication of this fact is that context switches always happens in kernel mode, which is expected since it is a core system task.

It is important to understand that a context switch generates significant overhead and, in fact, most of the scheduling overhead comes from context switching. It is caused by the need to switch address spaces and by the fact that context switching is not cache friendly. This is the reason why a context switch between threads (LWP) is almost inexpensive compared to context switching different processes: step 5 in the procedure is skipped because threads share an address space, so there is no need to switch it (again, this is why they are **lightweight** processes).

1.3.4 Tasks lifetime

Tasks have a life cycle: a new child process task is created every time a task uses fork-like system calls. As shown in Section 1.3.1, once a process is created some resources are inherited from the father, depending on the `clone()` flags, while `fork()` will duplicate the calling process. There are some resources that will always be inherited and there is no reason to duplicate, such as the executable object code (the `.text` memory segment in Linux). The new process will be in the runnable state and ready to be scheduled. When the process needs to wait for a particular resource, it goes into a sleep state; it will then become runnable again when the resource is available, or after a pre-defined time when the syscall `sleep()` is used. A process can also go from running to runnable: this happens if the process is preempted or if it gives up the CPU voluntarily. This last case happens, for instance, if the process needs to do I/O operations for which it does not need the CPU. This way no processor time is wasted and another task is scheduled.

A process terminates by executing `exit()` or when it receives a signal (in-

cluding SIGHUP, SIGINT, SIGKILL, SIGTERM, and others) from other processes which have the privileges to do so. Upon exit, its *exit state* will initially be set to the *zombie* state. A zombie process is a process that terminated, but its process descriptor and entry in the pid hash table are still present in memory and accessible (for example, by `ps -aux`). Tasks' resources are not deallocated immediately because the parent process may want to access some of this information, most likely the *exit status*, or may want to synchronize with the child process termination via `wait()` or `waitpid()`. This is actually a relevant resource leak because `task_struct` is almost 2KB in size. Hence, if there are many zombie processes then a big portion of memory is simply wasted until the parent process executes a `wait()/waitpid()`. More in details, a `task_struct` plus its kernel stack consumes around 10KB of low kernel memory, that is `THREAD_SIZE + sizeof(struct task_struct)`, assuming that kernel stacks are 8KB in size (`thread_info` and pidhash entry are too small to be relevant).

After terminating and sending a signal to the parent, a task will remain zombie until its parent performs a `wait()`, upon which the parent gets information about the terminated child. Subsequently, `release_task()` is executed and the last data structures from the descriptor get detached. `detach_pid()` is called twice to clear the entry in both the PID and TGID hash tables, then `task_struct` is finally deallocated. Zombie processes are impossible to kill externally: they can not receive signals as they no longer exists, so a wait by the parent is the only way to clean the memory occupied by the zombie data structure. Suppose that the parent of a zombie process exits without waiting: the child will be an orphan process so it will be become a child of `init`. Luckily, the ancestor process (`init`) has a routine that waits periodically to reap possible zombie processes; so the child process will simply be waited by `init` and get cleared. This mechanism ensures that memory won't be cluttered by zombies and leaves the pid table in a consistent state.

States and exit states of a process are defined in `include/linux/sched.h` as following.

```
1  /* Used in tsk->state: */
2  #define TASK_RUNNING          0x0000
3  #define TASK_INTERRUPTIBLE    0x0001
4  #define TASK_UNINTERRUPTIBLE 0x0002
5  #define __TASK_STOPPED        0x0004
6  #define __TASK_TRACED         0x0008
7  /* Used in tsk->exit_state: */
8  #define EXIT_DEAD             0x0010
9  #define EXIT_ZOMBIE           0x0020
10 #define EXIT_TRACE  (EXIT_ZOMBIE | EXIT_DEAD)
```

- `TASK_RUNNING` is either a process that is ready to be run (in which case it is more like "runnable") or that is actually running.
- `TASK_INTERRUPTIBLE` and `TASK_UNINTERRUPTIBLE` are both states in which a task is sleeping, waiting for some condition to be true. The former allows

a process to be woken up by signals, the latter does not: an uninterruptible task will ignore any signal and will only wake up on his condition. This distinction is the reason why, as we will see later, the routine that wakes up tasks is called `try_to_wake_up()`.

- `__TASK_TRACED` means that another process is tracing this one, usually a debugger such as `ftrace`.
- A task in `__TASK_STOPPED` is not running and cannot be scheduled: this happens upon stop signals or any signal from a tracing process.

The values associated to these states are defined like this so that they can be used for bitmasks, which is the standard way to handle flags. Each flag is a power of 2 (in hexadecimal) so flags can be combined with bitwise operator `|` or be tested with `&`. For example, checking if a task is sleeping can be easily done like this:

```
1  if(tsk->state & (TASK_INTERRUPTIBLE | TASK_UNINTERRUPTIBLE))
2      printk("task %d is waiting for something", tsk->pid);
```

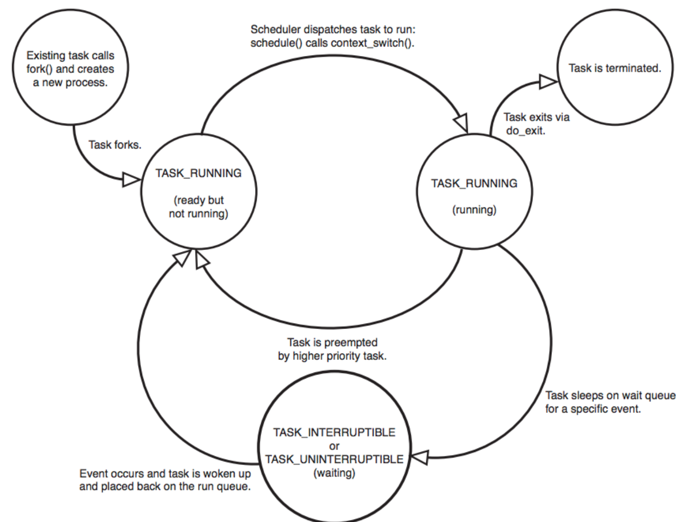

Figure 1.8: State machine of task states

Chapter 2

Tracing events in Linux

2.1 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 [33]:

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 [26].

2.2 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.

Another powerful tool that enable both kernel and userspace tracing is *systemtap*, that will be discussed in Chapter 4.

2.2.1 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* [16].

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 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.

2.2.2 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.

2.3 Tracing with ftrace

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* [10], *eBPF* [1], *sysdig* [17], *LTTng* [4]).

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.

2.3.1 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 2.1: 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

```
lorenzo@localhost:/proc> ls
1      1147  1344  1492  20    29   356   382   46    705   98             interrupts     partitions
10     1148  135   1494  21    3    357   383   465   706   99             iomem          sched_debug
100    116   136   15    22    30   358   384   47    717   acpi           ioports        schedstat
1006   117   1361  1510  2238  31   359   385   48    718   asound         irq            scsi
101    1174  1362  1512  2239  310  36    386   49    723   bootconfig     kallsyms       self
10187  118   1363  1523  2247  32   360   387   50    752   buddyinfo      kcore          slabinfo
102    12    1387  1552  2278  322  361   3873  510   753   bus            keys           softirqs
10204  1202  1391  1589  23    34   362   388   544   781   cgroups        key-users      stat
103    1205  1393  16    2305  343  363   389   545   7834  cmdline        kmsg           swaps
104    1245  14    1602  2361  344  364   39    546   7852  config.gz      kpagecgroup    sys
10442  1247  1401  1606  24    345  365   390   547   790   consoles       kpagecount     sysrq-trigger
106    1248  1413  168   2418  346  37    391   548   791   cpuinfo        kpageflags     sysvipc
107    1260  1418  17    2444  347  375   392   549   797   crypto         latency_stats  thread-self
108    1269  1423  179   2501  348  376   393   55    799   devices        loadavg        timer_list
109    127   1433  18    2532  349  3765  4     550   815   diskstats      locks          tty
11     128   1435  180   2554  35   377   40    551   823   dma            meminfo        uptime
110    129   1437  181   2595  350  378   41    552   9     driver         misc           version
111    1293  1440  182   26    351  379   42    6     96    dynamic_debug  modules        vmallocinfo
112    1298  1442  183   27    352  38    43    602   962   execdomains    mounts         vmstat
113    13    1452  1913  28    353  380   44    685   9674  fb             mtrr           zoneinfo
114    134   1486  1930  2806  354  3806  45    6858  97    filesystems    net
1143   1343  1489  2     287   355  381   456   704   972   fs             pagetypeinfo
```

Figure 2.1: The procfs special filesystem

be tedious: this is why utilities like `ps` are convenient front-ends for the user.

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 2.2 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.

```
localhost:/sys/kernel/debug/tracing # ls
README                     function_profile_enabled  set_ftrace_filter          trace_marker
available_events           instances                 set_ftrace_notrace         trace_marker_raw
available_filter_functions kprobe_events             set_ftrace_notrace_pid     trace_options
available_tracers          kprobe_profile            set_ftrace_pid             trace_pipe
buffer_percent             max_graph_depth           set_graph_function         trace_stat
buffer_size_kb             options                   set_graph_notrace          tracing_cpumask
buffer_total_size_kb       per_cpu                   snapshot                   tracing_max_latency
current_tracer             printk_formats            stack_max_size             tracing_on
dyn_ftrace_total_info      saved_cmdlines            stack_trace                tracing_thresh
dynamic_events             saved_cmdlines_size       stack_trace_filter         uprobe_events
enabled_functions          saved_tgids               synthetic_events           uprobe_profile
error_log                  set_event                 timestamp_mode
events                     set_event_notrace_pid     trace
free_buffer                set_event_pid             trace_clock
```

Figure 2.2: 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 2.3, 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

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

Figure 2.3: Types of tracers on by distribution (OpenSUSE Tumbleweed)

analyze other methods like `trace-cmd` (Section 2.3.4) and `kernelshark`.

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

2.3.2 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.

2.3.3 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 2.2 that there is a directory called "events". It contains a folder for each *event subsystem* (as shown in Figure 2.4). Now let us focus on `kvm` events. Figure 2.5 shows its contents: there is a folder for each event, containing information about it and a switch to enable/disable it2.6.

```
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 2.4: 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 2.5: Every event associated with kvm

```
localhost:/sys/kernel/debug/tracing/events/kvm/kvm_entry # ls
enable  filter  format  hist  id  trigger
```

Figure 2.6: 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.

2.3.4 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 [19].

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` 2.7 and `report` 2.8. 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 2.7 shows the output of `trace-cmd record`). After the recording phase, we can read the `trace.dat` file by the `report` command (Figure 2.8 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 2.7: 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 2.8: Output of the command *trace-cmd report*

Chapter 3

The Intel Skylake family of processors

3.1 Introduction

In this chapter we are going to discuss various trait of Intel's processors regarding power management, focusing on architectures subsequent to Skylake. This is because from Skylake onward, Intel introduced the power management technology of interest, known as Intel's Speed Shift or Hardware-Controlled Performance State (HWP), presented in Section 3.4. Nevertheless, Sections 3.2 and 3.3 describe some useful concepts that also applies to previous generations.

Skylake (SKL) is Intel's successor to Broadwell, a 14 nm process microarchitecture that exists in two configurations, targeted to different purposes [21]:

Server for enthusiasts and servers

Client for mainstream workstations, desktops, and mobile devices

3.2 Performance states

Performance states, or more briefly p-states, are responsible of dynamically changing the frequency and the voltage of the system. These states are extremely useful when idling or when full CPU power is not required. Lowering the voltage will in fact decrease the power consumption and increase the time taken to complete a task, which is often acceptable. A quick raise in voltage is instead desirable when the workload requires low latency and great computational power. P-states represents the biggest variable in power and performance contexts, to the point that their impact is greater then all other power management features combined.

MSR	Address	Bits	Description
IA32_APERF	0xE8	63:0	Always running measure of time. Increments when processor is active at the current operating frequency of the part.
IA32_MPERF	0xE7	63:0	Always running measure of time. Increments when processor is active at the CPU base frequency of the part.
PPERF	0x64E	63:0	Productive performance count, which provides a quantitative metric to software of hardware's view of workload scalability.

Table 3.1: MSRs involved in DVFS, performance counters

3.3 MSR

Since their relevance in DVFS, this section introduces the most important Model Specific Registers (MSR), whose usage is related to p-states and will be explained in section 3.4.1. MSRs are x86 instruction set control registers, responsible for OS-relevant tweaks such as debugging, program execution tracing, computer performance monitoring, and toggling certain CPU features [5]. Here we will analyze only the ones of interest. It is worth noting that the prefix of the architectural MSRs, that is "IA32_", is often omitted since they all begin with it. Architectural MSRs are a subsets of MSRs carried from Intel Architecture 32 bits (IA32), whose name is left so for historical reasons. During our analysis we are also going to elide their names.

The first three MSRs we are going to see are the performance counters shown in Table 3.1: `APERF`, `MPERF` and `PPERF` (the only non architectural one of three).

Values of these registers are not relevant by themselves, but they become so if compared to each other, providing an hardware's feedback mechanism to software. For example, the average frequency of a processor is obtained by calculating the ratio of `APERF` and `MPERF` and multiplying it by the base frequency. This ratio also indicates whether the CPU is going turbo or not by checking if it is greater than 1 (turbo on) or less than 1 (turbo off). Another useful parameter is the ratio between `PPERF`, that can be seen as a counter of non stalled cycles, and `APERF`. That indicates if the workload is scalable.

`HWP_REQUEST` is considerably more complex, due to bits range significance, and is therefore shown in Table 3.2.

One more useful MSR is `HWP_REQUEST_PKG`, that, as one would expect, is the equivalent of `HWP_REQUEST` but for the whole package, at the sole exception of bits 63:42 that are reserved, and is thus omitted.

Table 3.3 shows, with less detail, other registers not directly related to DVFS, but still of significance within the general framework of these decisions, and therefore worth mentioning.

Table 3.2: IA32_HWP_REQUEST register, address 0x774

Bits	Description
7:0	Minimum_Performance: minimum performance hint to achieve the required quality of service
15:8	Maximum_Performance: maximum performance that is expected to be supplied by HWP
23:16	Desired_Performance: if 0 is hardware autonomous, otherwise conveys an explicit performance request
31:24	Energy_Performance_Preference: from 0 (max performance) to 0xFF (max power performance). It influences the rate of performance increase and decrease
41:32	Activity_Window: moving workload history observation window for performance optimization. If 0 is hardware determined
42	Package_Control: derive bits 41:0 from HWP_REQUEST_PKG
58:43	reserved, all 0
59	Activity_Window Valid: when set, derive bits 41:32 from HWP_REQUEST_PKG even if bit 42 is set
60	EPP Valid: when set, derive bits 31:24 from HWP_REQUEST_PKG even if bit 42 is set
61	Desired Valid: when set, derive bits 23:16 from HWP_REQUEST_PKG even if bit 42 is set
62	Maximum Valid: when set, derive bits 15:8 from HWP_REQUEST_PKG even if bit 42 is set
63	Minimum Valid: when set, derive bits 7:0 from HWP_REQUEST_PKG even if bit 42 is set

MSR	Address	Description
IA32_TIME_STAMP_COUNTER	0x10	Always running measure of logical processor time. Increments when the processor is active or idle at the CPU base frequency of the part
IA32_THERM_STATUS	0x19C	Contains status information about the processor's thermal sensor and automatic thermal monitoring facilities. Also exists for the package at address 0x1B1, IA32_PACKAGE_THERM_STATUS
TEMPERATURE_TARGET	0x1A2	Contains thermal information complementary to IA32_THERM_STATUS, like maximum temperature. It is package scoped

Table 3.3: Auxiliary MSRs involved in DVFS

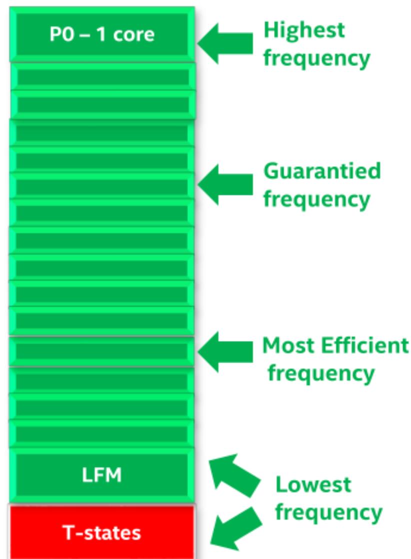

Figure 3.1: P-states schema. Image borrowed from IDF15 [32]

3.4 Hardware-Controlled Performance State

As mentioned before, p-states are the core of power management, and deciding whose is responsibility has been debated for years. Historically has always been software's, since its unique information (process that are running, their priority and the dependency between each other, etc...) are not negligible benefits. However, Intel decided to migrate these decisions to hardware from the 6th generation, introducing Hardware-Controlled Performance State. This change has many advantages:

- less overhead in monitoring/calculating
- faster changes detection
- more hardware related info, like power, thermal, available resources

Before continuing we need to understand what is the real task to get done when choosing the "right" performance. P-states range from P0 to Pn, where P0 is the highest turbo frequency, and Pn is the lowest one (LFM in Figure 3.1); the number of p-states in this interval and their values are processor dependent.

If we imagine to have a task that needs to get done, we have two factors to consider:

1. the energy that the system needs to complete the task (green in Figure 3.2); low if this happens in a short amount of time, raises if in poor performances. It can be approximated at $\sim\frac{1}{f}$

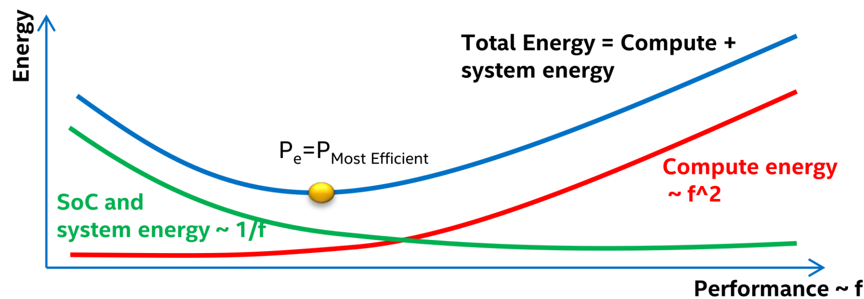

Figure 3.2: Most efficient P-state. Image borrowed from IDF15 [32]

2. the energy that the computation needs to achieve the task (red in Figure 3.2). Raises at higher performances. It can be approximated at $\sim f^2$

The total energy consumed is the sum of the compute energy and the system energy. This sum implies that there is an optimal point (yellow P_e in Figure 3.2), where this value is the minimum.

The goal of HWP is to always select P_e. HWP does so every 1ms, utilizing an algorithm known as EARtH (*Energy Aware Race to Halt* [28]).

Going back to our main problem, we still need to define what is the role of software in such mechanism. In voltage-frequency control, the voltage and clocks that drive circuits are increased or decreased in response to a workload. The operating system requests specific P-states based on the current workload. The processor may accept or reject the request and set the P-state based on its own state [8].

Despite its name, this system aims thus at a cooperation between hardware and software to provide p-states. HWP gives in fact software the ability to supply a target frequency range to operate within along with performance guidance hints. This allows software to use its unique information to provide guidance and for hardware to optimize the selection of p-states within those software provided constraints. If software has no unique information to provide, hardware has the ability to autonomously select p-states.

This collaborative environment is possible thanks to an interface between the two worlds, that ideally would join the natural repsonsiveness of hardware with the software "picture" taken from the operating system leading to smarter DVFS decisions.

3.4.1 HWP software interface

Software utilizes writes to MSRs rather than executing instructions to handle p-state transitions. Typically, the operating system controls HWP operation for each logical processor via the writing of control hints to the IA32_HWP_REQUEST MSR. It can control HWP by writing both IA32_HWP_REQUEST and IA32_HWP_REQUEST_PKG MSRs, taking advan-

tage of the "valid" bits, 63:59, shown in Table 3.2. The OS may override HWP's autonomous selection of performance state with a specific performance target by setting the Desired_Performance field to a non-zero value, however, the effective frequency delivered is subject to the result of energy efficiency and performance optimizations, which are influenced by the Energy Performance Preference field. Software may disable all hardware optimizations by setting Minimum_Performance = Maximum_Performance.

For the user, writing to MSRs is not however much intuitive nor easy. For this purpose Intel provides an utility named msr-tools [6], that allows to read (`rdmsr`) and write (`wrmsr`) MSRs. Its usage is straightforward and covers almost any use case. It is useful to look at their help function, `rdmsr -h` and `rdmsr -h`, that shows a comprehensible list of available features:

```
# rdmsr
Usage: rdmsr [options] regno
  --help         -h  Print this help
  --version      -V  Print current version
  --hexadecimal  -x  Hexadecimal output (lower case)
  --capital-hex  -X  Hexadecimal output (upper case)
  --decimal      -d  Signed decimal output
  --unsigned     -u  Unsigned decimal output
  --octal        -o  Octal output
  --c-language   -c  Format output as a C language constant
  --zero-pad     -0  Output leading zeroes
  --raw          -r  Raw binary output
  --all          -a  all processors
  --processor #  -p  Select processor number (default 0)
  --bitfield h:l -f  Output bits [h:l] only

# wrmsr
Usage: wrmsr [options] regno value...
  --help         -h  Print this help
  --version      -V  Print current version
  --all          -a  all processors
  --processor #  -p  Select processor number (default 0)
```

Below an example that reads the EPP field from HWP_REQUEST:

```
1  $ rdmsr -f 31:24 0x774
2  $ 80
```

The only thing that is not treated is the write of specific bits. This problem will be the subject of Section 5.2.3.

Chapter 4

The SystemTap tool

4.1 Introduction

When it comes to tracing, having a performing tool is essential: remaining low on resources allows to avoid interference with the running system without a noticeable footprint, and this is preferable even at the expense of ease of use.

"SystemTap provides free software (GPL) infrastructure to simplify the gathering of information about the running Linux system. This assists diagnosis of a performance or functional problem. SystemTap eliminates the need for the developer to go through the tedious and disruptive instrument, recompile, install, and reboot sequence that may be otherwise required to collect data. SystemTap provides a simple command line interface and scripting language for writing instrumentation for a live running kernel plus user-space applications. [...] Among other tracing/probing tools, SystemTap is the tool of choice for complex tasks that may require live analysis, programmable on-line response, and whole-system symbolic access. SystemTap can also handle simple tracing jobs" [18].

In the case of SystemTap, there is not in fact the need of a convoluted syntax in order to have a reasonable expressive power, since it has almost no overhead while still maintaining a lean syntax that resembles `C`. This uncompromising characterization makes it a suitable choice when dealing with more complex and dynamic scenarios. In this chapter we are going to introduce its basics, focusing on the features used in the experiments illustrated in Chapter 5.

For a deeper understanding it is especially suggested to look into the following man pages:

stap the front-end to SystemTap

staprun the back-end of SystemTap

stapprobes supported probe points (see Section 4.2.3)

stapfuncs supported functions

4.2 Language overview

SystemTap is a tracing/scripting/event-action programming language, written in `C` and `C++` that was firstly released in 2005. While the first two definitions are intuitive (tracing and scripting), the last one deserves further explanation. Its event-action design is one of the key features of this tool: the language allows the link of a point in the source code (or of an event in the kernel) to an handler, generally a function, that is executed synchronously and that can contain arbitrary portion of code; having that power enables fine-grained tracing, corresponding to basically any event that needs an analysis in both kernel and user space. Despite being stable software, it is still in development (many features are still being added), and among its contributors appear corporations like Red Hat, IBM, Oracle and Intel. The language is strictly typed, expressive, declaration free, procedural, prototyping-friendly, and inspired by `awk` and `C`.

Since its similarities with `C` and its intuitive nature, we are going to discuss only its peculiarities that differentiates with it substantially or that simply are unique to the language.

4.2.1 Types of scripts

There are two types of scripts in SystemTap:

probe script regular programs that identifies probe points and associated handlers

tapset scripts collection of probes and functions organized in libraries, can not be run directly

Example of probe script:

```
1 // will run only once, hence the name
2 probe oneshot {
3     printf("hello world\n")
4 }
```

4.2.2 Built-in functions

In addition to trivial functions like the formatting ones (`printf` ecc. . .), SystemTap provides useful built-in that helps the user with recurrent tasks.

```
1 tid()              // the id of the current thread
2 pid()              // the process (task group) id of the current thread
3 uid()              // the id of the current user
4 execname()         // the name of the current process
5 cpu()              // the current cpu number
6 task_cpu(TASK)     // the scheduled cpu of the task
7 pid2task(PID)      // the task struct of the given process id
8 gettimeofday_s()   // number of seconds since epoch
9 get_cycles()       // snapshot of hardware cycle counter
```

```
10  pp()                 // description of the probe point being currently handled
11  ppfunc()             // the function name in which this probe was placed
12  print_backtrace()    // print a kernel backtrace
13  print_ubacktrace()   // print a user-space backtrace
14  target()             // print target PID, if set
```

The code above shows some those functions and explains in the comments their return values, for a full list run `man -k "function::"`.

4.2.3 Probe points

Probe points identify an event that can trigger a SystemTap's event handler. Its syntax is a dotted-symbol sequence *à la* Domain Name System (DNS), parameterized like a function whose arguments are strings, numbers and wildcards. A probe point needs/can (depends on which one is being used):

- a prefix that identifies the family of the target (kernel, scheduler, process, module, etc. . .)
- a suffix that further qualifies the exact point to probe (return, maxactive(n), statement(0xXXXX), label(s), etc. . .)

```
1  // examples of probes
2  probe scheduler.ctxswitch {...}
3  probe process(my_program_pid).end {...}
4  probe process("/bin/bash").begin {...}
5  probe syscall.*
```

SystemTap offers a huge variety of probe points (some of which are contained in tapsets), regarding kernel and user space. Here we will only look on into some of them, since at SystemTap version 4.5/0.183 there are ∼300 types of different probes! In order to see a comprehensive list of available probes and tapsets run: `man -k "probe::"` and `man -k "tapset::`", respectively.

Basic probes

The first two probes we are going to look are strongly related to each other: `begin` and `end`, two probes of the *special* family. The former is the second portion of code that get executed when SystemTap is run (the first one is the initialization of global variables), the latter is the last one and get executed when the script ends and the `exit()` function is called. These are useful for preparing the environment and for analyzing what has been collected, respectively.

Another helpful probe family is the `timer`. Having a timer that runs arbitrary portion of code at regular interval (s, ms, ns, etc. . .) has obviously many potential use cases.

Below an example that takes advantages of the explained probes.

```
global count

probe begin {
    printf("The script is starting...\ncount equals %d\n", count)
}

probe end {
    printf("The script is ending...\ncount equals %d\n", count)
}

probe timer.s(1){
    count ++
}
```

This simple example counts the number of second elapsed between its launch and the time to whom the user stops it, e.g. with `^-C`. A run will print out:

```
The script is starting...
count equals 0
^CThe script is ending...
count equals 3
```

More complex and useful scripts can be found at SystemTap's website [18].

4.2.4 Target variables

Apart from "usual" variables, there is one unique type of variables, known as target or context variable (since their meaning is *contextual* to the point being probed). These variables are preceded with a "$" and are available for certain kernel probes and allow access to variables defined in the source code at that location. Their availability can be checked with:

```
stap -L 'family.probe("probename_or_wildcard")'
```

For example the output of `stap -L 'kernel.trace("*sched*")'`, is:

```
kernel.trace("btrfs:btrfs_ordered_sched") $work:struct btrfs_work const*
kernel.trace("btrfs:btrfs_work_sched") $work:struct btrfs_work const*
kernel.trace("irq_vectors:reschedule_entry") $vector:int
kernel.trace("irq_vectors:reschedule_exit") $vector:int
```

Context variables can also be printed inside a probe by accessing the variable `$$vars`.

In addition to those variables, some probes and functions, not necessarily kernel's one, may contain useful additional values. These can be shown by looking at individual man pages, e.g.: `man 'probe::scheduler.ctxswitch(3stap)'`.

4.2.5 Embedded C and guru mode

SystemTap provides an advanced mode called *guru mode*, where it is possible to bypass code and data safety checks and in addition to that, to embed standard `C` code. `C` code needs to be wrapped between "%{" and "}%" and can be extremely useful for accessing `#include` instructions. It is possible to embed functions' body, single expressions, top level instructions and pragma comments (compiler optimizations).

The following code snippet shows all the four options in use.

```
%{
#include <linux/in.h>
#include <linux/ip.h>
%} // top level

/* Reads the char value stored at a given address: */
function __read_char:long(addr:long) %{ /* pure */ // pragma comment
    STAP_RETURN(kderef(sizeof(char), STAP_ARG_addr));
    CATCH_DEREF_FAULT ();
%} // function body

/* Determines whether an IP packet is TCP, based on the iphdr: */
function is_tcp_packet:long(iphdr) {
    protocol = @cast(iphdr, "iphdr")->protocol
    return (protocol == %{ IPPROTO_TCP %}) // expression
}
```

For enabling *guru mode* see Section 4.3.

4.2.6 Statistics and aggregates

SystemTap has built-in functionalities to quickly gather and manipulate numerical values in large volume: statistics functions and aggregates. Aggregate instances are used to collect this data, they operate without exclusive locks, and store only aggregated stream statistics. Extracting functions perform statistics computations when called.

The usage of aggregates is fairly straightforward:

```
timestamps <<< gettimeofday_s() // simple array
reads[execname()] <<< count     // associative array
```

Below a comprehensive listing of available functions, where s is the aggregates with whom the aggregations are calculated:

@count(s) returns the number of samples in s

@sum(s) returns the sum of all samples in s

@min(s) returns the minimum of all samples in s

@max(s) returns the maximum of all samples in s

@avg(s) returns the average of all samples in s

@hist_linear(s, L, H, W) returns a visual representation of a linear histogram of s, where L and H represent the lower and upper end of a range of values and W represents the width (or size) of each bucket within the range

@hist_log(s) returns a visual representation of a base-2 logarithmic histogram

4.3 Command line flags and arguments

The SystemTap tool has many command line options, here we will analyze only the one of interests of the `stap` executable.

-g enables guru mode, explained in Section 4.2.5

-i interactively build the script

-t collect timing information on the number of times probe executes and average amount of time spent in each probe-point. Also shows the derivation for each probe-point

-T NUM exit the script after NUM seconds

-o NAME send standard output to file NAME

-s NUM use NUM megabyte buffers for kernel-to-user data transfer

-D NAME=VALUE add the given `C` preprocessor directive to the module `Makefile`

–suppress-time-limits suppress time's related checks.This option requires guru mode

-x PID sets target() to PID

-c CMD sets target() to PID, start the probes, run CMD, and exit when CMD finishes

-k keep the temporary directory after all processing. This may be useful in order to examine the generated C code, or to reuse the compiled kernel object

-v increase verbosity of the session (see Section 4.4 for more details)

For the meaning of function `target()`, from flags `-x` and `-c`, see Section 4.2.2. These options are extremely relevant when filtering on a specific process. Usage of flags `-k` and `-v` is explained in Section 4.4.

A SystemTap script can have command line arguments with a similar syntax to usual shells, like e.g. bash. Arguments are accessed via $ or @, depending on if the value is a number or a string, respectively.

4.4 Real-word example and operating principle

A SystemTap session begins when a SystemTap script (.stp file) is run with stap, or alternatively by using the shebang convention [14]. This session occurs in the following fashion:

1. First, SystemTap checks the script against the existing tapset library (normally in /usr/share/systemtap/tapset/) for any tapsets used. SystemTap will then substitute any located tapsets with their corresponding definitions in the tapset library
2. SystemTap's tools translates the script to C in a temporary directory under /tmp/, running the system C compiler to create a kernel module from it
3. staprun loads the module, then enables all the probes (events and handlers) in the script
4. As the events occur, their corresponding handlers are executed
5. Once the SystemTap session is terminated, the probes are disabled, and the kernel module is unloaded

In order to better understand this process it can be useful to keep the generated translation of C kernel-code, with the flag -k, and to increase passes verbosity with -v.

Below a full example that trace syscalls (see Chapter 1.2.1 for details) as follows:

1. probes on any syscall
2. when a syscall is made:
    - checks if the process that is doing a syscall is the target of the script, if that is the case it prints out the name and the number of the syscall and increments syscalls_tar
    - increments syscalls_tot
3. before exiting, prints out the total number of syscalls registered during the session, from both the target and the operating system

```
1   // syscalls.stp
2   global syscalls_tar, syscalls_tot
3
4   probe syscall_any {
5       if (pid() == target()){
6           printf("Target makes syscall: %s, number %d\n", name, syscall_nr)
7           syscalls_tar ++
8       }
9       syscalls_tot ++
10  }
11
```

```
12 probe end {
13     printf("\nTotal number of syscalls made by target is %d\n", syscalls_tar)
14     printf("Total number of syscalls is %d\n", syscalls_tot)
15 }
```

This script is called with the flags explained in Section 4.3 and has as the target (flag `-c`), a simple GNU-`find` command that counts the number of lines on each file of the linux kernel source code whose size is between 10KB and 20KB.

```
sudo stap -g -v -c "find /usr/src/linux/ -type f \
                    -size +10k -size -20k \
                    -exec wc -l {} \; > /dev/null" \
                -o results.txt -k ./syscalls.stp
```

In standard out we get a detailed description of the passes:

```
Pass 1: parsed user script and 480 library scripts using
        121880virt/104512res/11472shr/92780data kb, in 260usr/30sys/282real ms.
Pass 2: analyzed script: 3 probes, 4 functions, 95 embeds, 6 globals using
        138092virt/122004res/12656shr/108992data kb, in 240usr/140sys/383real ms.
Pass 3: translated to C into "/tmp/stapHvnDbd/stap_1143606_src.c" using
        138092virt/122004res/12656shr/108992data kb, in 20usr/0sys/25real ms.
Pass 4: compiled C into "stap_1143606.ko" in 6810usr/780sys/6683real ms.
Pass 5: starting run.
Pass 5: run completed in 980usr/450sys/1868real ms.
Keeping temporary directory "/tmp/stapHvnDbd"
```

And by doing a `cat results.txt`:

```
Target makes syscall: rt_sigreturn, number 15
Target makes syscall: rt_sigaction, number 13
Target makes syscall: rt_sigprocmask, number 14
[...]
Target makes syscall: rt_sigprocmask, number 14
Target makes syscall: rt_sigprocmask, number 14
Target makes syscall: exit_group, number 231

Total number of syscalls made by process is 259
Total number of syscalls is 217841
```

Chapter 5

Extracting processor internal features

5.1 Introduction

From Intel's Skylake processors onward, *dynamic voltage and frequency scaling* (DVFS) decisions happen in hardware. This feature is called Intel Speed Shift technology, discussed in Chapter 3. Although the algorithm is publicly available, the parameters controlling these decisions are silicon-dependent, therefore "naturally" difficult to take advantage of from a software perspective. The problem is that software is the gist of the workload done on computers, and the role of operating systems in controlling such work is often overlooked, despite being one of the most important part in determining both the speed and the power consumption decisions. An optimized OS can make the difference in both these aspects by staying in low power states or bursting the CPU, when it should ideally do so. Getting this balance right is alone as crucial as difficult, and moreover this adaptive behavior is limited by the lack of deep communication between hardware and software.

Our goal is to understand DVFS internals by inferring the above-mentioned parameters, in order to take conscious OS-level choices. The process of reverse-engineering such mechanism is ambitious, therefore still not at its final stage.

5.1.1 A simple overview

In order to achieve our goal we need to have a convenient way to examine MSRs (see Section 3.3 for more details), which are exposed to the user to monitor and control the CPU behavior. To summarize the phases to be discussed later on, we need to:

- have a configurable benchmark during which we trace our MSRs, according to various parameters, and

- analyze output data.

Of course, we do not want it to be tedious to work with, so we also need some sort of automation between various tests.

5.2 The tool chain

The tools needed to conduct the target analysis may vary substantially. The choice of tools must balance availability, simplicity, fitness. Also, the user must be comfortable enough with them. Our choice is:

Tracing: systemtap It is a full-featured tracing environment, still simpler than the raw `ftrace`. The key feature used is the capacity to read MSR in response to given events (see Chapter 4 for more details).

Benchmarking: rt-app It allows a very fine control of the temporal characteristics of the workload.

Automating execution: bash Very standard shell that enables scripting.

Log analysis: Jupyter (python) Well-known interactive data science and scientific computing open-source software, especially thanks to Jupyter Notebooks.

5.2.1 Systemtap script

The first phase consists in gathering data from the CPU, during a controlled benchmark (see Section 5.2.2). This stage is extremely delicate because the wrong tool (or its wrong usage), could potentially consume an excessive amount of resources, hence biasing significantly the evaluation. In other words, it is the only moment when we *really* care about the overhead. As discussed in Chapter 4, `systemtap` is an appropriate way to trace since it compiles in kernel's modules.

One of many useful `probes` of `systemtap` (see Section 4.2.3 for details) allows us to trace and link some actions (a function), to an interval, as explained in Section 4.2.3. This feature is needed to have a temporal overview of what is happening inside the CPU during our benchmark. Assuming everything else is configured properly, the code below does exactly that:

```
1  // log MSRs each £1 microseconds
2  probe timer.us($1) {
3      // attach some actions to the timer...
4      fill_entry()
5  }
```

The `fill_entry` function populates arrays containing:

- the following `MSRs`: `MPERF`, `APERF`, `PPERF`, `TSC` (see Section 3.3 for more information about their meaning)

- the temperature, determined by IA32_TEMPERATURE_TARGET and IA32_THERM_STATUS
- the CPU where it is executed systemtap
- the CPU where it is executed rt-app
- the state of the benchmark, 0 if it is sleeping, 1 otherwise

The readings of registers are made possible by one of the most powerful features of systemtap: the capability to embed C code, discussed in Section 4.2.5. Below an example function, used to read a register responsible for the temperature:

```
function read_dts:long() %{
  u64 dts, pkg_therm_status;
  rdmsrl(MSR_IA32_PACKAGE_THERM_STATUS, pkg_therm_status);
  dts = (pkg_therm_status >> 16) & 0x7F;
  STAP_RETURN(dts);
%}
```

Most of other items in the list are made trivial by systemtap built-in functions (see Section 4.2.2 for an overview of them): CPUs of systemtap and rt-app are found with cpu() and task_cpu(pid2task(target())), respectively; the temperature is found in C similarly to turbostat [20]. The less straightforward one is the status of the running benchmark. It needs in fact a probe from the scheduler family.

```
// check for context switches regarding £target
probe scheduler.ctxswitch {
  if (next_pid == target())
    running = 1
  else if (prev_pid == target())
    running = 0
}
```

The code checks if, in the scope of the current context switch, the benchmark is involved: if it is so, it changes the running variable accordingly (1 if it is the next process to be scheduled, 0 if it is the one that is being de-scheduled). See Section 4.2.2 and 4.3 for details about target().

Readings frequency

Looking at the systemtap script shown in Section 5.2.1, one wonders what can be an appropriate $1, that is the amount of time between each read of MSRs.

Reading MSRs is a cost-intensive task at around ∼1000 CPU cycles. We decided to test its efficiency with a simple systemtap script that loops 10 million times and evaluates the time spent for and between readings, and finally stores them in a set thanks to the aggregation operator (<<<, discussed in

Section 4.2.6). This test aims to address two questions: what is the times in microseconds required to read these registers, as well as check if there may occur any errors if the interval between readings is too short. The latter is achieved by a property of `MPERF`, namely the fact that it should never "fall", go down, over time. The `if` statement verify exactly this case.

```
1  prev = 0
2  i = 0
3  old_read = 0
4
5  while(i++ < 10000000){
6      start_t = gettimeofday_ns()
7      new = read_mperf()
8      end_t = gettimeofday_ns()
9
10     if(new <= prev){
11       printf("End, invalid mperf value:%d vs old:%d, limit found us\n",
12              new, prev)
13       break
14     }
15
16     prev = new
17
18     // stat on time per read
19     elapsed_t = end_t - start_t
20     time_on_read <<< elapsed_t
21
22     // stat on time bewteen read, skipping first
23     if (old_read)
24       intervals <<< end_t - old_read
25
26     old_read = end_t
27 }
```

We then used some built-in functions, presented in Section 4.2.6, to obtain a basic report of gathered data contained in aggregates.

```
---Statistics on time spent on each read---
time_on_read min:63us avg:71us max:26095us
total:717290330us count:10000000 variance:27123
value |-------------------------------------------------- count
    8 |                                                         0
   16 |                                                         0
   32 |                                                       116
   64 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  9961146
  128 |                                                      3974
  256 |                                                     33517
  512 |                                                        25
 1024 |                                                       168
 2048 |                                                        82
 4096 |                                                       234
```

```
 8192 |                                                     728
16384 |                                                      10
32768 |                                                       0
65536 |                                                       0

Avg amount of reads per microsecond:0.01394

-----------------------------------------------------------------

---Statistics on time between each read performed---
intervals min:163us avg:179us max:76286us
total:1794140863us count:9999999 variance:77980
 value |-------------------------------------------------- count
    32 |                                                       0
    64 |                                                       0
   128 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  9923505
   256 |                                                   66053
   512 |                                                    7323
  1024 |                                                     338
  2048 |                                                     259
  4096 |                                                     640
  8192 |                                                    1856
 16384 |                                                      22
 32768 |                                                       1
 65536 |                                                       2
131072 |                                                       0
262144 |                                                       0

Avg amount of loops per microsecond:0.00557

Total script time (exec of loop): 1794142401us
```

The first thing we notice is that there are no errors, values of MPERF grow constantly as expected. Besides that, these graphs show an average of ~80us and ~180us per readings and between readings, respectively. Approximating these values to 100us, we find a reasonable value to use in our systemtap script before encountering unexpected results. This value will also be the minimum allowed in the bash script, where many other checks are made. As mentioned in Section 3.4, we also know that the *optimal point* P_e of frequency and energy consumption is updated by HWP every 1ms, so by staying below this threshold, we hopefully get a more accurate view of MSRs' variations.

The calling script and its parameters

Going back to the original script, that is record_msr.stp, the last thing to observe is how it is executed during the automation phase 5.2.3.

```
# shell code that calls systemtap
sudo taskset --cpu-list "$STAP_CPU" \
  ./record_msr.stp -c "$BENCHMARK" \
  -o ./data/log/"$filename".log \
  -s "$MAXFILESIZE" -D MAXMAPENTRIES="$MAXARRAYENTRY" \
  --suppress-time-limits \
  "$interval"
```

Among the used flags and parameters it is important to say that we needed to suppress security checks done by `systemtap`'s compiler. These include both time and space controls. In order to minimize work done during the tracing itself we decided to print data to `stdout` at the end of the execution, thanks to the flag `-o $filename.log` and the `probe end` statement, that are both explained in Chapter 4.

5.2.2 rt-app workload

Our benchmark of choice was `rt-app`, a test application that starts multiple periodic threads in order to simulate a real-time periodic load [12].

`rt-app` is a tool that can be used to emulate a use case. Not only the sleep and run pattern can be emulated but also the dependency between tasks like accessing same critical resources, creating sequential wake up or syncing the wake up of threads. The use case is described in a `json` like file whose main objects are:

- resources
- global
- tasks

Despite `rt-app`'s many available options through the configuration file, we decided to go for a minimal setup, that certainly does not take advantage of the most powerful core functionalities. This choice is due to the fact that a more complex benchmark is not necessarily better in our context, indeed a simpler one generates a more linear and clean log file, easier to study and understand. That said, below the list of `rt-app`'s fundamentals we used:

“sleep” sleep time of a task in usec

“run” run time of a task in usec

“duration” total duration of the use case in sec

“calibration” the CPU used to calibrate the nsec per loop value

For a better understanding of `rt-app` it is strongly suggested taking a look at its documentation [13], from which we took part of the content of this section.

```
1  {
2    "tasks": {
3      "thread1": {
4        "instance": 1,
5        "loop": -1,
6        "run": 800000,
7        "sleep": 200000
8      }
9    },
10   "global": {
11     "duration": 30,
12     "calibration": "CPU0",
13     "default_policy": "SCHED_FIFO"
14   }
15 }
```

Figure 5.1: rt-app code

rt-app template of choice

Figure 5.1 shows a complete and working configuration file, that is the template that is going to be the most widely in this study, where:

- one instance of a thread is spawned
- there is no predefined number of loops
- the run time is 0.8s
- the sleep time is 0.2s
- the total duration is 30s
- the calibration is done on CPU 0
- the chosen scheduler is `SCHED_FIFO`

Alternatives benchmark

Despite our first choice as a workload program was `rt-app` we did not want to limit our analysis to that, so in the script 5.2.3 it is allowed to use any program as the preferred benchmark. Since the tracing environment was designed with `rt-app` in mind, support for this type of run is limited, in the sense that fewer information are logged and general user-experience feels more diminished, contrary to `rt-app` whose essential parameters are configurable.

5.2.3 bash script

The `bash` script, named `test_msr.sh`, is the core that "glues" together all parts of the tool chain, by providing a simple command line interface (cli) utility that automates the benchmark and its tracing.

In addition to coordinating the tools, the script also does some security checks to avoid errors that could lead to compromise the testing machine (e.g. wrong CPU values, inconsistent/dangerous parameters, running identical benchmark, etc.).

A full list of flags and their values can be found by executing `test_msr.sh -h`:

```
Usage :  test_msr.sh [-h] [-v] [-i n] [-l n] [-p n]
                     [-o] [-e n] [-w n] [-B s] [-b n] [-s n]

    Options:
    -h    Display this message
    -v    Display script version
    -i n  Set us interval between MSR reads.
          It must be greater then 100.
    -l n  Set a benchmark load percentage.
          It must be between 1 and 100.
    -p n  Set the us period of the benchmark.
          It must be between 1 and 10000000.
    -o    Online: launch Jupyter at the end.
    -e n  Energy performance preference (EPP):
          between 0 max and 255 min.
              VALUE STRING          EPP
              performance           0
              balance-performance   128
              normal, default       128
              balance-power         192
              power                 255
    -w n  Activity-window: sliding window in us used by HWP to
          maintain avg frequency. 0 means hardware-managed.
          The value must be between 0 and 127 * 10^7.
          Note: this value must be set together with -e n,
          where n is not 0.
    -B s  Set custom benchmark, must be set without -l -p
    -b n  Set (valid) custom CPU for the benchmark.
    -s s  Set (valid) custom CPU for systemtap.

    Defaults:
        [i]nterval  = 100
        [l]oad      = 80
        [p]eriod    = 1000000
        [o]nline    = not set
        [e]PP       = 128 - powersave
```

```
        [w]indow    = 0 - hardware-managed
        [s]tap_cpu  = 0
        [b]ench_cpu = 0
        [B]ench_alt = not set

    Dependencies:
        bash
        GNU coreutils
        jq
        systemtap
        rt-app
        msr-tools
        [python]
        [jupyter]
```

If everything is set-up correctly, it does the following:

1. makes a request to the `HWP` infrastructure 3.4
2. prints out a summary of how the set-up was configured
3. starts `rt-app` and `systemtap` with chosen values
4. eventually open jupyter notebook
5. reset system preferences of `HWP` (EPP and window) 3.3

All these steps are trivial since their simple scripting nature, also thanks to the external tools that have been used (e.g. `jq` [3] is extremely useful when dealing with `json` files). The only one that is less obvious and require some additional manipulation besides simple variables tweaking is the first one, the `HWP` request.

How to make HWP requests

Despite the useful tools provided by Intel for `MSRs` configuration(see Section 3.4.1 for details), direct interactions with these programs can be counter intuitive due to how the `HWP_REQUEST` register needs to be written.

As discussed in Section 3.3, we have many configurable parameters for `HWP`, provided by `HWP_REQUEST`, we repeate two of them here for convenience:

1. Energy_Performance_Preference (bits 31:24, RW)
2. Activity_Window (bits 41:32, RW)

In this context we want to set this two fields, the window and the energy performance preference (EPP). In order to do so we need to avoid overriding all the register, so we need a $\vee$ between the current value of `HWP_REQUEST` and our preferred configuration. This operation alone would not justify a whole another

script, but the tricky part comes when dealing with the window: the first 3 bits are used for the exponent, the last 7 for the desired value.

We also know from Chapter 3 that `HWP_REQUEST` is a core dependent `MSR`, as opposed to `HWP_REQUEST_REG_PKG`, so we also need the CPU where we want to perform this.

This “complexity” and the fact that this script has its own logic and use, also outside of this study, has led us to make it an independent file.

```
#! /usr/bin/env bash

# support script that makes hwp_request through intel/msr-tools
# usage: hwp_request.sh CPU WIN EPP
#
# WIN:
# 10 bits:
#           10:7 -> 3 used for exp
#            6:0 -> 7 used for value
# max value is
#              111 1111111
#            =   7 127
#            = 127 * 10 ^ 7
#
# EPP:
# 8 bits: represents performance from max 0 to min 255

set -e

CPU=$1
HWP_REQUEST_REG=0x00000774

# properly parse activity window as understood from HWP
function parse_window {
  win="$1"

  for ((e = 0; ; e++)); do
    [[ $win -le 127 ]] && break
    win=$((win / 10))
  done

```

```
  echo $(((e << 7) | win))
}

current_hwp=$2

# epp
if [[ -n $3 ]]; then
  new_epp=$(printf '0x%x' "$3")
  current_hwp=$((current_hwp & ~(new_win << 24)))
  current_hwp=$((current_hwp | (new_epp << 24)))
fi

# activity window
if [[ -n $4 ]]; then
  new_win=$(parse_window "$4")
  current_hwp=$((current_hwp & ~(new_win << 32)))
  current_hwp=$((current_hwp | (new_win << 32)))
fi

# make request
new_hwp=$(printf '0x%x\n' $current_hwp)
sudo wrmsr --processor "$CPU" "$HWP_REQUEST_REG" "$new_hwp"
```

5.2.4 Data analysis in Jupyter

Among the alternatives for data analysis our choice fell on python-jupyter for two main reasons:

- the ease of use
- its interactive nature

The former point is easily explained by looking at the enormous support that both projects got in recent times, these tools are well documented and get constantly updated. The large number of packages available in python, together with the power of jupyter, make it a simple, yet powerful, environment even for users with little to no experience.

The latter is mostly due to jupyter notebooks, files produced by the jupyter notebook app that contain code and rich text divided in cells, that can be run independently from one to another. This makes it really convenient for our use case, since it fits the “research” scope, where many trials and errors happen.

In this section we will see the basics core functionalities that were used in the analysis. Our description will be concise since, despite the seemingly length of the notebook file, many parts are mechanics and therefore of less interest. Note that all the “pure algorithmic” parts are postponed to Section 5.4, and will be discussed “locally” to each case study, since they heavily depend on the context.

Below the action performed to prepare the dataframe, starting from the log file generated from `systemtap` 5.2.1:

1. store the content of the log file in a variable named `data` via `pandas`’s [9] dataframes

2. fix the `time` field of the log, converting it to milliseconds, independently from the sampling period of `systemtap`

3. derive values of `APERF`, `MPERF`, `PPERF` through the following snippet that use `numpy` [7]:

```
1 mperf_der = np.diff(data.mperf)/np.diff(data.time)
2 aperf_der = np.diff(data.aperf)/np.diff(data.time)
3 pperf_der = np.diff(data.pperf)/np.diff(data.time)
```

All the other collected values (e.g. the temperature) are ready, since they were not collected “raw” but already processed in a reasonable unit of measure or simply because they did non need any alteration.

In order to dynamically compare multiple plots without re-executing our code, we took advantage of another useful package from jupyter: `ipywidgets`. With little scripting we were able to build a widget that updates live and offers the possibility to:

- choose the **graph view** of the log, by selecting only some parts of the plot (e.g. in *period* only one period is shown), executing live data manipulation (e.g. in *ratios* `MSRs` values get divided), adding/removing traces (e.g. in *freq* we add the CPU frequency)

- select a specific **log file**

- change the **granularity** of the log, that is the value of the rolling average (this feature helps with the smoothness of the plot)

- change the **path** where to look for log files

Figure 5.2 shows the widgets with the above-listed options.

Once all values from the widget are set, these are passed to a function that updates the plot, that is rendered with our library of choice: `plotly`. Built on top of `plotly.js`, `plotly.py` is a high-level, declarative charting library [11]. Its simplicity allows to easily write interactive plots with few line of code, in addition they can be zoomed, panned and exported in various formats, thus completing our tool chain.

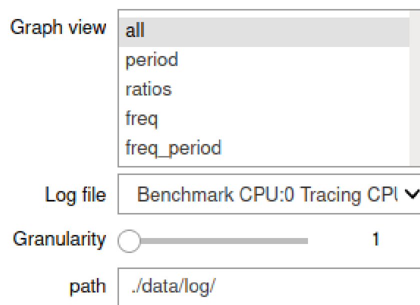

Figure 5.2: Example ipywidgets

5.3 Testing machine setup

In order to give more context, in this section we are going to briefly see what is the setup on the testing machine.

We used an Acer machine, whose Intel's processor specs are fully available at Intel's site [2]. In Table 5.1 are shown software's and hardware's highlights.

software	version
OS	GNU/Linux (Arch)
Kernel	5.11.10-arch1-1
rt-app	v1.0-137-g3a95bf5
systemtap	4.5/0.183

hardware	version
CPU	i5-6200U CPU @ 2.30GHz
max-freq	2800MHz
min-freq	400MHz
architecture	x86_64

Table 5.1: Environment specifications

We decided to boot the machine in `multi-user.target`, that is a `tty` without a graphical-user-interface (GUI). That choice is intended to lower the use of resources outside of `rt-app`; since the GUI is considered one of the most resource-heavy program, we cut off its interference with the goal of this project.

The processor is equipped with HWP and was set to the default turbo configuration, which was "on". We checked these configurations with the commands shown below.

```
# check turbo
grep flags /proc/cpuinfo | grep -q '\sida\s' && echo on || echo off
# check HWP
journalctl -b | grep -q pstate && echo on || echo off
```

Many other useful information about the processor can be found by seeing Intel p-state settings: `cat /sys/devices/system/cpu/intel_pstate/*`.

5.4 Experiments

The tool chain explained in Section 5.2 leads us to a "complete environment" that can generate reproducible benchmarks (5.2.2), trace (5.2.1) `MSRs` (3) and allow to analyze generated data (5.2.4).

After an empiric observation of plots, we noticed that there were common/recurrent patterns, so we decided to examine the most interesting ones.

All the patterns refer to a single "period", that translates to the period of `rt-app`. As mentioned earlier in Section 5.2.2, the benchmark is indeed characterized by a `run` time and a `sleep` time, repeated for an amount of time equal to `global.duration`; *period* means the union of both a single `run` and `sleep`. Figure 5.1 describes the rt-app code used to simulate the workload and shows a period of $1s = 1000000us = 800000_{run} + 200000_{sleep}$.

Analyzing Figure 5.3 we detected several behaviors of the processor DVFS. Below, we list the most relevant ones:

(A) `APERF` has a step-like growth

(B) `MPERF` drops in correspondence of `APERF` steps

(C) `APERF` is mostly higher than `MPERF`

(D) `MPERF` rises at idle

(E) temperature variations between idle/running cores

(F) `APERF` sporadically slows down during running phases

Due to the limited time, only (A), (B), (C) and (E) are investigated in greater depth. Section 5.4.1 illustrates the phenomenon (A), Section 5.4.2 the phenomenon (B), Section 5.4.3 the phenomenon (C), Section 5.4.4 the phenomenon (E).

5.4.1 Steps of frequency growth

Looking closer at one period in the plots, the first thing we noticed was the regular path of `APERF` growth during the ascent phase that follows sleep intervals of all periods, e.g. the one shown in Figure 5.4.

If we add another trace to the plot, the frequency of the processor, that is

$$\frac{APERF_DER \times BASE_FREQUENCY}{MPERF_DER} \tag{5.1}$$

in each point, where $BASE_FREQUENCY$ is the base/regular frequency, $APERF_DER$ and $MPERF_DER$ are the derivatives of `APERF` and `MPERF` respectively, we can see in Figure 5.5 how the steps of `APERF` follow exactly the steps of the frequency, as expected 3.3.

Right away `APERF` steps clearly seem a promising direction in the context of DVFS, thus all experiments focus on various correlation of `APERF` and other factors.

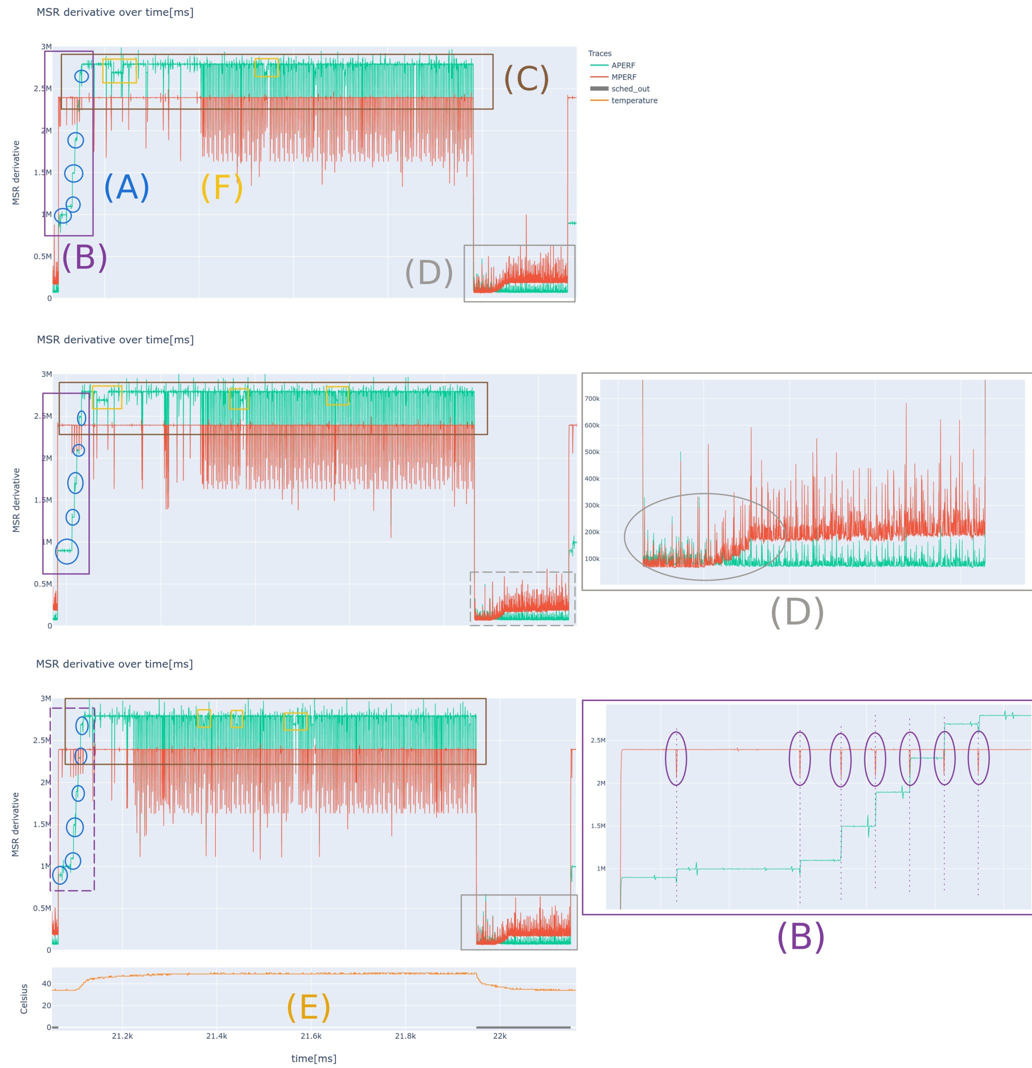

Figure 5.3: Vertical alignment of three sample periods. Dashed squares are zoomed on the right for better understanding. Lines of the same color identify the same phenomenon.

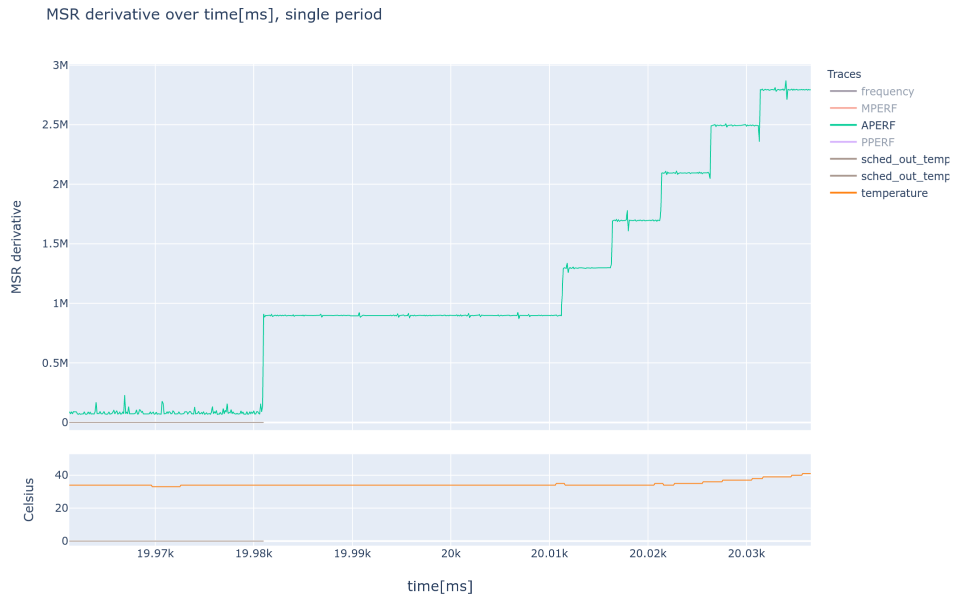

Figure 5.4: Regular, step-like, APERF tendency

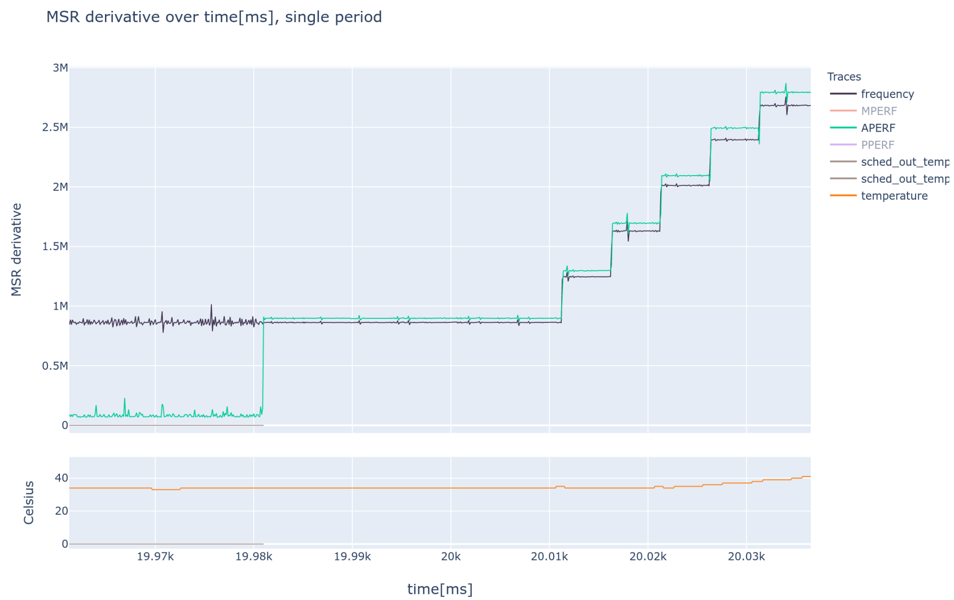

Figure 5.5: Plot with frequency

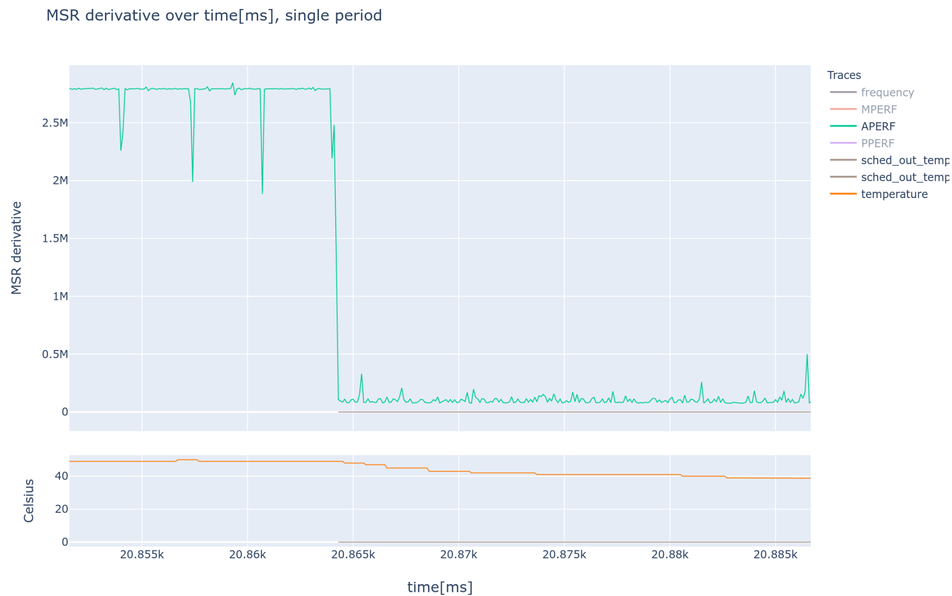

Figure 5.6: Drop example

Our approach to determine the pattern of frequency growth in presence of the release of a new workload is as follows:

1. we determine the instants in time when the speed has changed and then
2. we compute the average frequency during each step.

Step detection algorithm

The first pass to make in order to analyze these steps is obviously to programmatic extract them from the log file generated via `systemtap` 5.2.1. In that phase we took a naive approach, to avoid over-complicating the algorithm. This detection must not be overly accurate, as stated before, only for "false positive", that is steps that may or may not be in fact "real" steps, but should not miss any actual step, or in other words should not produce "false negative".

Before proceeding on the explanation of this phase it is important to note why there may be "false positive": our log files, and by reflection our plots, contain a lot of "noise", that is spikes in both Y axis direction; this is caused by the very high frequency in which we read `MSRs` (see Section 5.2.1).

Going back to the algorithm, our goal is to locate and extract the time (the `X` axis) where the rise begins in the plot (e.g. 5.4). We also need to find where the plot of `APERF` "drops" (its usage will be clearer later), or in other words the point just before the `sleep` begins.

In order to achieve that detection we used a basic loop that find the `delta`, that is the difference between each point `i` and the point found after `length` values in the plot. Below a simplistic version of the python code responsible for this; its real implementation does not go far away from what is shown here.

```
while i + length < len(aperf):
    delta = aperf[i + length - 1] - aperf[i]

    if delta > threshold_spikes:
        spikes.append(i)
    elif delta < threshold_drops:
        drops.append(i)

    if found:
        i += length
    else:
        i += 1

return spikes, drops
```

`length`, `threshold_drops` and `threshold_spikes` are pretty much hard-coded value, derived from an empiric observations of plots. The `i_th` point is considered a step if the `delta` is *sufficient*, meaning it is big enough to be considered as such: it increase/decrease of an amount bigger than `threshold_spikes` /`threshold_drops`, in a small interval of time, equals to `length`.

The need for a mean

By applying this algorithm, we encounter a problem that was already mentioned: our data is too “noisy”. This reflects to an inflated number of steps/drops detected. In order to mitigate that we decided to apply a rolling mean.

Our first choice was to “do it manually” together with the calculation of the derivative, like shown below.

```
 while i < len(data) - rolling_value - 1:
        dx = data[i + rolling_value].time - data[i].time
        aperf = data[i + rolling_value].aperf - data[i].aperf
        aperf_der.append(aperf/dx)
        i += 1
```

However, by doing that our code slowed down dramatically, due to the large log file generated 5.2.1. That was solved by simply using `numpy` [7], a python package used for scientific computing written in python and C.

```
data.aperf = np.diff(data.aperf)/np.diff(data.time)
data.aperf = data.rolling(rolling_value).mean()
```

By using that version of the mean, we reduced the run time of that section from ∼5m40s to ∼50s on average.

It is worth noting that this approach is not exactly the same as the previous one: in fact it does not calculates the derivative together with the rolling mean, but splits the process into sequential phases. Anyway, that did not seem to produce any unexpected results, therefore our investigation on that topic ended here.

Setup

Below `rt-app`'s configuration file with which the benchmark was started. Since its parameters were not crucial in this experiment we decided to leave them as the default one.

```
{
  "tasks": {
    "thread1": {
      "instance": 1,
      "loop": -1,
      "run": 800000,
      "sleep": 200000
    }
  },
  "global": {
    "duration": 30,
    "calibration": "CPU0",
    "default_policy": "SCHED_FIFO",
    "gnuplot": false
  }
}
```

To sum up:

- benchmark done in CPU: 0 , tracing done in CPU: 0
- traced each 100 us with a benchmark load of 80%
- the benchmark has run for 30s with a period of 1000000 us
- HWP request parameters: EPP = 128 , activity window = 0

Results

Figure 5.7 shows a zoom of the plot generated with data computed by the algorithm that detect the steps of `APERF`.

Figure 5.8 shows the distribution of `APERF` steps, by plotting:

- on the x axis the length of each step, that is the time spent on that particular step
- on the y axis the height of each step, that is the amplitude of the step, representing the frequency increase/decrease

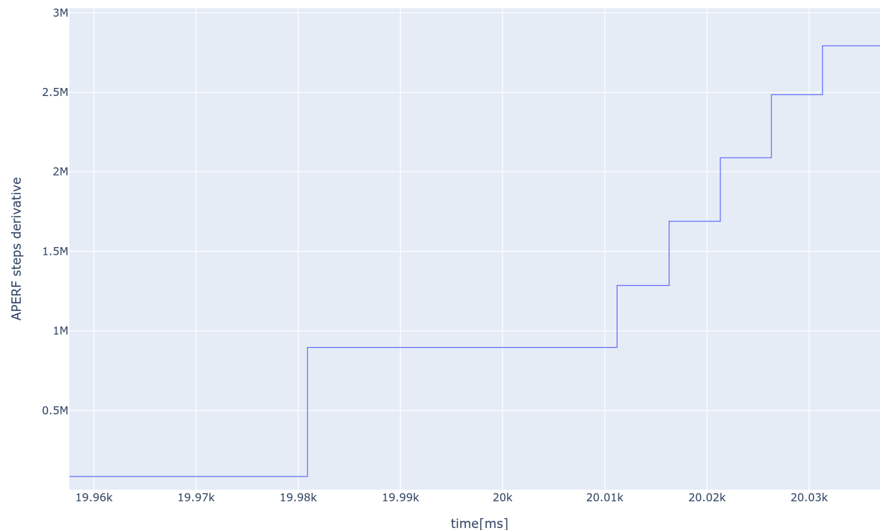

Figure 5.7: Plot of collected steps, zoom of one period

Leaving aside the noisy steps on the highest and lowest part of the plot, the figure clearly shows various patterns on both the height and time coordinates. Among the many, the straight line is very noticeable. This line shows that the steps are almost equidistant from a frequency point of view, meaning that they probably differ by a multiple. Another observation is that the line is almost exactly straight, which implies a fixed duration of the steps.

Figure 5.9 is taken from Intel and shows a similar result to Figure 5.7.

5.4.2 Frequency switch overhead

Figure 5.10 shows both the plots of `APERF` 5.5 and `MPERF`. We can observe that whenever `APERF` changes to a different frequency value, `MPERF` is affected as well. This behavior is explained by the necessary overhead to switch from a clock frequency to another. In fact the CPU circuitry need some time to adapt to the new running frequency. Processors cannot select any frequency between the maximum and minimum frequency available, however they can choose a different multiple (also mentioned in Section 5.4.1 and shown in Figure 5.8) of the base clock and set to it through p-states, which are explained in Section 3.2. The fact is that this change cannot happen live, and requires the processor to be temporarily switch off and turned back on at a higher frequency, hence the slowdown visible in Figure 5.10. Indeed `MPERF` grows at a constant rate configured at boot time, so when it slows down it is due to the processor being

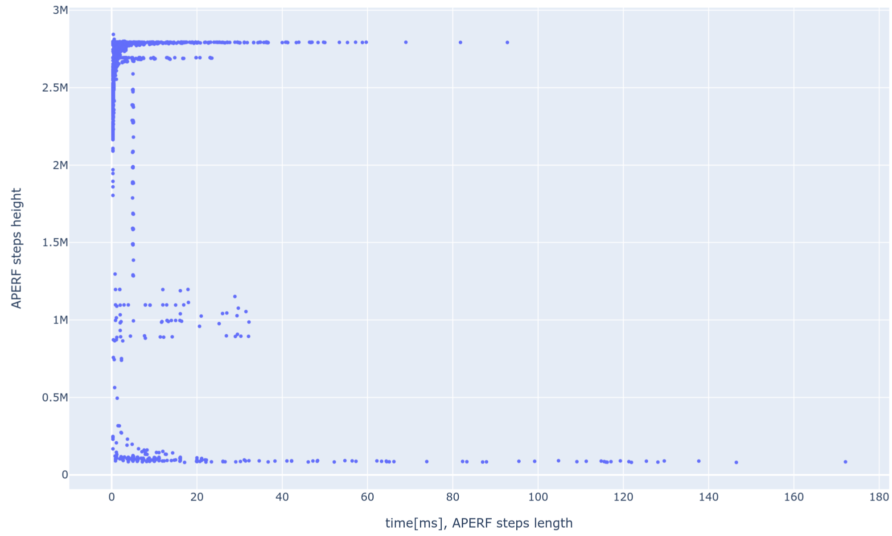

Figure 5.8: Distribution plot of collected steps

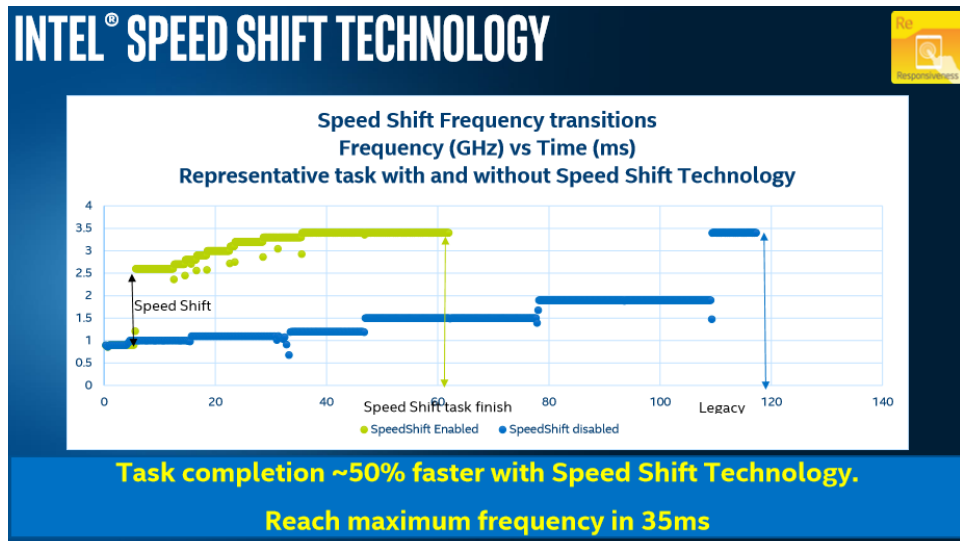

Figure 5.9: Intel's data on HWP

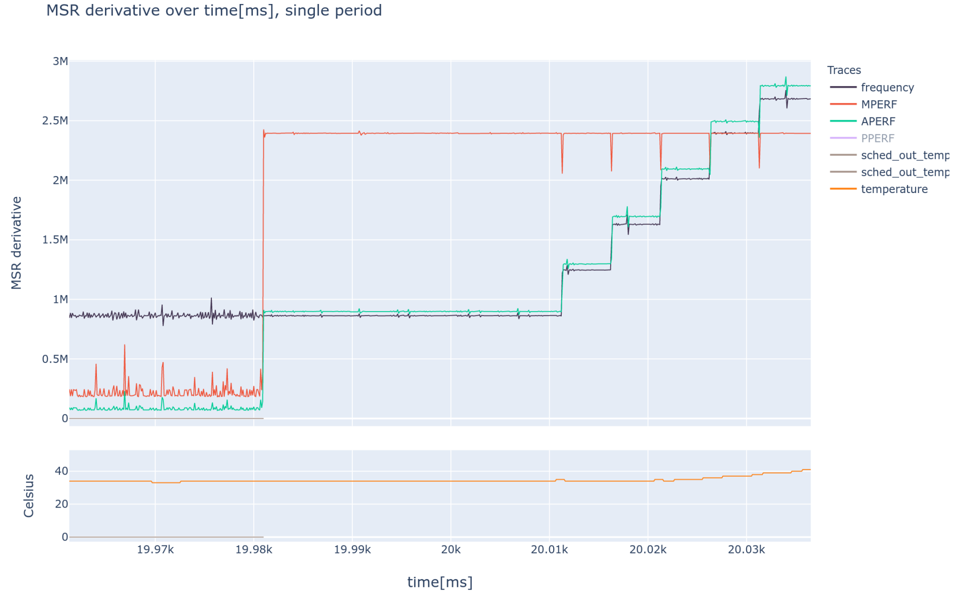

Figure 5.10: Plot with MPERF

idle (see Section 3.3 for a better understanding of `MPERF`). This behavior is well known, and among the many interesting aspects behind it, the overhead of this operation stands out and is the focus of many papers about DVFS.

Setup

Since this experiment is essentially one prosecution of the one shown in Section 5.4.1, its setup is left unchanged.

Results

Our research on that topic is unfortunately very superficial and limited. The decision was to "manually" estimate an average overhead on the processor with the following formula:

$$T \times \left(1 - \frac{f_{drop}}{f_{nom}}\right) \tag{5.2}$$

where T is the sampling interval, f_{drop} is the frequency registered when the switch is happening, f_{nom} is the frequency right before (or after) the switch. For a better understanding on those value, see Figure 5.11, that is the detail of the first purple circle on figure's zoom (B) 5.3.

We tested the overhead with a sampling interval T of 100us, and the average was $\sim$14us. Another thing that we noticed in this calculations was that at higher

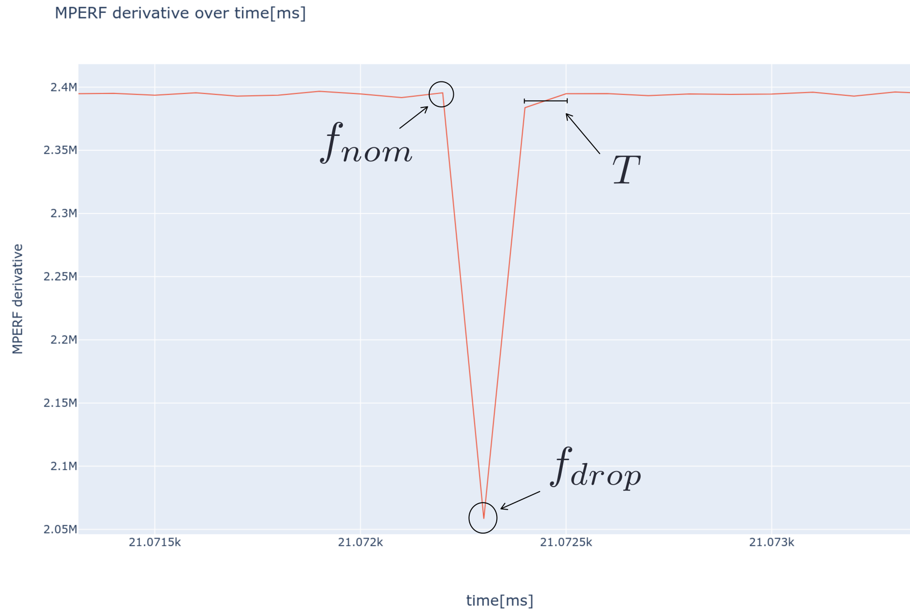

Figure 5.11: Frequency overhead parameters illustrated on sample period.

f_{nom}	f_{drop}	result
2.3955	2.05831	14.075976
2.39472	2.06796	13.645019
2.39456	2.06022	13.962482
2.39472	2.07801	13.225346

Table 5.2: Registered samples of nomimal and drop frequencies. Column *result* shows the overhead in microseconds computed using formula 5.2.

frequencies the overhead was lesser. Table 5.2 shows data used for this test and respective results.

5.4.3 Turbo always on

One of the recurrent patterns that stands out when looking at the three sample period in Figure 5.3, is the fact that `APERF`, stays almost always above `MPERF`. To be more precise, this happens approximately when `rt-app` starts and the processor increases its speed (the steps progress explained in Section 5.4.1), but does not happen when idling. Figure 5.12 shows the first sample period, which clearly reaffirms what was said.

The processor stays at turbo no matter what the load is, and oddly enough, also to what the duration of the benchmark is. The former statement can be

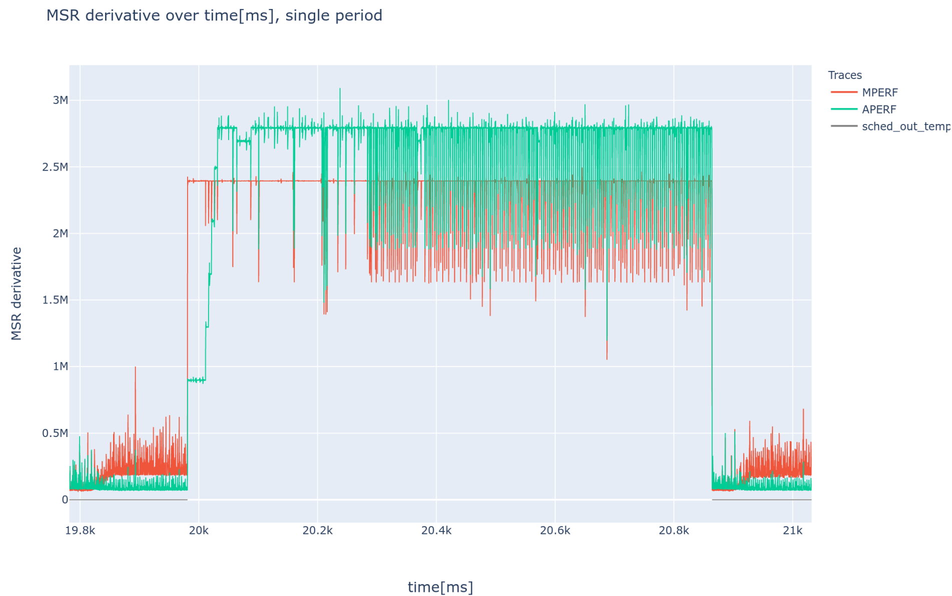

Figure 5.12: APERF stays "high"

seen in Figure 5.13 and 5.14, where even a 10% 100% load respectively, did not change this behavior, despite going idle for a short amount of time in Figure 5.14. The latter has been tested with many long (up to ∼8hours) benchmarks that were run without any idle phase.

Setup

When focusing on that experiment, our main focus was to understand if it was possible to have ratio of `APERF` and `MPERF` a lower than 1, that means no turbo. In order to do so we tried a lot of benchmarks, many of which are almost equals to the ones shown previously. The only types of benchmarks that are of interest to us are the one that do not use `rt-app`, as they are not shown elsewhere. For the purpose aforementioned we wanted in fact to try different kind of benchmarks, especially because this behavior seemed "strange" at first, as naturally we did not expect that a processor could have run at turbo frequencies for long periods. We took advantage of the flag `-B` of the `bash` script 5.2.3, that gives the possibility to pass any ececutable to the script and trace during it. For the long and intensive benchmarks we opted for `sysbench`, since it provides a number of algorithms to generate random numbers that are distributed according to a given probability distribution [34]. For example, the longest benchmark that has been tested (the one that lasted 8hours), was generated with the command shown below:

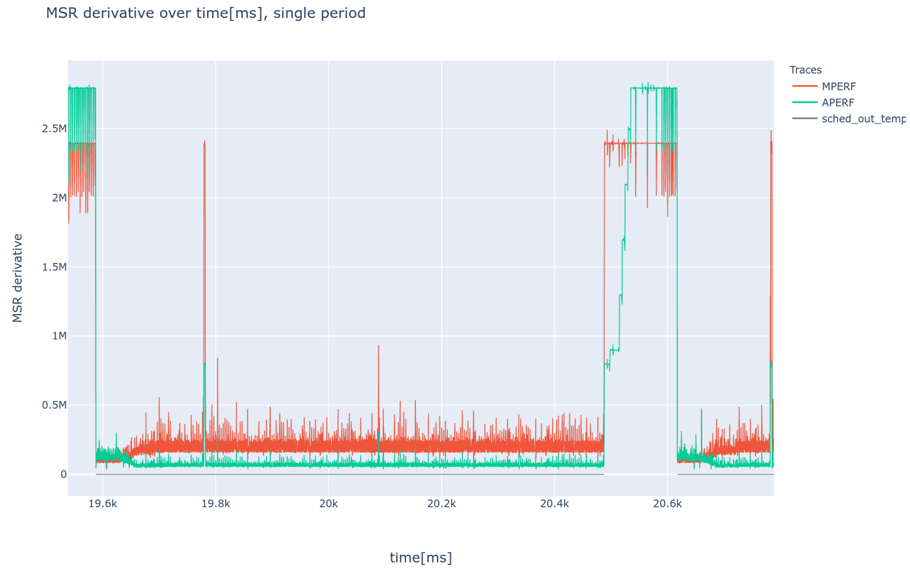

Figure 5.13: Light benchmark

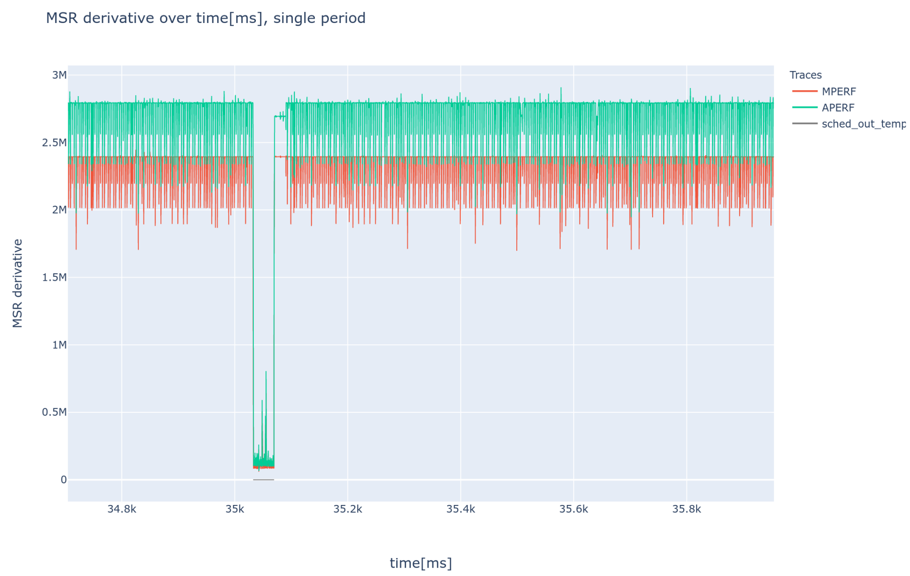

Figure 5.14: Intensive benchmark

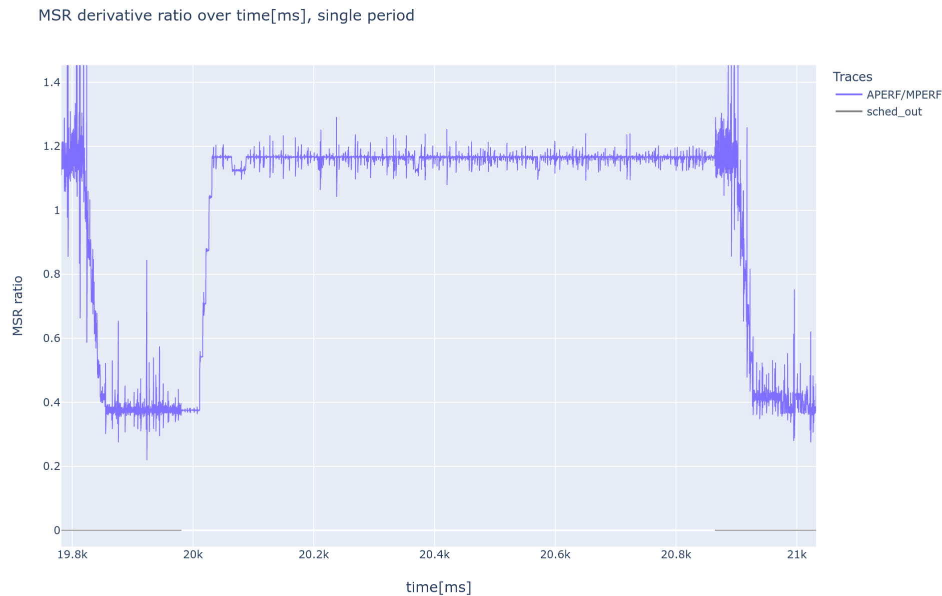

Figure 5.15: $\frac{APERF}{MPERF}$ ratio

```
test_msr.sh -i 1000000 -B "sysbench --threads=10000 \
           --cpu-max-prime=10000000 cpu run"
```

Results

By looking at the plot shown in Figure 5.15, that is ratio between `APERF` and `MPERF` of Figure 5.12, we can see that it is approximately between 1.1 and 1.2. This value tends to 1.1 when the benchmark is longer enough; this still means that turbo is active, as discussed in Chapter 3.

Our conclusion about this pattern was simply that the tested processor 5.3 is capable of maintaining, although "low", turbo frequencies for long period of time without particular throttling.

5.4.4 Temperature variations

The temperature is a key element regarding power consumption and frequency scaling, it is in fact one of the parameters involved when the CPU wants to make a change in its speed. This is due to the fact that an abnormal high temperature can potentially damage machine's parts, so it is fundamental to refer to it when trying to "go faster".

Our experiment wants to study the consequences that an intensive task, pinned on one CPU, cause on the whole package (all the CPUs).

Setup

This experiment required tweaking both the `systemtap`'s and `bash`'s scripts. Our goal is to have one *main* CPU, that run the benchmark and trace as explained in Section 5.2.1, while the others stay idle and trace only the temperature. All the CPUs need to exit when the benchmark ends.

We chose as the main the CPU 0 and made a basic synchronization mechanism. In order to achieve that, after storing the CPUs' number in an array called `STAP_CPU`, we do the following:

(A) start `rt-app` and `systemtap` on the main CPU (the first element in `STAP_CPU`), as described in Section 5.2.1

(B) save its `pid` in the array `ALL_PIDS`

(C) stop the process

(D) for each other CPUs, start a "reduced" version of `systemtap` script, named `record_msr_multi.stp`

(E) save all `pids` in `ALL_PIDS`

(F) stop all the processes immediately after their start

(G) after all processes have done this cycle (started and then stopped), resume them together

(H) make the shell wait for all the processes to end

This complexity is needed because without any control, there is an higher chance of "misalignment" between CPUs' time and executions. The `bash` code that does the synchronization mechanism is the following:

```
1  ALL_PIDS=()
2
3  # main process
4  sudo taskset --cpu-list "${STAP_CPU[0]}" \       # (A)
5    ./record_msr.stp -c "$BENCHMARK" \
6    -o ./data/log/multi/"$filename""${STAP_CPU[0]}".log \
7    -s "$MAXFILESIZE" -D MAXMAPENTRIES="$MAXARRAYENTRY" \
8    --suppress-time-limits \
9    "$interval" \
10   &
11
12 ALL_PIDS+=($!)                                    # (B)
13
14 sudo kill -TSTP "${ALL_PIDS[0]}"                  # (C)
```

```

for stap_cpu_curr in "${STAP_CPU[@]:1}"; do      # (D)
  sudo taskset --cpu-list "$stap_cpu_curr" \
    ./record_msr_multi.stp \
    -o ./data/log/multi/"$filename""$stap_cpu_curr".log \
    -s "$MAXFILESIZE" -D MAXMAPENTRIES="$MAXARRAYENTRY" \
    --suppress-time-limits \
    "$interval" \
    "${ALL_PIDS[0]}" \ # main process pid, arg £2 in systemtap
    &

    ALL_PIDS+=($!)                                # (E)

    sudo kill -TSTP "$!"                          # (F)
done

sudo kill -CONT "${ALL_PIDS[@]}"                  # (G)

wait "${ALL_PIDS[@]}"                             # (H)
```

We also used a shortened version of the `systemtap` script, that trace only the temperature and exit when our main process, the one run on CPU 0, ends. The latter task is accomplished by passing an extra argument to the script, the `pid` of the main process, and by adding the following snippet:

```
// exit when pid £2 ends
probe process($2).end {
    exit()
}
```

Results

Figure 5.16 shows collected temperatures for the CPUs and the package, Figure 5.17 the derivative of `APERF`, traced on the core where `rt-app` started, that is the CPU 0. Both figures are zoomed on three sample periods.

This experiment highlights the following points:

- the high-usage of one core has an impact on the others, as expected, since heat transfers to nearby materials
- there is one sensor for physical core. It was clear indeed how data gathered on CPU 0/CPU 2 and CPU 1/CPU 3 came from the same sensor,

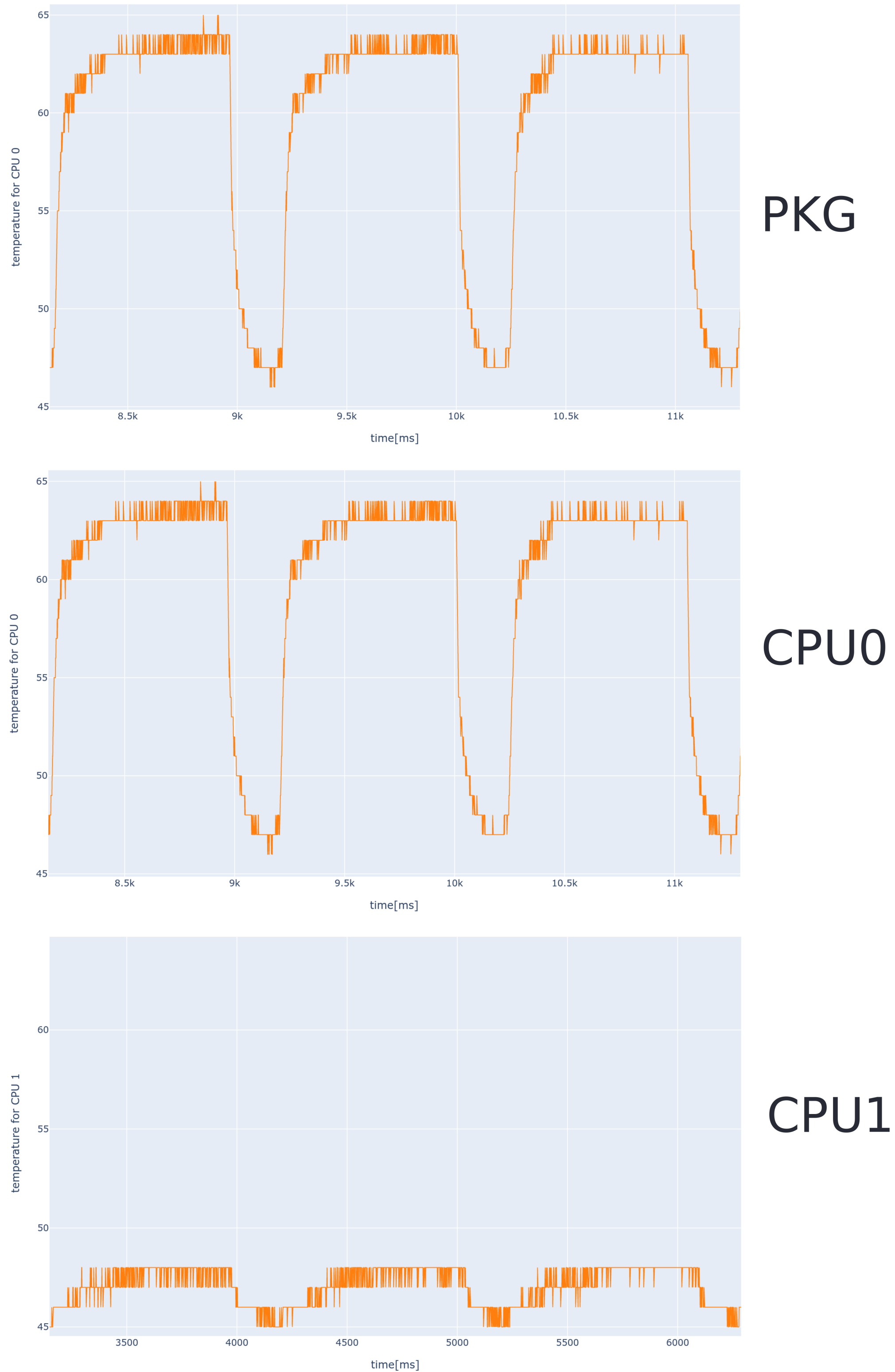

Figure 5.16: Temperature variations registered simultaneously on all cores

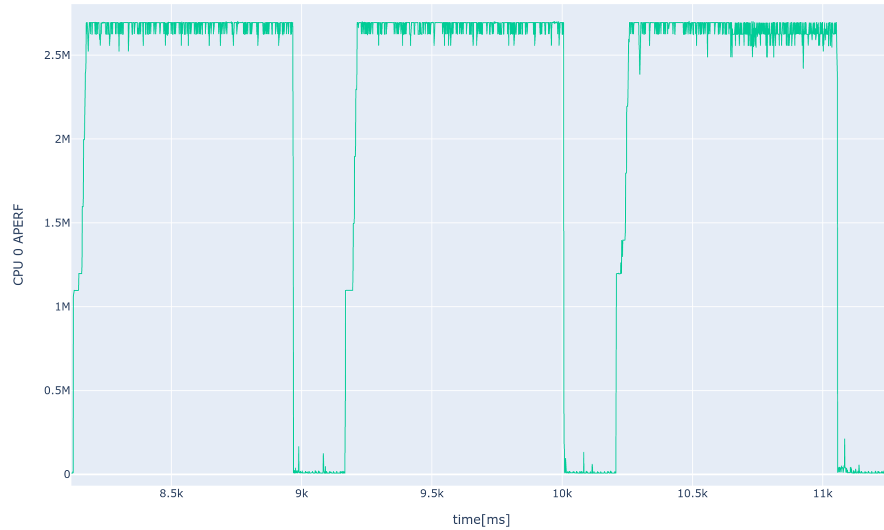

Figure 5.17: APERF values traced on CPU 0

respectively. For this reason, Figure 5.16 reports only one CPU of each pair

- the package sensor is the same sensor of CPU 0 (and CPU 2 for the aforementioned point)

Bibliography

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

[2] Intel® core™ i5-6200u processor. https://ark.intel.com/content/www/us/en/ark/products/88193/intel-core-i5-6200u-processor-3m-cache-up-to-2-80-ghz.html.

[3] jq. https://github.com/stedolan/jq.

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

[5] MSR. https://en.wikipedia.org/wiki/Model-specific_register. Model Specific Registers.

[6] msr-tools. https://github.com/intel/msr-tools.

[7] numpy. https://numpy.org/. The fundamental package for scientific computing with Python.

[8] P-state. https://www.intel.com/content/www/us/en/develop/documentation/vtune-help/top/reference/energy-analysis-metrics-reference/p-state.html. Intel User Guide, P-State.

[9] pandas. https://pandas.pydata.org/. pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.

[10] 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.

[11] plotly. https://plotly.com/. plotly.py is an interactive, open-source, and browser-based graphing library for Python.

[12] rt-app. https://github.com/scheduler-tools/rt-app.

[13] rt-app tutorial file. https://github.com/scheduler-tools/rt-app/blob/master/doc/tutorial.txt.

[14] Shebang. https://en.wikipedia.org/wiki/Shebang_(Unix). #!interpreter [optional-arg], in Unix systems.

[15] stap man page.

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

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

[18] Systemtap website. https://sourceware.org/systemtap/.

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

[20] turbostat. https://github.com/torvalds/linux/tree/master/tools/power/x86/turbostat/turbostat.c#L4640. turbostat - Report processor frequency and idle statistics.

[21] Wikichip. https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client), https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server). WikiChip is the preeminent resource for computer architectures and semiconductor logic engineering, covering historical and contemporary electronic systems, technologies, and related topics.

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

[23] Marco Cesati and Daniel P. Bovet. *Understanding the Linux Kernel, Third Edition*. O'reilly, 2005.

[24] Stefano Chiavazza. Linux scheduler internals: Resource management via cgroups. Master's thesis, University of Turin, July 2019.

[25] Winston Saunders Corey Gough, Ian Steiner. *Energy Efficient Servers: Blueprints for Data Center Optimization*. Apress, Berkeley, CA, 2015.

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

[27] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. Inside 6th-generation intel core: New microarchitecture codenamed skylake. *IEEE Micro*, 37(2):52–62, 2017.

[28] Rotem Efraim, Ran Ginosar, C. Weiser, and Avi Mendelson. Energy aware race to halt: A down to earth approach for platform energy management. *IEEE Computer Architecture Letters*, 13(1):25–28, 2014.

[29] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2.*

[30] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 4: Model-Specific Registers.*

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

[32] Efraim Rotem. Intel® architecture, code name skylake deep dive: A new architecture to manage power performance and energy efficiency. 2015. Efraim Rotem presentation at Intel Developer Forum 2015 edition.

[33] 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`.

[34] sysbench. `https://github.com/akopytov/sysbench`. sysbench GitHub page.

[35] SystemTap, Red Hat. *SystemTap Beginners Guide.*

[36] SystemTap, Red Hat. *SystemTap Tapset Reference Manual.*

[37] SystemTap, Red Hat, IBM, Intel. *SystemTap Language Reference.*

[38] Ken Thompson and Dennis Ritchie. Ken Thompson and Dennis Ritchie Explain UNIX (Bell Labs), about 1980. AT&T Bell Labs promotional film.

[39] Linus Torvalds. Torvalds about hybrid kernels. `https://www.realworldtech.com/forum/?threadid=65915%5C&curpostid=65936`.