SAPIENZA
UNIVERSITÀ DI ROMA

Energy efficient spin-locking in multi-core machines

Facoltà di INGEGNERIA DELL'INFORMAZIONE, INFORMATICA E STATISTICA
Corso di laurea in INGEGNERIA INFORMATICA - ENGINEERING IN COMPUTER SCIENCE - LM
Cattedra di Advanced Operating Systems and Virtualization

Candidato
Salvatore Riviecccio
1405255

Relatore
Francesco Quaglia

Correlatore
Pierangelo Di Sanzo

A/A 2015/2016

0 - Abstract

In this thesis I will show an implementation of spin-locks that works in an energy efficient fashion, exploiting the capability of last generation hardware and new software components in order to rise or reduce the CPU frequency when running spinlock operation. In particular this work consists in a linux kernel module and a user-space program that make possible to run with the lowest frequency admissible when a thread is spin-locking, waiting to enter a critical section. These changes are thread-grain, which means that only interested threads are affected whereas the system keeps running as usual. Standard libraries' spinlocks do not provide energy efficiency support, those kind of optimizations are related to the application behaviors or to kernel-level solutions, like governors.

Table of Contents

1 - Energy-efficient Computing

Given the arising demand for computing capacity great interest was shown in improving efficiency of energy usage in computing systems. There are a lot of definitions for energy efficiency but a good operational one is “using less energy to provide the same service”. Energy is the limiting resource in a huge range of computing systems, from embedded sensors to mobile phones to data centers. However, although it is a central topic, there is still a huge margin of improvement on which to work.

The greatest issue in some systems is probably how to measure energy efficiency. To address this compelled problem, computing elements of the latest generation, like CPUs, graphics processing units, memory units, network interface cards and so on have been designed to operate in several modes with different energy consumption levels. High-frequency performance monitoring and mode switching functions have also been exposed by co-designed abstract programming interfaces.

The actual challenge of energy-efficient computing is to develop hardware and software control mechanisms that take advantage of the new capabilities.

1.1 - TDP and Turbo Mode

The thermal design power (TDP) represents the amount of power the cooling system in a computer is required to dissipate. It means that a full-loaded system has to run under this “limit” if it wants to operate as manufacturer specific. The TDP and current energy consumption are a good metric to evaluate the energy efficiency of a system.

The TDP value as not to be intended as the maximum power that a processor can consume, in fact it is possible for the component (that can be a processor, a GPU and so on…) to consume more than the TDP power for a short period of time without it being thermally significant.

In multi-cores scenarios, the request of full performance on several cores simultaneously can easily exceed the TDP and in order to avoid this problem modern processors work with different frequencies, depending on the number of active cores and the maximum frequency declared by each core. For example, (this is the case) if some CPUs run at lowest frequency the system can provide more power on cores that demand full performance.

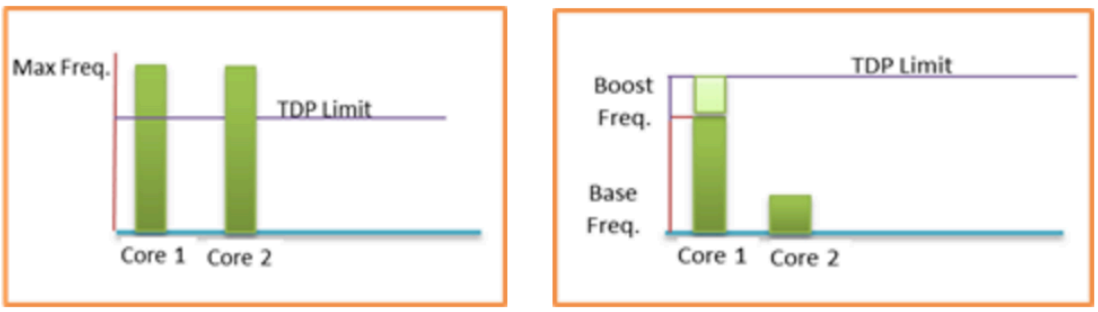


Figure 1: Example of TDP limit

Avoiding for the moment the problem of how to decide whether a core must turn at a minimum frequency, this solution is achieved by introduction of turbo mode, that on Intel architecture is called **Intel® Turbo Boost**[1].

This technology, born in 2008 on Nehalem microarchitecture, allows processor cores to run faster than their base frequency, if the operating condition permits. A Power Control Unit (PCU) firmware decides based on the:

- Number of cores active
- Estimated current consumption
- Estimated power consumption
- Processor temperature

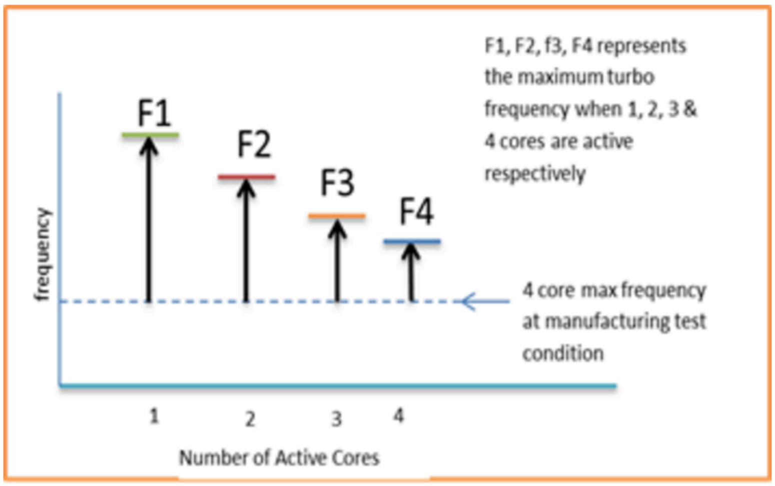

F1, F2, f3, F4 represents the maximum turbo frequency when 1, 2, 3 & 4 cores are active respectively
F1
F2
F3
F4
frequency
4 core max frequency at manufacturing test condition
1
2
3
4
Number of Active Cores

Figure 2: Frequency escalation

The PCU uses some internal models and counters to predict the actual and estimated power consumption.

1.2 - RAPL

In order to get better efficiency in energy usage it has been necessary to increase the efficiency of measurement methods on real systems.

Before Intel Sandy Bridge microarchitecture, decision on how to improve performance using Turbo-Boost was driven by models, which tends to be conservative, avoiding specific scenarios where the power consumption can be higher then TDP for too long and violating TDP specification. Sandy Bridge provides a new on-board power meter capability that makes it possible to do a better analysis and takes more accurate decisions. In addition all these new calculated metrics can be exported through a set of **Machine Specific Registers** (MSRs), this interface is called **RAPL**[2], Running Average Power Limit.

RAPL provides a way to set power limits on supported hardware and dynamically monitoring the power consumption of the system makes it possible to reassign power limits based on actual workloads.

In Linux RAPL is implemented as **power cap** driver from Kernel 3.13 and there are some utility softwares that help to get energy information:

- **TURBOSTAT** -- a linux kernel tool that is capable to display wattage information thought the usage of the MSRs.
- **POWERTOP** -- a solution that provides power consumption of CPU, GPU and DRAM components.
- **POWERSTAT** -- which measures the power consumption of a pc that has a battery power source or support for RAPL interface. The output also shows power consumption statistics. At the end of a run, powerstat will calculate the average, standard deviation and min/max of the gathered data.

1.3 - Combined Components

Another important solution from the point of view of energy efficiency which deserves to be mentioned is the **AMD FUSION APU**[3][4], Accelerated Processing Unit. It combines CPU and GPU in a single entity, trying to improve performance and minimize energy consumption. This has led to an emerging industry standard, known as the heterogeneous systems architecture (HSA). The net effect of HSA is to allow the CPU and GPU to operate as peers within the APU, dramatically reducing the energy overhead.

By keeping components as closely as possible **HBM**, a new type of memory chip with low power consumption, was introduced. The HBM graphics memory is a 3-D vertical stack connected to the GPU over a silicon carrier (2.5D packaging). The resulting silicon-to-silicon connection consumes more than three times less power than DDR5 memory and, beyond performance and power efficiency, HBM is capable to fit the same amount of memory in 94% less space.

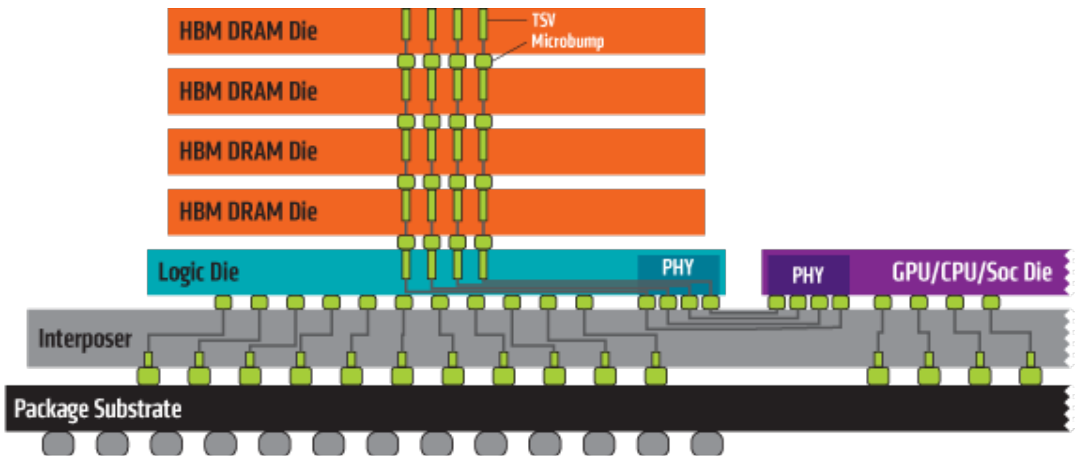


Figure 3: HBM

2 - The Linux Architecture

Let's now talk about the environment where this work has been done. A brief summary of the architecture will allow me to be more clear on my own solution. Linux[5] is a highly popular version of UNIX operating system. It is open source as its source code is freely available and editable, also it was designed considering UNIX compatibility. An operating system is actually a stack of software, each item designed for a specific purpose.

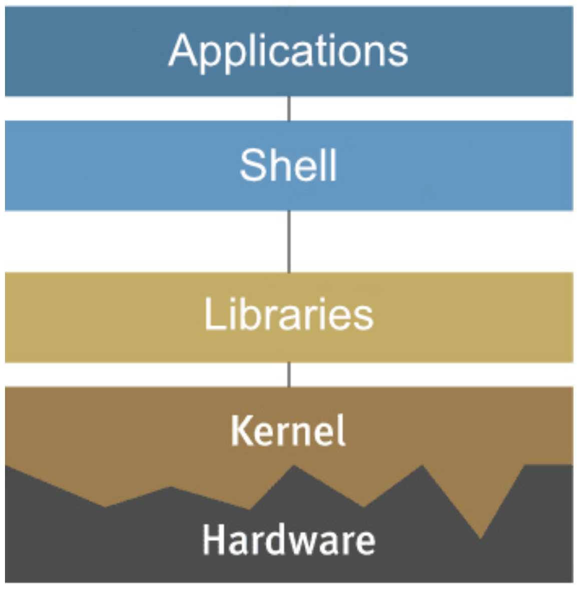


Figure 4: Linux stack

- **Kernel** -- It is the core of an operating system: it hides the complexity of device programming to the developer providing an interface for the programmer to manipulate hardware, manages communication between devices and software and manages the system resources (like CPU time, memory, network, ...).
- **System Libraries** -- They exposed methods for developers to write software for the operating system. We can, for example, demand for process creation and manipulation, file handling, network programming and so on, avoiding to

communicate with kernel directly. The library shields off the complexity of kernel programming for the system programmer and do not requires kernel module's code access rights.

- **System Utility** -- They are built using the system libraries and are visible to the end user, make him able to manipulate the system. They include methods for manage processes, navigate on the file system, execute other applications, configure the network and more.

2.1 - The Kernel

A kernel has generally four basic responsibilities:

- device management
- memory management
- process management
- handling system calls

When we talk about device management we need to consider that a computer system has connected several devices, not only the CPU and memory, but also I/O devices, graphic cards, network adapters and so on. Since each device operates differently from an other, the kernel needs to know what a device can do and how to interact with it. This information is maintained in the so-called device driver, without it the system is not capable of controlling any device.

In addition to drivers the kernel also manages the communication between the devices: there can be many shared components along all drivers and kernel rules the access in order to maintain system consistency. Most of the times communications follow strict rules without which there would be no guarantee on the quality of the communication.

Another very important thing about the kernel is the memory management. The kernel is responsible for keeping track of the memory areas used and not, to give memory space to a process who required it and to deny access to an unauthorized one. To do this, it uses the concept of virtual memory addresses:

It maps memory addresses used by a program, called virtual addresses, into physical addresses in computer memory.

This technique free the applications from having to manage a shared memory space, increase security due to memory isolation and makes possible to conceptually use more memory than might be physically available, using the technique of paging.
Different from user-space applications the kernel is always resident in main memory.

To ensure that each process gets enough CPU time, the kernel gives priorities to processes and gives each of them a certain amount of CPU time before it stops the process and hands over the CPU to the next in the priority queue. *Process management* not only deals with CPU time delegation (called scheduling), but also with security privileges, process ownership information, communication between processes and more.
The scheduler, the one who orchestrates the processes CPU time, is an essential component for the development of the low frequency spin-lock, we will enter after in the detail.

System calls are those components that allow the programmer to control (in a certain way) the kernel, to obtain information or to ask the kernel to perform a particular task. Obviously, a system call must be safe, for example it must not allow malicious code to run with kernel privilege.

2.2 - System Libraries

Even with a Kernel full of functionalities we cannot expect to do much without a way to invoke these features. This kind of triggers are mainly made by user applications, that must know how to place this system calls to Kernel (it is very system specific). Additionally, each kernel has a different set of supported system calls. Because of this, standards were created and each operating system declares to support these standards by implementing the specifications in its own way but keeping the exposed interface similar to other systems.

The most well-known system library for UNIX-like systems is the **GNU C Library**, namely **glibc**. It allows access to many important operations to the programmer, such as mathematical operations, input/output support, memory management and file operations. This allow us to write code that can be used on any system that supports such library.

So it is possible, without knowing kernel internals, to develop a software once, and then rebuild to many platforms.

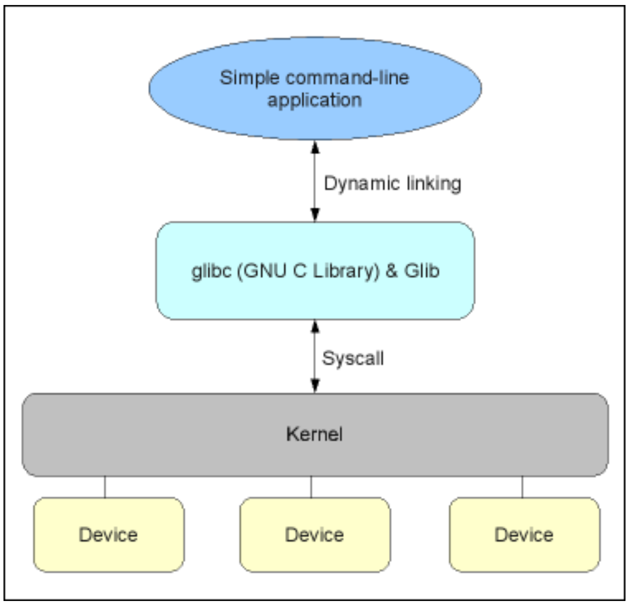


Figure 5: Stack of command calls

2.3 - System Tools

With a kernel and some programming libraries we cannot manipulate our system yet. We need access to commands, input we give to the system that gets interpreted and executed. System tools are all those things that allow us to do:

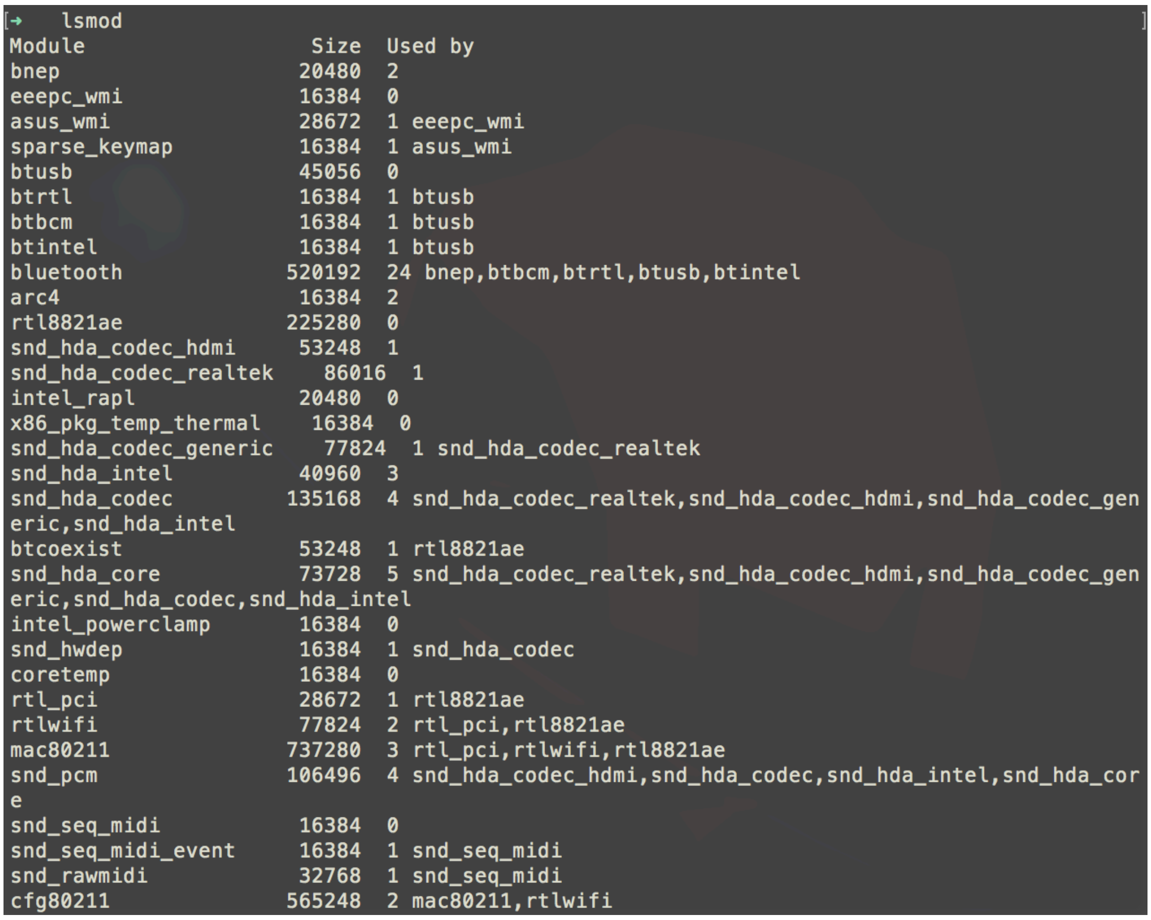
- file navigation: change directory, create/remove files, obtain file listings, ...
- information manipulation: text searching, compression, listing differences between files, ...
- process manipulation: launching new processes, getting process listings, exiting running processes, …
- privilege related tasks: changing ownership of files, changing user ids, updating file permissions, ...
- and more.

Because of the different implementations from a system to another (and not only for this) it is difficult to provide a solution that is energy efficient by remaining at the level of system library. We need to go deeper in the system.

3 - Into the Core

The solution I provide has been developed for Linux Kernel and takes advantage of several utility exposed by the system itself. At this level we are no longer system library programmer, we have to know kernel internals but the good thing is that we can “add” parts to the kernel in order to get new functionality.

Linux allows developers to create software modules that can be load and unload into the kernel upon demand, without the need to reboot the system or recompile the whole kernel. Working with modules is a common procedure, in fact we can see all the modules attached to kernel simply by running the command **lsmod**, which gets its information by reading the file /proc/modules.



```
➜  lsmod
Module                  Size  Used by
bnep                   20480  2
eeepc_wmi              16384  0
asus_wmi               28672  1 eeepc_wmi
sparse_keymap          16384  1 asus_wmi
btusb                  45056  0
btrtl                  16384  1 btusb
btbcm                  16384  1 btusb
btintel                16384  1 btusb
bluetooth             520192  24 bnep,btbcm,btrtl,btusb,btintel
arc4                   16384  2
rtl8821ae             225280  0
snd_hda_codec_hdmi     53248  1
snd_hda_codec_realtek    86016  1
intel_rapl             20480  0
x86_pkg_temp_thermal    16384  0
snd_hda_codec_generic    77824  1 snd_hda_codec_realtek
snd_hda_intel          40960  3
snd_hda_codec         135168  4 snd_hda_codec_realtek,snd_hda_codec_hdmi,snd_hda_codec_gen
eric,snd_hda_intel
btcoexist              53248  1 rtl8821ae
snd_hda_core           73728  5 snd_hda_codec_realtek,snd_hda_codec_hdmi,snd_hda_codec_gen
eric,snd_hda_codec,snd_hda_intel
intel_powerclamp       16384  0
snd_hwdep              16384  1 snd_hda_codec
coretemp               16384  0
rtl_pci                28672  1 rtl8821ae
rtlwifi                77824  2 rtl_pci,rtl8821ae
mac80211              737280  3 rtl_pci,rtlwifi,rtl8821ae
snd_pcm               106496  4 snd_hda_codec_hdmi,snd_hda_codec,snd_hda_intel,snd_hda_cor
e
snd_seq_midi           16384  0
snd_seq_midi_event     16384  1 snd_seq_midi
snd_rawmidi            32768  1 snd_seq_midi
cfg80211              565248  2 mac80211,rtlwifi
```

Figure 6: Example of lsmod output

/proc/ is a virtual filesystem, a software component that allows the operating system to access the file system through standard functions that are independent of the real file system or from the media used for storing data. It's sometimes referred to as a process information pseudo-file system, it means that doesn't contain 'real' files but runtime system information (e.g. system memory, hardware configuration, devices mounted, modules attached, etc).

Kernel modules need to be compiled a bit differently from regular user-space apps. Once again the Linux system comes to help, providing the kbuild system, a build process for external loadable modules that is fully integrated into the standard kernel build mechanism. It deals automatically with settings in sub-level Makefiles and at the end we can simply write in our personal makefile as follow,

```
obj-m += module.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

and run **make** command from the shell.

Once we get our module builded we can load it into the kernel with the command **insmod.**

modprobe instead is the clever version of insmod. insmod simply adds a module where modprobe looks for any dependencies (if that particular module is dependent on any other module) and loads them.

Differently from user-space programming, there is no main function in the modules. When a module is loaded the code from the init_module function is executed and similarly, when the module is unloaded, the code executed will be that of the cleanup_module function. It is therefore mandatory to integrate these functions into our own code or use other functions specified with module_init and module_exit calls respectively. All the code we have write in our module is now in the kernel, pending to be triggered.

Another difference is that we cannot make the include of the headers that we are usual to use: The reason is that the headers like stdio.h or stdlib.h are part of the standard C library and, as all system libraries, it is defined in user-space. The only external functions we are allowed to use are the ones provided by the kernel itself, the definition of all declared symbols is resolved upon insmod'ing.

“System libraries (such as glibc, libreadline, libproplist, whatever) that are typically available to user-space programmers are unavailable to kernel programmers. When a process is being loaded the loader will automatically load any dependent libraries into the address space of the process. None of this mechanism is available to kernel programmers: forget about ISO C libraries, the only things available is what is already implemented (and exported) in the kernel and what you can implement yourself.

Note that it is possible to "convert" libraries to work in the kernel; however, they won't fit well, the process is tedious and error-prone, and there might be significant problems with stack handling (the kernel is limited to a small amount of stack space, while user-space programs don't have this limitation) causing random memory corruption.

Many of the commonly requested functions have already been implemented in the kernel, sometimes in "lightweight" versions that aren't as featureful as their userland counterparts. Be sure to grep the headers for any functions you might be able to use before writing your own version from scratch. Some of the most commonly used ones are in include/linux/string.h.

Whenever you feel you need a library function, you should consider your design, and ask yourself if you could move some or all the code into user-space instead.’[6]

3.1 - The Ring Model

The kernel is largely focused on accessing and using resources. These resources are often also contended by the user space programs and the kernel must keep things tidy, without giving unconditional access when it is demanded. To ensure these types of access, a CPU can run in different modes. For each modes there is a degree of freedom which we can operate in the system. The Intel x86 architecture has 4 of these modes, which are called **rings**, but Unix-like systems mostly use only 2:

- Ring 0, that is the highest ring (it also known as "Supervisor mode")
- Ring 3, that is the lowest ring which is also called "user mode".

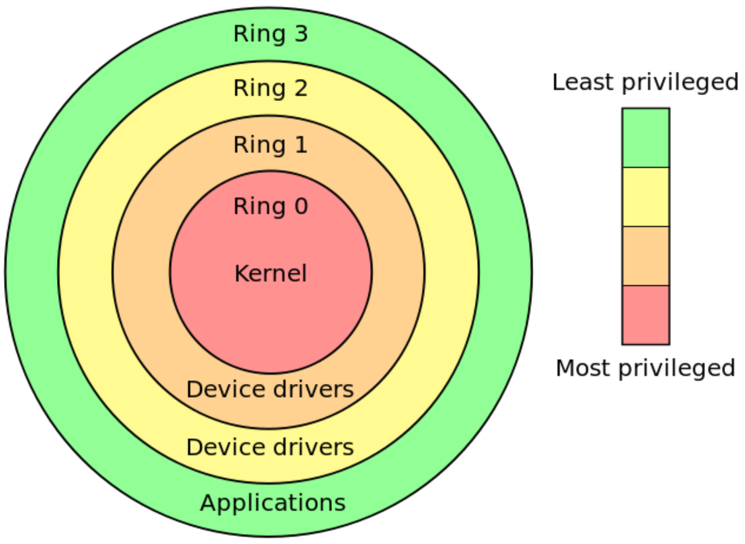


Figure 7: Ring model

Typically, we use a library function in user mode (ring 3). The library function calls can invoke one or more system calls, and these system calls execute on the library function's behalf. Since the system calls are part of the kernel itself they execute in supervisor mode and, once they have finished their tasks, they return and execution gets transferred back to user mode.

At this switching between user and kernel mode it is associated a high cost in performance. Historically it has been proven that a simple call as `getid` method has a cost of about 1000-1500 cycles on many types of machines. Of these just around 100 are used for the actual switch (70 from user to kernel space, and 40 back), the rest is "kernel overhead”.[7][8]

Functions are often moved through the rings in order to gain better performances. In fact, in Linux, we have an injection of vDSO sections in the application code where normally would be required a system call, i.e. a ring transaction. vDSO (virtual dynamically linked shared object) is a Linux kernel mechanism for exporting a carefully selected set of kernel space routines to user space applications. These functions use static data provided by the kernel preventing the need for a ring transition and granting a more lightweight procedure than a syscall (system call).

3.2 - Devices

So, keeping in mind the ring model, we must be aware to do not produce messy code when we are talking about kernel programming. Device drivers introduce an abstraction that allows the devices to be used without knowing internal details or vendor specification:

On UNIX, any hardware component is present in `/dev` folder as a device file, where it is kept the all the information about communication. The drivers (That is essentially a class of kernel module) might connect for example the file /dev/sda to the actual HD mounted on the system. A user-space program like gparted can read /dev/sda without ever knowing what kind of hard disk is installed.

Here an example of devices attached on a system:

```
➜   ls -l /dev/sda[1-5]

brw-rw---- 1 root disk 8, 1 dic 24 20:48 /dev/sda1
brw-rw---- 1 root disk 8, 2 dic 24 20:48 /dev/sda2
brw-rw---- 1 root disk 8, 5 dic 24 20:48 /dev/sda5
```

The pair of numbers in red are called major and minor number respectively. The major number tells which driver is used to access the hardware component. Each driver is assigned a unique major number; all device files with the same major number are controlled by the same driver. In this case all disk partitions are controlled by the driver associated with the number 8.

In order to distinguish between pieces of hardware minor number is used. For instance the 3 partitions are identified by the numbers 1,2 and 5.

The yellow ‘b’ at the start means that we are working with a **block device**. There are 2 types of devices: **block devices**, which are marked with char ‘b’ and **character devices**, which are marked by ‘c’. The difference is that block devices have a buffer for requests, so it can be possible to choose the best order in which to serve all the requests on the device. This is important in the case of storage devices like mechanical hard disk, where it's faster to read and write sectors which are close to each other. Another difference is that block devices can only work with blocks (whose size can differ according to the device), both as input and output, whereas character devices are allowed to use any size specified.

4 - Low Frequency Spin-lock

In this paragraph I will deeply explain the idea and the implementation of my energy efficient solution of spin-locks. In the end I'll show some measures taken on my system by using RAPL.

Let’s have now a little recap of what a spin-lock is and where to use.

4.1 - Spin-lock vs. Mutex

When we talk about spin-locks and mutex we are talking about critical section: We are interested in one or more shared resources, but someone else in the system will contend those resources. To keep the system consistent we need to serialize access to resources and to grant that every process will access to a demanded resource eventually. For this purpose there are two main approaches that are the aforementioned spin-lock and mutex. The former is a mechanism in which the process that needs a resource polls the lock on resource until it gets it. It is also called as busy-waiting or active-waiting. Process will be busy in a loop till it gets the resource. The latter instead puts the requesting processes (and which have not have been granted the resources) in a waiting queue, releasing system resources as for example CPU time.

So the question that we must answer is where to use one or the other.

- Spin-locks are best used when a piece of code cannot go to sleep state like Interrupt service routines or in general Kernel code.
- Mutexes are best used in user space program where a sleeping process does not mean a performance degradation, or at least not significantly.

Criteria	Mutex	Spinlock
Mechanism	Test for lock. If available use the resource. If not goes to wait queue.	Test for lock. If available use the resource. If not, loop again and test the lock until it gets the lock.
When to use	Used when putting process to sleep is not harmful like user space programs. Used when there will be considerable time before process gets the lock.	Used when process should not be put in sleep like interrupt service routines. Used when lock will be granted in reasonably short time.
Drawbacks	Incurs process context switch and scheduling cost.	Processor is busy doing nothing until lock is granted, wasting CPU cycles.

Table 1: mutex vs. spinlock

4.2 - low_freq_module

Tinkering with kernel stuff is not something that can be done in a simple application and changing processor frequencies is one of these things. My actual implementation concerns a userspace implementation of the spin-lock that interacts with a module kernel, called `low_freq_module`. The kernel module passes through the usage of some pseudo files that are located in sub-directories down in the **`/sys`** folder. **`/sys`** is where is attached the virtual file-system **`sysfs`** that provides a means to export kernel data structures, their attributes, and the linkages between them to userspace.

```
UbuntuServer: /sys/devices/system/cpu
➜   la
totale 0
drwxr-xr-x 8 root root    0 dic 27 16:10 cpu0
drwxr-xr-x 8 root root    0 dic 27 16:10 cpu1
drwxr-xr-x 4 root root    0 dic 27 16:10 cpufreq
.
.
.
```

As we can se there is one folder for each CPU; my server is equipped with an Intel Celeron 1007U consisting of two cores so we have cpu0 and cpu1.

Going down in the sub-folders (for example cpu0) we can find several files that hold useful information about frequencies and not only. Here are listed those of interest with `la` command:

```
UbuntuServer: /sys/devices/system/cpu/cpu0/cpufreq
➜   la
.
.
.
-r-------- 1 root root 4,0K dic 27 16:37 cpuinfo_cur_freq
-r--r--r-- 1 root root 4,0K dic 27 16:37 cpuinfo_max_freq
-r--r--r-- 1 root root 4,0K dic 27 16:37 cpuinfo_min_freq
-r--r--r-- 1 root root 4,0K dic 27 16:37 scaling_cur_freq
-rw-r--r-- 1 root root 4,0K dic 24 20:49 scaling_governor
-rw-r--r-- 1 root root 4,0K dic 27 16:37 scaling_max_freq
-rw-r--r-- 1 root root 4,0K dic 27 16:37 scaling_min_freq
-rw-r--r-- 1 root root 4,0K dic 27 16:37 scaling_setspeed
.
.
.
```

`cpuinfo-max-freq` and **`cpuinfo-min-freq`** hold respectively information about the maximum and the minimum frequency admissible in the system.

In **`scaling_governor`** file we can specify the actual governor that runs on the system; a governor is the kernel component that is responsible for determining what frequency policy should be followed. The most used governors are:

- **OnDemand** -- is the default choice on most systems, balanced settings that offer a good compromise: energy consumption and performance.
- **Powersave** -- as the name suggests with powersave is set both the maximum and the minimum frequency to the lowest possible value.
- **Userspace** -- it allows to manually set frequencies.
- **Conservative** -- it is like OnDemand, i.e. sets a minimum and maximum frequency with a time when reaching such limits. The only difference is that

conservative reaches the limit in a double time compared to OnDemand being the type of ramp less leaning. The advantage however affects the battery life. In fact more leaning is the ramp (i.e. faster, tending upward) more we will have a battery consumption.

- **Performance** -- is similar to PowerSave with the difference that sets the maximum clock frequency for both minimum and maximum CPU working frequencies.

Writing on the **`scaling_setspeed`** file we can specify any frequency, as long as it is between the minimum and the maximum and that the userspace governor is set.

`scaling_min_freq` and **`scaling_max_freq`** keep information about the minimum and maximum frequency respectively and specify the frequency range on which the governor can operate, lowering down to the minimum and raising up to the maximum. The difference with the first two files is that this is a current (and non-absolute) measure of the system. They can be specified regardless of the running governor.

So in the `init_module` function of my module I keep references to some of these files: cpuinfo_min/max_freq and scaling_min/max_freq for each CPU. It is done with the Virtual File System (VFS) operations, like `vfs_write`, `vfs_read` and so on. This is a kernel way (but not the only) for doing with files on the system (remember that we have no access to library function calls like `fopen/ fwrite…`). The purpose of a VFS is to allow a program to access different types of file systems in a uniform way; it specifies an interface (or a "contract") between the kernel and a concrete file system.

My approach scales to any number of cores and is configured to work even with heterogeneous CPUs, (which have different frequency specifications). The variable **`n_proc`** (number of cores) is passed through the module parameters mechanism that makes possible to modules to take command line arguments. To allow arguments to be passed to my module, I declare the variables that will keep the values of the command line arguments as global and then I use the `module_param()` macro, which is defined in `linux/moduleparam.h`. At runtime, insmod fills the variables with any command line arguments that are given, like **`insmod mymodule.ko myvariable=5`**.

Another parameter that i pass to the module is the maximum number of threads admissible for a given system; all numbers between 0 and `max_thread` are also thread identifiers (TIDs). I store these IDs in an array of int, where the index corresponds to the identifier itself, so to have constant access to the structure. When we get the TID we need to distinguish the cases in which we do it in kernel or user space: in fact from the kernel point of view there are only PIDs (process identifier) and TGID (thread group id) to which threads of a process belongs to. User space instead a call for getting the PID of a thread would simply return the TGID.

Here an example of how it works:

```
                     USER VIEW
 <-- PID 55 --> <----------------- PID 54 ----------------->
                     +---------+
                     | process |
                    _| pid=54  |_
                  _/ | tgid=54 | \_ (new thread) _
       _ (fork) _/   +---------+                  \
      /                                        +---------+
+---------+                                    | process |
| process |                                    | pid=56  |
| pid=55  |                                    | tgid=54 |
| tgid=55 |                                    +---------+
+---------+
 <-- PID 55 --> <--------- PID 54 --------> <--- PID 56 --->
                     KERNEL VIEW
```

This identifier is used to distinguish the processes that operate with low frequency spin-locks from the others: In fact, the value in the array will be 1 when it is one of the processes involved and 0 otherwise.

There is also a bit-mask that marks which are the under-clocked cores; in this way, I do not have to set again the frequency down if two involved processes enter in the same core subsequently. For that purpose the kernel offers a set of functions that modify or test single bits atomically. Because the whole operation happens in a single step, no interrupt (or other processor) can interfere. Atomic bit operations are very fast, since they perform the operation using a single machine instruction without disabling interrupts and keeping coherence across processors.

The last thing that I do in `init` is to register the driver of char device, this is useful to interact with the user-space application.

4.3 - Char Device

The operations on the char device are defined in the `file_operation`[9] structure; it holds pointers to functions defined by the driver that perform various operations. This structure is something like this:

```c
struct file_operations
{
    struct module *owner;
    loff_t(*llseek) (struct file *, loff_t, int);
    ssize_t(*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t(*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
    ssize_t(*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t(*aio_write) (struct kiocb *, const char __user *, size_t,
                             loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int,
                        unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t(*readv) (struct file *, const struct iovec *, unsigned long,
                           loff_t *);
    ssize_t(*writev) (struct file *, const struct iovec *, unsigned long,
                            loff_t *);
    ssize_t(*sendfile) (struct file *, loff_t *, size_t, read_actor_t,
                              void __user *);
    ssize_t(*sendpage) (struct file *, struct page *, int, size_t,
                              loff_t *, int);
    unsigned long (*get_unmapped_area) (struct file *, unsigned long,
                                            unsigned long, unsigned long,
                                            unsigned long);
};
```

We can specify all these operations as needed. However, some operations may not be implemented by a driver. For example, a driver that handles a keyboard won't need to write in a structure in general. The corresponding entries in the file_operations structure should be set to `NULL`.

We can easily make assigning to the structure in the following way:

```c
struct file_operations fops = {
    .read = device_read,
    .write = device_write,
    .open = device_open,
    .release = device_release
};
```

As discussed in the previous sections, char devices are accessed through device files, usually located in `/dev`. The major number specifies which driver is associated while the minor number is used only by the driver itself to differentiate which device it's operating on.

Adding a driver to a system means registering it within the kernel. This is done by assigning it a major number during the init_module call. We can do this by using the `register_chrdev` function that takes 3 arguments:

- **unsigned int major** -- is the major number we want to request, if 0 is passed it's handled automatically by the kernel.
- **const char *name** -- is the name of the device.
- **struct file_operations *fops** -- is a pointer to the `file_operations` table for driver we specify.

The return value of this function is an integer that is the major number assigned, or a negative value in case of failure.

The call `unregister_chrdev` handles the effective unregistering of the driver from the kernel.

In the drivers of my module I have implemented only the `open`, `release` and `ioctl` functions. `open` and `release` are useful to keep information about the current open channels on the device.

Most physical and non-physical devices are used for output as well as input, but sometime it is not always enough or it is not what a device should do. Imagine we had a serial port connected to a modem (even if we have an internal modem, it is still implemented from the CPU's perspective as a serial port connected to a modem). What we expect the `write` function does is to send commands or data through the phone line, while the `read` function allows to receive things from the modem, either responses to commands or the data. However, this doesn't solve the problem of what to do when we need to configure the serial port itself, for example to set the rate at which data is sent and received. Unix-like systems provide a special function called `ioctl` (short for Input Output ConTroL). Every device can have its own `ioctl` commands, which can be read ioctl's (to send information from a process to the kernel) and/ or write ioctl's (to return information to a process). The `ioctl` function is called with 3 parameters:

- The device descriptor file.
- The ioctl number.
- A long integer that we can cast to use it to pass anything.

The `ioctl` number encodes the major device number, the type of the `ioctl`, the command, and the type of the parameter. This `ioctl` number is usually created by using a macro call (`_IO`, `_IOR`, `_IOW` or `_IOWR` --- depending on the type) in a header file.

In the `low_freq_module` the `ioctl` function is used by the user-space application to communicate to the kernel to set and reset CPU frequency. The 2 main commands are the LFM_SET_TID and LFM_UNSET_TID where the function does:

- Retrieves the TID of the requesting process from the third parameter of `ioctl` call and populates correctly the TIDs array mentioned before.
- Set/Clear the bit associated to the performing core respectively.
- Set/Reset the core frequency respectively.

4.4 - The schedule problem

Unfortunately, the methodologies presented do not cope with all possible scenarios. Let's imagine that a process is queued to get into the critical section and, before start to spin-locking, request to set the frequency of the core on which it is running to minimum. Eventually, if it does not enter the critical section in its quantum, it will be de-scheduled leaving the core without resetting the proper frequency and again, it may happen that the same process will be scheduled on another core without benefiting of spin-locking at low frequencies.

We need an internal component that traces the processes that are going to get into the CPU; this component is the Scheduler.

The **Completely Fair Scheduler** (CFS) is a process scheduler which was merged into the 2.6.23 release of the Linux kernel in the October 2007 and it is now the default scheduler. It handles CPU resource allocation for executing processes, and aims to maximize overall CPU utilization while also maximizing interactive performance.

In order to get the new features we needed we have to patch the scheduler, and this was done thanks to the work carried out by Professor Francesco Quaglia and Alessandro Pellegrini in the **schedule-hook** module[10]. This is a kernel module that dynamically patches the Linux kernel scheduler (without restarting or recompile the whole kernel) so as to allow running a custom function upon thread reschedule. For instance it retrieves the memory position of the corresponding machine instructions block from the system-map (typically available in Linux installations from the /boot directory of the root file system), and injects into this routine an execution flow variation such that control goes to a schedule hook() routine offered by the external module right before schedule() would execute its finalization part (e.g. stack realignment and return).

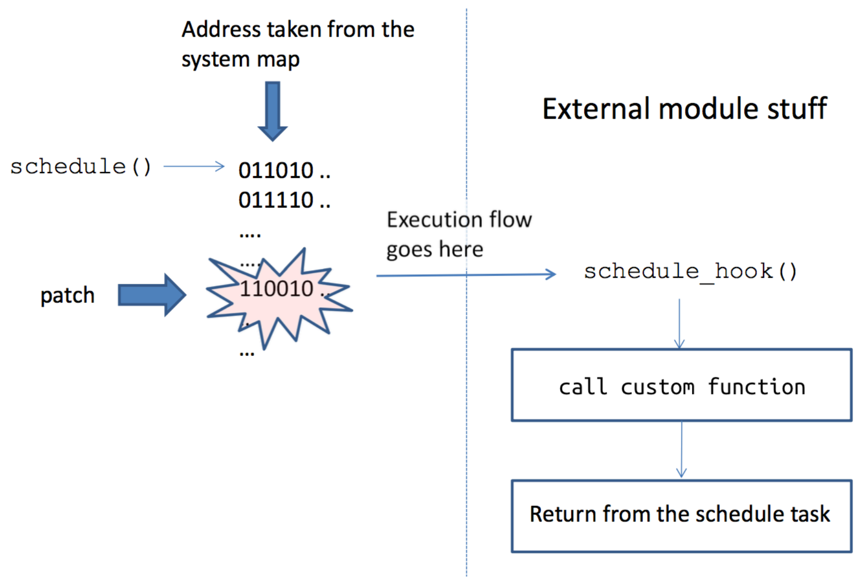


Figure 8: schedule-hook()

This module can be used either in cross compilation with the one containing the custom function or not.

The schedule-hook function is embedded within this module, it checks the value of a function pointer and in case it is not null the target function is called.

In cross compilation the function pointer is exported as a symbol to be updated while mounting the module containing the custom functions to be run otherwise, the function pointer is accessible as a pseudo-file called:

`/sys/module/schedule_hook/parameters/the_hook.`

In the latter configuration, after mounting the module, we can load any function pointer we would like (pointing to kernel stuff) by writing it on that pseudo-file.

The function pointer is the one associated with the `on_schedule` function present within the `low_freq_module`. This function takes the ID of the core on which the scheduler is running and the TID of the next process that will be scheduled by using the `task_struct current`. The `task_struct` is the data structure that describes a process or task in the system; in particular, there is the `pid` field (an integer) that keeps the process identifier. Then the function proceeds with tests to check in what state is the core and who is the next process to be scheduled.

If the process is one of those who use the low frequency spin-lock (it is registered within the array) and the core has not been lowered, then it is written to the **`scaling_max_freq`** file the minimum frequency; in this way the governor is able to use only the specified frequency.

If the process isn't registered and the core is still lowered then it proceeds to restore the normal system behavior.

If neither of these two cases occurs then it simply returns control to the `schedule-hook()` function.

4.5 - low_freq_spinlock implementation

My implementation of spin-locks is based essentially on a structure containing a volatile integer. This keyword (volatile) prevents an optimizing compiler from optimizing away subsequent reads or writes and thus incorrectly reusing a stale value or omitting writes. Practically it forces the flush to the main memory of the variable, ensuring that the read version is always the most up-to-date.

The struct is renamed `low_freq_spinlock_t` by using the keyword `typedef`, which we can use to give a type, a new name.

The implementation resembles the one of the original spinlocks; in fact, when we include the linux/spinlock.h header, we can declare and initialize a spinlock in this way:

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
```

or at runtime with:

```
spin_lock_init(spinlock_t *lock);
```

And similarly we can do with my implementation:

```
low_freq_spinlock_t my_lock = LOW_FREQ_UNLOCKED;
```

Or:

```
low_freq_init(low_freq_spinlock_t *lock);
```

Once the spinlock is initialized, the phases of lock and unlock can be invoked via the `void low_freq_op_lock(low_freq_spinlock_t *)` and `void low_freq_op_unlock(low_freq_spinlock_t *)` functions, just like normal spinlocks.

```c
void low_freq_op_lock(low_freq_spinlock_t *lock)
{
  if(__sync_lock_test_and_set(&(lock)->exclusion, 1))
  {
    set_low_freq();
    while(__sync_lock_test_and_set(&(lock)->exclusion, 1))
    {
      while ((lock)->exclusion)
      {
        ;
      }
    }
    reset_low_freq();
  }
}

void low_freq_op_unlock(low_freq_spinlock_t *lock)
{
  __sync_synchronize(); // Memory barrier.
  (lock)->exclusion = 0;
}
```

In the lock function there is a check on the volatile integer using the function `__sync_lock_test_and_set(&(lock)->exclusion, 1)`, that simply puts 1 in the lock and return the previous value.

This function is translated into less machine instructions than other synchronization solutions, like for example the `compare_and_swap`.

There is a small optimization in the inner loop that involves controlling the only variable of lock without synchronization, making the execution even lighter.

If the previous value of lock was 1 it means that the lock has already been taken.

The process then, in the `set_low_freq()` function, asks for its own TID by calling `syscall(__NR_gettid)` function, passing it to the `low_freq_module` via the `ioctl` call that we saw before and asking to lower the frequency.

```c
void inline set_low_freq()
{
  int tid = syscall(__NR_gettid);
  char s_tid[32];
  sprintf(s_tid, "%d", tid);
  char *argv[3] = {"ioctl", s_tid, "-s"}; //param: name, tid, mode
  ioctl_call(3, argv);
}
```

In Linux, there's a system call that will return a TID of calling thread. The name of the system call is gettid(). For some reason it is not implemented in glibc. Probably it is because it is a Linux specific system call. Anyway we have to use syscall() with the proper number that we can easily retrieve with the macro `__NR_gettid`.

The process then starts the phase of spinlock as long as the lock is not released.

Before entering the critical section the reset command is invoked using the module's `ioctl` function (similarly to the lock phase) and finally returns.

```c
void inline reset_low_freq()
{
  int tid = syscall(__NR_gettid);
  char s_tid[32];
  sprintf(s_tid, "%d", tid);
  char *argv[3] = {"ioctl", s_tid, "-u"}; //param: name, tid, mode
  ioctl_call(3, argv);
}
```

4.6 - ioctl

The actual link between user-space application and the module is made in the `ioctl.c` file.

The function that exposes the `ioctl` feature is `ioctl_call`. This function takes 2 arguments: An integer that stands for the number of parameters and an array of 3 strings (the parameters). The first string is the name of the function, ‘`ioctl`’ for instance, the second one is the TID of the process and the third is the command that we want to perform.

At the beginning of the function a file descriptor is opened to `/dev/low_freq_module`, the char device of my module, and there is a check on the third string passed as parameter:

- If the string is ‘-s’ it means that the command to be invoked is LFM_SET_TID,
- if the string is ‘-u’ then the performed command will be LFM_UNSET_TID,
- if ‘-is’ is passed the LFM_IS_PRESENT command will return 1 or 0 according to the fact that the TID is present in the TIDs array or not. Actually this command is never called.
- If the command is none of these, an error will be reported to the user.

In order to get my implementation work we have to include the `low_freq_spinlock.c` and `ioctl.c` files in the programs that want to use it.

5 - Measurements

To get more robust results I tried different kernel versions and ran the tests on both Intel and AMD machines. Thanks to RAPL support in the most recent kernels, I have managed to have precise measurements of energy consumption. One aspect that induced me to do different tests is that on some Intel architectures we can not change the CPU-core frequencies individually: If we want to turn down or raise the frequency of a core all the socket will be affected.

5.1 - Intel measures

The following measures were made on my own server that has the following hardware:

```
CPU:
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                2
On-line CPU(s) list:   0,1
Thread(s) per core:    1
Core(s) per socket:    2
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 58
Model name:            Intel(R) Celeron(R) CPU 1007U @ 1.50GHz
Stepping:              9
CPU max MHz:           1500,0000
CPU min MHz:           800,0000
BogoMIPS:              2993.26
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              2048K
RAM:
MemTotal:              3945408 kB
MemFree:               472172 kB
MemAvailable:          3006196 kB
KERNEL:
Linux 4.4.0-53-generic x86_64
```

I initially measured the system with `powerstat` to see its consumption, both in normal mode and with lowered frequencies.

```
Normal Mode:

  Time    User  Nice   Sys  Idle    IO  Run Ctxt/s  IRQ/s  Watts
17:35:00   1.0   0.0   0.0  99.0   0.0    1    161     72   2.51
17:35:01   0.5   0.0   0.0  99.5   0.0    1    107     63   2.33
17:35:02   0.0   0.0   0.5  99.5   0.0    1    117     68   2.33
17:35:03   0.0   0.0   0.0 100.0   0.0    1    126     76   2.33
17:35:04   0.0   0.0   0.0 100.0   0.0    1    108     65   2.33
17:35:05   0.0   0.0   0.0 100.0   0.0    1    119     71   2.33
17:35:06   0.0   0.0   8.0  92.0   0.0    1    304    205   2.65
17:35:07   0.0   0.0   0.0 100.0   0.0    1    155     96   2.34
17:35:08   0.0   0.0   0.0 100.0   0.0    1    121     67   2.34
17:35:09   0.0   0.0   0.0 100.0   0.0    1    187     97   2.35
17:35:10   1.5   0.0   0.5  98.0   0.0    1    136     90   2.37
17:35:11   0.0   0.0   0.5  99.5   0.0    1    171     93   2.35
-------- ----- ----- ----- ----- ----- ---- ------ ------ ------
 Average   0.1   0.0   0.2  99.7   0.0  1.0  118.7   70.6   2.34
  StdDev   0.3   0.0   1.0   1.1   0.2  0.0   30.4   20.2   0.05
-------- ----- ----- ----- ----- ----- ---- ------ ------ ------
 Minimum   0.0   0.0   0.0  92.0   0.0  1.0   88.0   53.0   2.32
 Maximum   1.5   0.0   8.0 100.0   1.5  1.0  304.0  205.0   2.65
-------- ----- ----- ----- ----- ----- ---- ------ ------ ------
Summary:
CPU:   2.34 Watts on average with standard deviation 0.05

Low Frequency Mode:

  Time    User  Nice   Sys  Idle    IO  Run Ctxt/s  IRQ/s  Watts
17:52:49   0.0   0.0   0.0 100.0   0.0    1    112     67   2.33
17:52:50   0.0   0.0   0.0 100.0   0.0    1    102     61   2.32
17:52:51   0.5   0.0   0.0  99.5   0.0    1    117     73   2.33
17:52:52   0.0   0.0   0.0 100.0   0.0    1    100     59   2.33
17:52:53   0.0   0.0   0.0 100.0   0.0    1    108     65   2.33
17:52:54   0.5   0.0   0.5  99.0   0.0    1    164     90   2.35
17:52:55   0.0   0.0   0.0 100.0   0.0    1    105     72   2.33
17:52:56   0.0   0.0   0.5  99.5   0.0    1    174    106   2.35
17:52:57   0.0   0.0   8.1  91.9   0.0    2    233    167   2.52
17:52:58   0.0   0.0   0.0 100.0   0.0    1    233    142   2.36
17:52:59   0.0   0.0   0.0  98.5   1.5    1    146     87   2.34
17:53:00   0.5   0.0   0.0  99.5   0.0    1    181     90   2.35
-------- ----- ----- ----- ----- ----- ---- ------ ------ ------
 Average   0.1   0.0   0.2  99.7   0.1  1.0  121.9   74.0   2.33
  StdDev   0.2   0.0   1.0   1.1   0.2  0.1   29.1   21.0   0.03
-------- ----- ----- ----- ----- ----- ---- ------ ------ ------
 Minimum   0.0   0.0   0.0  91.9   0.0  1.0   90.0   50.0   2.32
 Maximum   1.0   0.0   8.1 100.0   1.5  2.0  233.0  167.0   2.52
-------- ----- ----- ----- ----- ----- ---- ------ ------ ------
Summary:
CPU:   2.33 Watts on average with standard deviation 0.03
```

The fields have the following meaning:

1. **Time** -- the startup time of each monitoring instance.
2. **User** -- CPU usage of processes initiated by the current user.
3. **Nice** -- a special value (Kernel function) that prioritize the CPU time for applications. Depending on the “importance” of the process, the “nice” value changes, giving more or less CPU time for the process.
4. **Sys** -- CPU usage for the system software, such as CPU time used by the Kernel for instance.
5. **Idle** -- represent the “waiting percentage” of your CPU.
6. **IO** -- This refers to IO Wait.
7. **Run** -- Under this it shows the currently running processes.
8. **Ctxt/s** -- context switch rate.
9. **IRQ/s** = IRQ per second -- IRQ is a special signal (I/O technique) that hardware devices use to communicate with the CPU.
10. **Watts** -- energy consumption.

As we can see the energy consumption (in watts) of CPU are practically identical, this means that the system is most of the time in idle.
In this way I am sure that when I run measuring tools, all the difference in energy consumption will be due to my tests.

In the following 2 tests I used 3 threads, 1 that goes into critical section for indeterminate time and 2 that continue to spinlock on the two cores. The first test was done with a classical implementation of the spinlock and the second with the `low_freq_spinlock`.

normal_spinlock:

```
  Time     User  Nice   Sys  Idle    IO  Run Ctxt/s  IRQ/s  Watts
18:29:13 100.0   0.0   0.0   0.0   0.0    3     71    529   6.04
18:29:14 100.0   0.0   0.0   0.0   0.0    3     61    521   6.03
18:29:15  89.9   0.0   7.5   2.5   0.0    3    272    577   5.92
18:29:16 100.0   0.0   0.0   0.0   0.0    3    107    531   6.02
18:29:17 100.0   0.0   0.0   0.0   0.0    3    104    532   6.02
18:29:18 100.0   0.0   0.0   0.0   0.0    3    130    533   6.03
18:29:19 100.0   0.0   0.0   0.0   0.0    3     70    527   6.03
18:29:20 100.0   0.0   0.0   0.0   0.0    3     61    518   6.02
18:29:21 100.0   0.0   0.0   0.0   0.0    3     66    525   6.03
18:29:22 100.0   0.0   0.0   0.0   0.0    3     71    522   6.04
18:29:23 100.0   0.0   0.0   0.0   0.0    3     72    527   6.02
18:29:24 100.0   0.0   0.0   0.0   0.0    3     79    524   6.00
18:29:25 100.0   0.0   0.0   0.0   0.0    3     58    523   6.02
18:29:26 100.0   0.0   0.0   0.0   0.0    3    150    525   6.03
18:29:27 100.0   0.0   0.0   0.0   0.0    3     91    538   6.03
18:29:28 100.0   0.0   0.0   0.0   0.0    3     56    515   6.02
-------- ----- ----- ----- ----- ----- ---- ------ ------ ------
 Average  99.8   0.0   0.1   0.0   0.0  3.0   79.6  528.8   6.03
  StdDev   1.3   0.0   1.0   0.3   0.0  0.0   38.3   11.0   0.02
-------- ----- ----- ----- ----- ----- ---- ------ ------ ------
 Minimum  89.9   0.0   0.0   0.0   0.0  3.0   45.0  514.0   5.92
 Maximum 100.0   0.0   7.5   2.5   0.0  3.0  272.0  577.0   6.05
-------- ----- ----- ----- ----- ----- ---- ------ ------ ------
Summary:
CPU:   6.03 Watts on average with standard deviation 0.02
```

low_frew_spinlock:

```
  Time     User  Nice   Sys  Idle    IO  Run Ctxt/s  IRQ/s  Watts
18:22:57 100.0   0.0   0.0   0.0   0.0    3    170    558   4.17
18:22:58 100.0   0.0   0.0   0.0   0.0    3    102    529   4.17
18:22:59 100.0   0.0   0.0   0.0   0.0    3    126    536   4.17
18:23:00 100.0   0.0   0.0   0.0   0.0    3     95    530   4.18
18:23:01 100.0   0.0   0.0   0.0   0.0    3     87    527   4.17
18:23:02 100.0   0.0   0.0   0.0   0.0    3     66    528   4.17
18:23:03 100.0   0.0   0.0   0.0   0.0    3     54    519   4.17
18:23:04 100.0   0.0   0.0   0.0   0.0    3     74    533   4.17
18:23:05 100.0   0.0   0.0   0.0   0.0    4     55    516   4.18
18:23:06 100.0   0.0   0.0   0.0   0.0    3     55    524   4.18
18:23:07 100.0   0.0   0.0   0.0   0.0    3     85    522   4.18
18:23:08 100.0   0.0   0.0   0.0   0.0    3     87    536   4.17
18:23:09 100.0   0.0   0.0   0.0   0.0    3     59    517   4.17
18:23:10 100.0   0.0   0.0   0.0   0.0    3     66    527   4.16
18:23:11 100.0   0.0   0.0   0.0   0.0    3     55    517   4.17
18:23:12 100.0   0.0   0.0   0.0   0.0    3     59    527   4.17
-------- ----- ----- ----- ----- ----- ---- ------ ------ ------
 Average  99.9   0.0   0.1   0.0   0.0  3.0   69.8  524.2   4.17
  StdDev   1.0   0.0   1.0   0.0   0.0  0.2   24.0    8.2   0.01
-------- ----- ----- ----- ----- ----- ---- ------ ------ ------
 Minimum  92.0   0.0   0.0   0.0   0.0  3.0   47.0  513.0   4.16
 Maximum 100.0   0.0   8.0   0.0   0.0  4.0  175.0  558.0   4.19
-------- ----- ----- ----- ----- ----- ---- ------ ------ ------
Summary:
CPU:   4.17 Watts on average with standard deviation 0.01
```

Cores are all running (idle = 0) in user mode and there are exactly 3 threads: 1 sleeping and 2 spinlocking. On my hardware configuration the execution with the low frequency spinlock has 1.86 Watts less then the one with the normal spinlock implementation, with energy savings of about 30%.

Obviously here we are not considering the performance or the length of the sleep phase, i.e. the critical section. Let's see now more tests more in detail. The following tests initially create 2 threads, then keep spawning another one continuously as soon as one finishes in order to get always one in c.s. (critical section) and one spinlocking. It keeps spawning a number of threads as is required for a 90-second running execution based on the length of the sleep, i.e. if we have a sleep time of 10 sec we need 9 threads and so on.

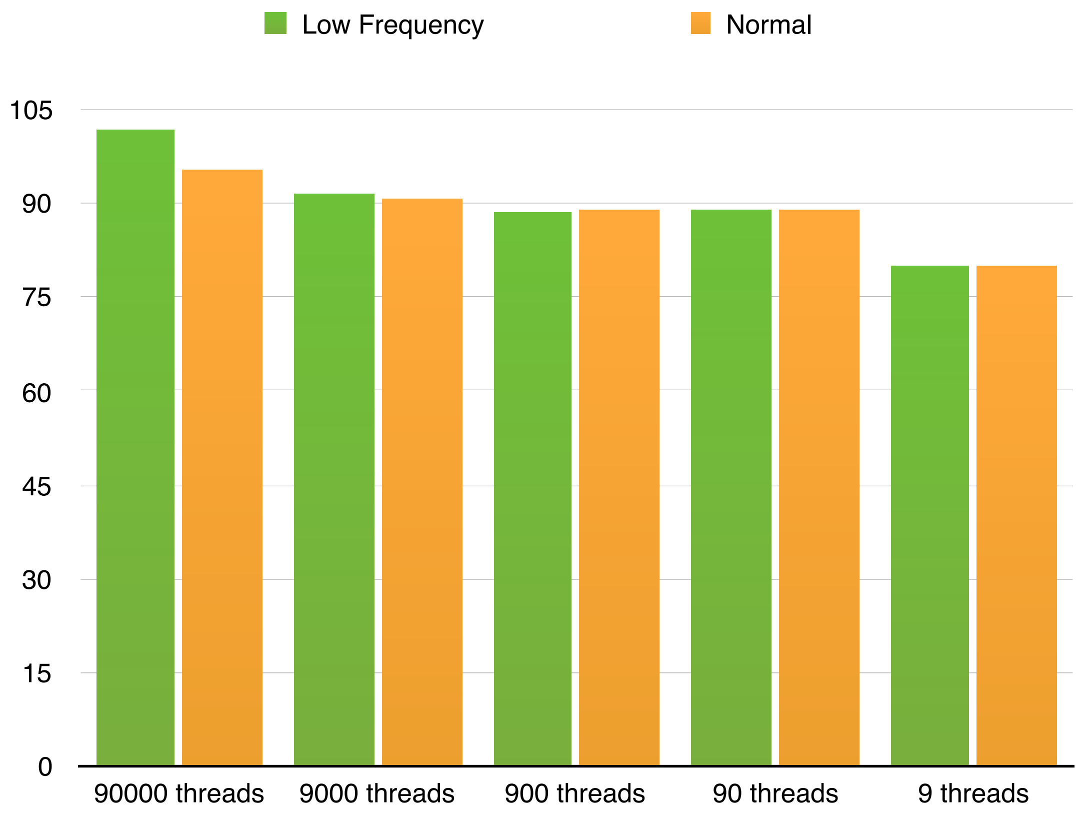


Table 2: (Intel) The execution time of the two implementations

This graph shows the execution time for the low frequency test and the normal one. With 90000 threads and 1 ms of sleep we have a performance degradation of 5 sec where all other measures are almost identical. Under one millisecond, there is high performance degradation (even 3 times slower for low frequency spinlock with 0.5 ms of sleep) due to the overhead that my implementation has. Another thing that impacts the performance is the fact that my system has a maximum transition latency of the CPU frequencies of 0,97 ms, under this threshold we may even experience system to freeze.

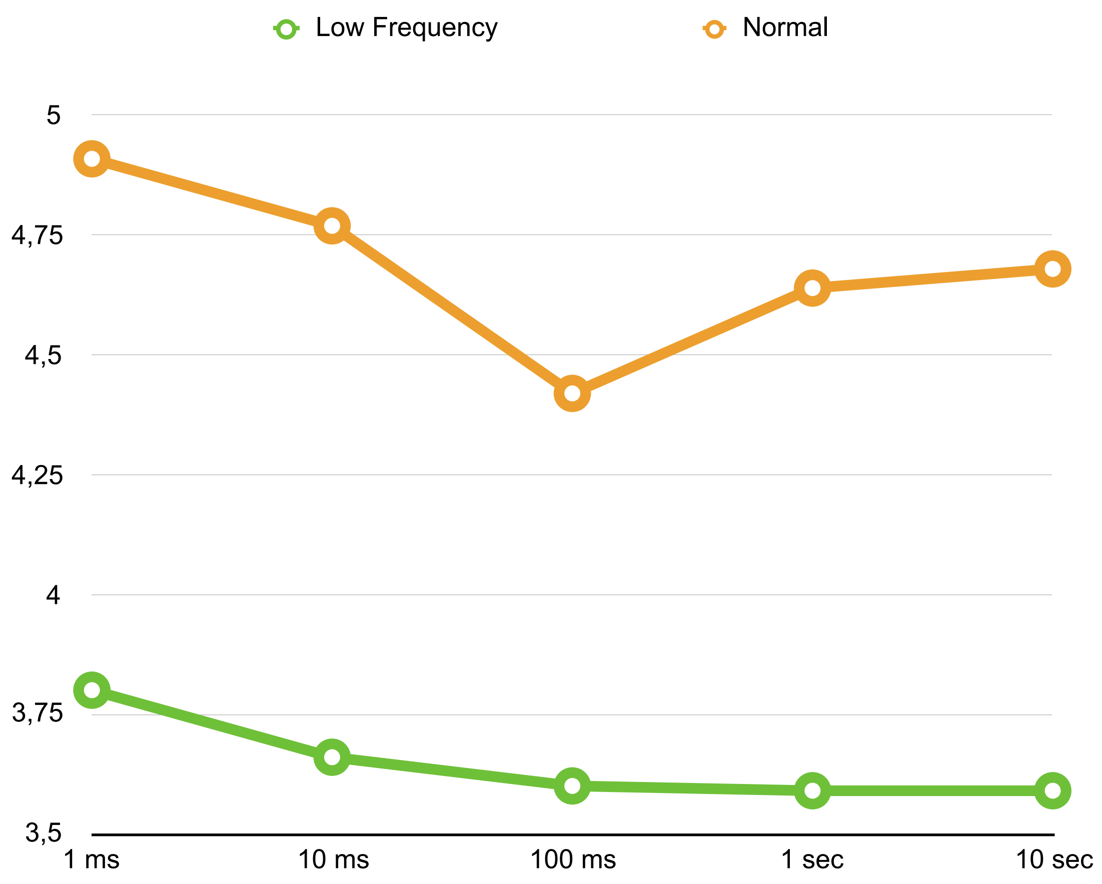


Table 3: (Intel) Energy consumption with 2 threads at a time

In the table 3 we can see the gain on energy consumption (in Watts). Between a millisecond and 100 milliseconds there is a large difference in energy consumed in the green line due to the overhead of `set_low_freq` in threads.

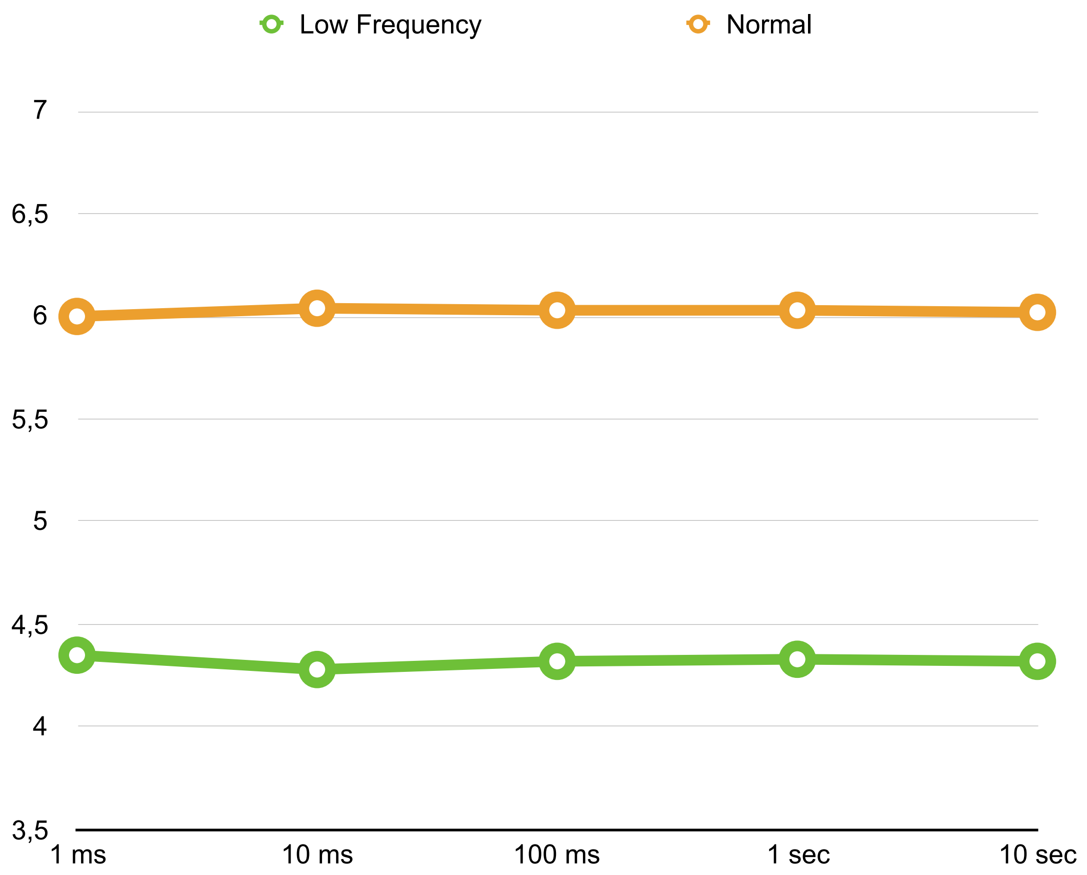


Table 4: (Intel) Energy consumption with 3 threads at a time

In table 4 we can see that there is not much difference between one millisecond and 10 seconds due to the fact that there is always 100% of workload per core. There is a small difference at one millisecond because there are more phases where the CPU rises to frequency, this is because there are more threads and they acquire the lock faster.

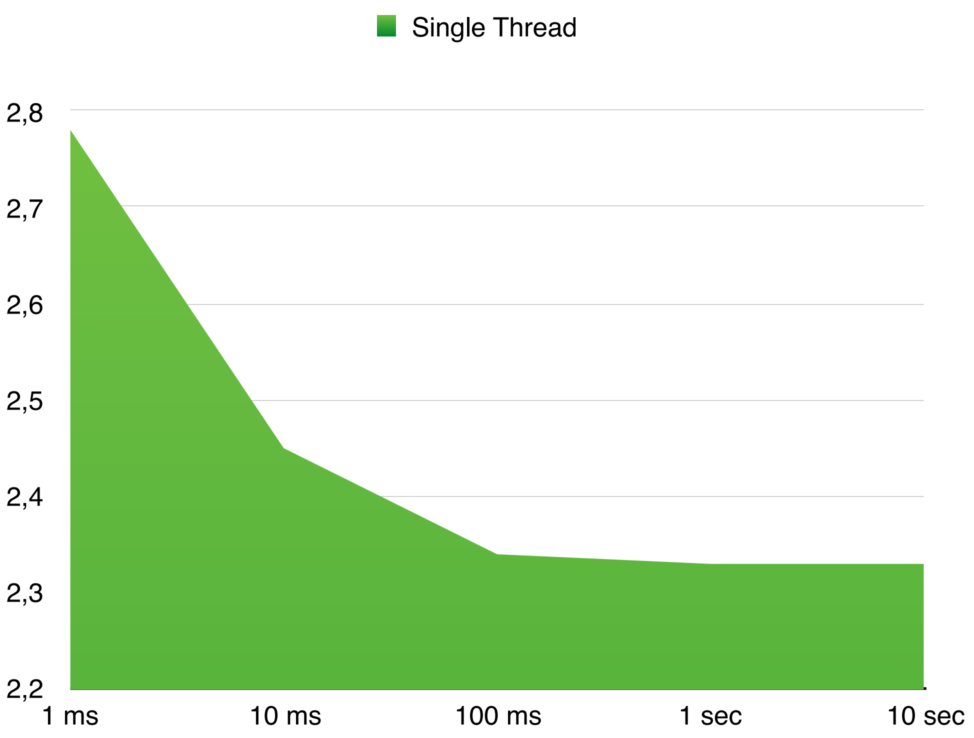


Table 5: (Intel) Energy consumption single thread (1)

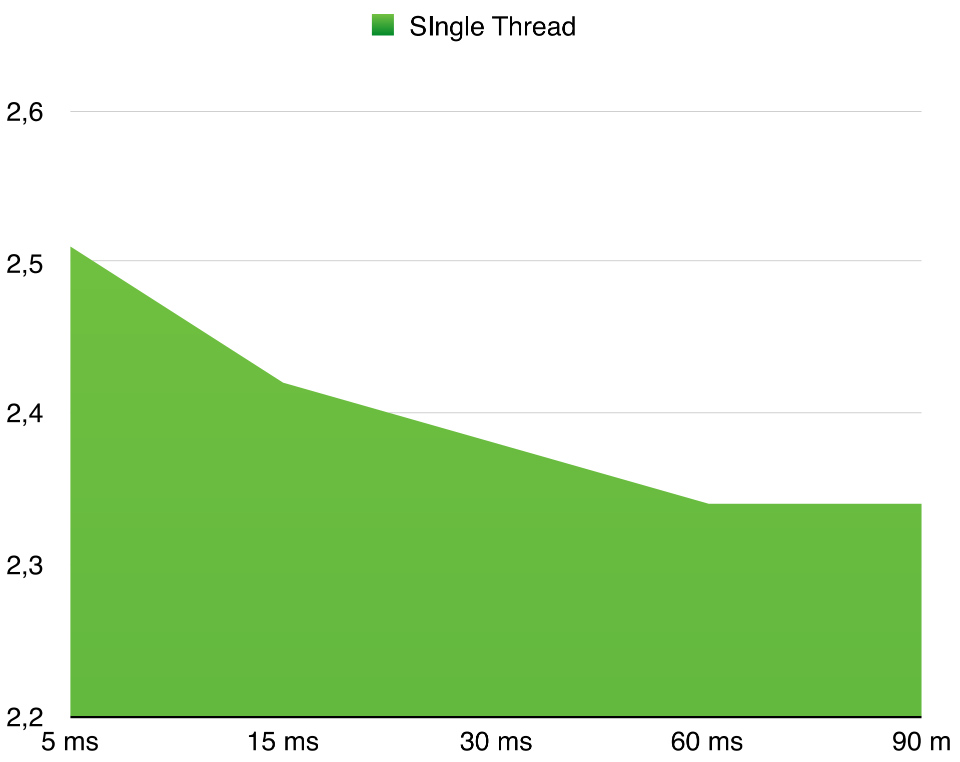


Table 6: (Intel) Energy consumption single thread (2)

Table 5 and 6 instead show a single thread that continuously invokes `set_low_freq` and `reset_low_freq`. In the midst there is a sleep phase that varies in length (1 ms to 10 sec). These tests were made to measure only the overhead of the two calls. The cost of calls is mitigated if there is a critical section of at least 100 milliseconds.

I would like to clarify that the values are valid on this system, and it is possible that the point where the measurement becomes constant can be before or after the 100 milliseconds on other systems. For the same reason the transition time to set the frequency may changes from system to system.

5.2 - AMD measures

The following measurements were made on the server of the “Dipartimento di Ingegneria Informatica, Automatica e Gestionale - DIAG”:

```
CPU:
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                32
On-line CPU(s) list:   0-31
Thread(s) per core:    1
Core(s) per socket:    8
Socket(s):             4
NUMA node(s):          8
Vendor ID:             AuthenticAMD
CPU family:            16
Model:                 9
Model name:            AMD Opteron(tm) Processor 6128
Stepping:              1
CPU max MHz:           2000.0000
CPU min MHz:           800.0000
BogoMIPS:              3990.16
Virtualization:        AMD-V
L1d cache:             64K
L1i cache:             64K
L2 cache:              512K
L3 cache:              5118K
RAM:
MemTotal:       66107700 kB
MemFree:        65153708 kB
MemAvailable:   65138320 kB
KERNEL:
Linux 3.16.0-4-amd64 x86_64
```

The version of `powerstat` is not entirely compatible with RAPL on the system under consideration, so I have implemented measurements via RAPL interface myself.

With the increased number of cores of this machine I made tests where increasing the size of the critical section and the number of threads independently. In this case the thread that holds the lock increments actively an integer (no sleep) for a certain amount of cycles in order to simulate the critical section, while other threads spinlocking. As usual I run normal and lowered frequency implementation of spinlocks.

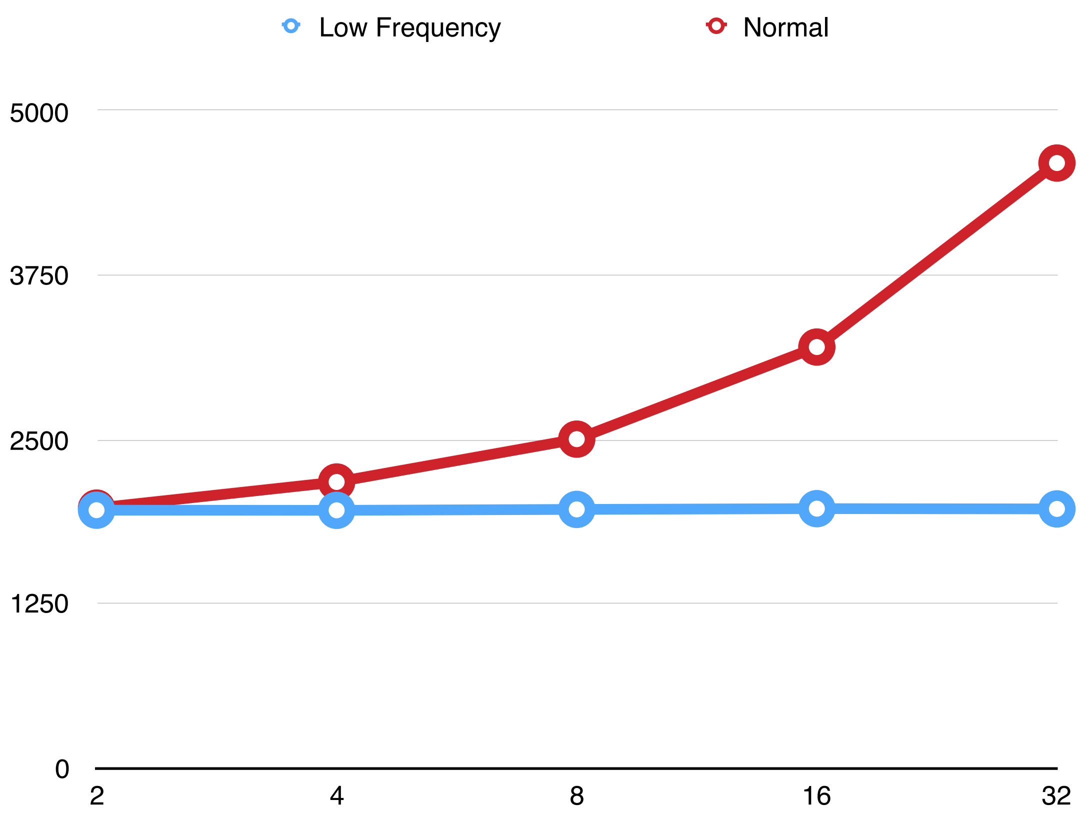


Table 7: (AMD) Power consumption of the two implementations

This test shows the energy consumption (in Joule) while increasing the number of threads that spinlock, the critical section remains unchanged at 1000000 cycles. As we expected normal implementation increases the energy consumption with the increasing of threads, instead my implementations shows no change in energy consumption due to the fact that all core (except one that is in c.s.) remain with the minimum frequency.

in the next graph I show the throughput, i.e. the number of critical section executed per second, of the same test as before.

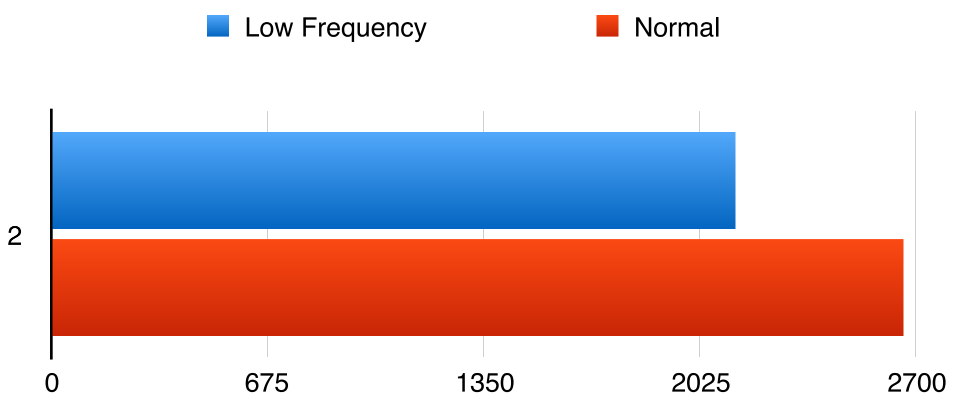


Table 8: (AMD) Performances of the two implementation

There is a performance degradation of about 20% in my implementation due to the fact that the critical section is too short. In the case that all 32 cores are running we have a 20% performance degradation but a reduction in energy consumption of 58%.

I run an additional test in the same manner as the previous one but with a critical section of one million cycles.

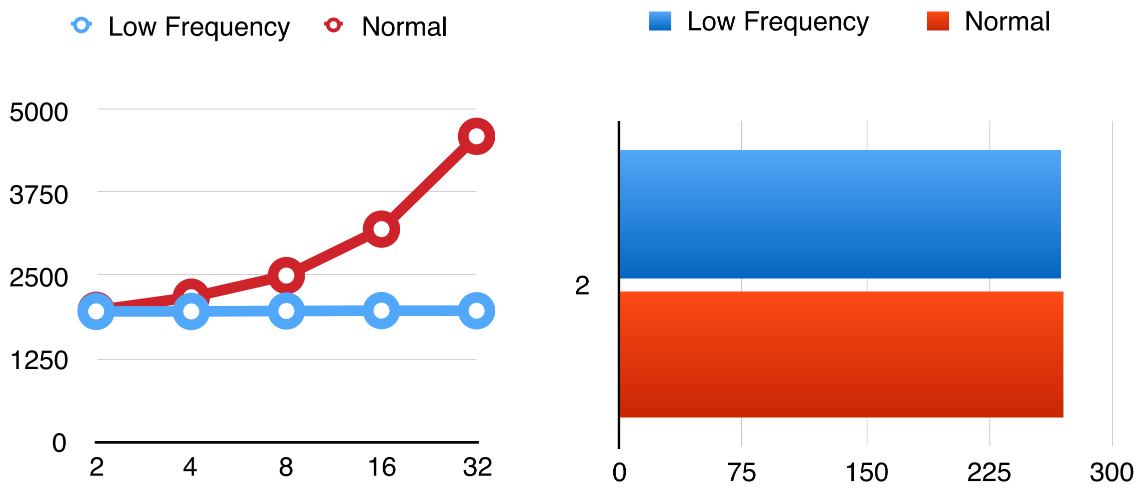
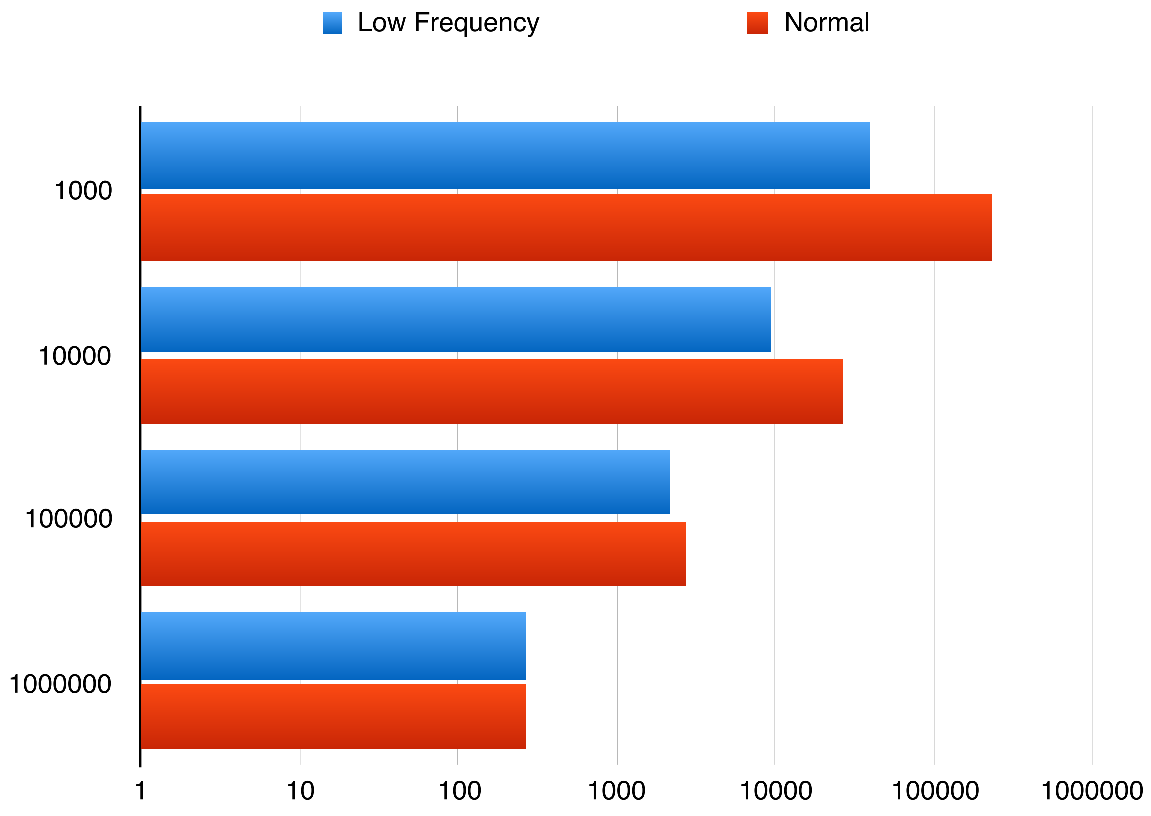


Table 9: (AMD) performances with one million cycles

Obviously the consumption follows the first test while this time we have no performance degradation; in fact we have the same values of critical section per second (which is less then before because c.s. is longer).

In the last graph I show the throughout (in a logarithmic scale) by varying the length of the critical section from 1000 cycles up to one million.



As expected the degradation of performance for critical sections that are too short is very high. It attenuates with the increasing of the critical section and eventually drops to zero.

6 - Considerations on improvements

In this section I would like to discuss possible improvements that could be made to this implementation in view of the work of other colleagues I'm interacting with.

One of the biggest benefits would be to reduce the overhead associated with lowering and raising frequencies in the kernel module:

In my current implementation, these operations are performed by passing through pseudo files, which has its own overhead. Actually I have made pointers to files global so that we do not have to execute the open function every time, but it remains the fact that there is the overhead due to the `vfs_write`.

Still on performance effects, writing on these pseudo files does not make the changes instantly but this changes have a transition time (which is 0.97 ms on my system). It would be very interesting to implement the frequency changes through the use of specific registers of the cores; These registers are also used by low-level utilities and are machine dependent. My colleague Daniele Zuco is currently working on this subject on his thesis.

We could do a more thorough study on AMD machines that have the advantage of enabling the frequency change with the granularity of the single core and, in this case, acquire statistical data that thanks to machine learning we could use to elaborate different approaches based on the use of spinlock and the length of the critical section. In fact, we can think of a system that is aware of the fact that a critical section is too short and decides not to pay the overhead of the `low frequency spinlock` autonomously. Or to a system that manages cases where minimizing the frequency may not be the best solution in terms of performance and energy consumption and then, thanks to the data acquired, to sets the most appropriate frequency.

Another great improvement would be to have the scheduler patch with the `on_schedule()` function already present in the kernel without the possibility of jumping to distant memory zones every time. In fact, since the patch is applied at runtime, there is no possibility to have compile-level optimizations.

7 - Conclusions

With my implementation of the spinlock there is an energy savings of **19**% up to **31**% on my the Intel machine and of about **58**% (in the best case) on the server of DIAG (AMD). As we have seen the added overhead, and the associated performance degradation, is soon tempered as the critical section becomes longer. This is especially because we need that the critical section must be at least as long as the transition time needed to the system to change the frequencies and that is bound to the use of pseudo-files.

The lower limit of the critical section can be made even smaller as soon as he system under consideration becomes more powerful, i.e. needs less time to transition; so that we can also have critical sections very brief where it is still convenient to use the low frequency spinlock. The expectation on servers with hundreds of cores is that energy saving can be even greater in terms of percentage.

The most important feature is probably the fact that the whole architecture is configurable at runtime without the need to restart or recompile the kernel. The introduction of my library is, in this way, transparent to the system end also to end-users because the APIs are practically equal to those used by the Linux spinlock library, and it is therefore possible to automate through a simple script the passage from the old spinlock present in the various programs of the system to the low-frequency ones.

List of Figures

List of Tables

Acknowledgments

I would like to thank my rapporteur Francesco Quaglia, Pierangelo Di Sanzo, Alessandro Pellegrini, and all the PhD students of high performance computing who have supported me and helped in the development of the code, to compare the tests and to make them trustworthy. I also thank the “Dipartimento di Ingegneria Informatica, Automatica e Gestionale” for making me available the instrumentation useful to the development of my thesis.

References

- [1] Intel Turbo Boost: http://files.shareholder.com/downloads/INTC/0x0x348508/C9259E98-BE06-42C8-A433-E28F64CB8EF2/TurboBoostWhitePaper.pdf
- [2] RAPL: https://01.org/blogs/2014/running-average-power-limit-—-rapl
- [3] APU: http://www.amd.com/en-us/press-releases/Pages/amd-fusion-apu-era-2011jan04.aspx
- [4] https://www.greentechmedia.com/articles/read/improving-the-energy-efficiency-of-computing-while-moores-law-slows
- [5] Linux: http://swift.siphos.be/linux_sea/whatislinux.html
- [6] https://kernelnewbies.org/FAQ/LibraryFunctionsInKernel
- [7] http://www.di.ens.fr/~pouzet/cours/systeme/bib/publ_1995_liedtke_ukernel-construction.pdf
- [8] https://web.stanford.edu/~ouster/cgi-bin/papers/osfaster.pdf
- [9] http://www.tldp.org/LDP/lkmpg/2.6/html/lkmpg.html#AEN567
- [10] https://github.com/HPDCS/schedule-hook