Cryogenic:
Enabling Power-Aware Applications on Linux

Alejandra Morales*, Wilfried Daniels†, Danny Hughes†, Christian Grothoff*

*Free Secure Network Systems Group, Technische Universität München, München, Germany
Email: {alejandra.morales,grothoff}@tum.de

†iMinds-DistriNet, KU Leuven, Heverlee, Belgium
Email: {firstname.lastname}@cs.kuleuven.be

***Abstract*—Modern hardware devices that are idle for a certain period of time enter into sleep mode as a means of reducing power consumption. Naturally, devices should remain in sleep mode for as long as possible to maximize energy savings. However, a growing number of applications perform non-urgent background tasks, which may force hardware to wake up. It would be better if such non-urgent activities could be scheduled to execute only when the respective devices are already in use, as this would maximize the duration of sleep-states. This requires cooperation between applications and the kernel, as only the kernel can coordinate between applications that access the same hardware, and only applications know which tasks can be deferred and for how long.**

This paper presents the design and implementation of Cryogenic, a POSIX-compatible API that enables the clustering of tasks that access the same hardware. Specifically, the Cryogenic API allows applications to defer tasks until devices are used by other processes. This way, non-urgent tasks can choose not to wake up the device they require and instead defer their execution until either other tasks force the device to be powered on, or until the task becomes urgent.

Cryogenic has been realized as a new Linux module, which integrates with the existing POSIX event loop system calls. This allows the use of Cryogenic on many different platforms, as long as the platform uses a Linux-kernel at the heart of the operating system.

In addition to describing the design and implementation of Cryogenic, this paper contains experimental results that demonstrate Cryogenic's ability to reduce power consumption using physical measurements on a Raspberry Pi.

***Index Terms*—Task scheduling, energy efficiency, POSIX.**

I. Introduction

Energy is a critical resource for mobile devices as it determines their lifetime. Modern smartphones and tablets allow users to install a large number of applications on their devices and some of these applications consume energy even when the user is not interacting with them, due to background tasks. Two examples of this behaviour are Google Latitude [1] and Google Calendar [2]. The former was a feature of Google Maps that allowed users to update their current location and share it with their contacts. The task ran in the background in order to periodically update the location of the device, so that users were able to see their own location and the location of their friends on a map. Google Calender allows users to share calendars or single events with contacts. If users enable background data on their devices, new events are automatically updated and the user does not need to specifically request synchronization.

Modern devices often have the capability to detect that they have been idle for a certain period of time and then enter into sleep mode as a means of reducing power consumption [3]. Naturally, waking up (i.e. resuming the activity of) suspended devices takes both time and energy [4], [5], [6] and it is therefore desirable to avoid unnecessary wake-ups. In this paper, we present an approach to reducing the energy consumption of background tasks that typically prevent hardware from entering sleep mode or even force hardware to wake up.

The key contribution of this paper is *Cryogenic*, a mechanism that permits developers to implement power-aware applications. Cryogenic achieves this by enabling the deferment of non-urgent tasks that require use of a device that is currently idle. As a result, background tasks wait until the corresponding device is used by some other process, and are then executed in a clustered fashion. This behaviour *maximizes sleep periods* for every device and *eliminates expensive wake-up operations*. Application developers may also specify a time-limit on task deferral. Thus, Cryogenic allows programmers to trade-off between the responsiveness of an application and the amount of energy it consumes.

In contrast to prior energy saving approaches, Cryogenic is implemented as a POSIX-compliant Linux kernel module, making it easy to install using the normal Linux module installation procedure. The Cryogenic API is simple and elegant, allowing developers to schedule deferred background tasks with just a few lines of code. Furthermore, installation of the kernel module has no effect upon legacy software.

II. Related work

Prior work on reducing the power consumption of energy-constrained devices has produced operating systems, kernel extensions and specialized hardware.

A. Cinder

Cinder [7], [8] provides three basic mechanisms for energy management: *isolation*, *delegation* and *subdivision*. *Isolation* prevents applications from consuming too much energy or

starving other applications, *delegation* allows an application to lend its energy to other applications and *subdivision* allows an application to partition its available energy. By combining the three mechanisms, Cinder provides fine-grained monitoring and control of application energy consumption. Cinder achieves this by extending the HiStar [9] operating system with two new kernel objects: *reserves* and *taps*.

A *reserve* represents a right to use a given quantity of energy. All threads are associated with a reserve from which they draw energy, and every reserve has a label that controls which threads can manipulate it. As an application consumes energy, the relevant quantity is subtracted from the corresponding reserve, and the kernel ensures that no application performs actions for which its reserve does not have enough energy. Threads may subdivide their available energy in order to delegate some of their reserve to other threads and may then track the energy that these threads consume. This provides accounting information and permits applications to be made power-aware.

A *tap* is an special-purpose thread whose job is to transfer energy between reserves. While reserves provide quantities of energy that may be consumed by threads, taps control the rate at which these quantities may be consumed. A tap is composed of a rate, a source reserve, a sink reserve, and a label that determines the privileges needed to transfer energy from the source to the sink.

Reserves and taps form an abstract *graph of energy consumption rights*, wherein the system battery is represented as the root reserve, whose energy is subdivided and thus allocated to reserves for each thread.

Discussion: The design philosophy of Cinder is quite different to that of Cryogenic. Cinder provides advanced energy management features, through a new operating system; however, it provides no support for the execution of legacy applications. In contrast, Cryogenic is realized as a standard Linux kernel module that can be easily installed on existing Linux systems and, critically, it does not interfere with the execution of legacy software. Considering overhead, migrating an application to the Cinder operating system is likely to require a complete redesign. In contrast, the modifications necessary to use Cryogenic are minimal: typically just a few lines of code are needed to defer non-urgent tasks.

B. Energy-aware processing in the Windows 7 kernel

Processor activity has a significant influence on the power consumption of a system. Modern processors enter into a low-power state during periods of idleness between the execution of instructions. However, if the idle period is too short, the power required to enter and exit the low-power state could be greater than the power saved. The Windows 7 kernel provides two mechanisms that aim to reduce the power consumption of periodic software activity: Timer coalescing [10] and intelligent timer tick distribution [11].

Timer coalescing: The Windows kernel scheduler is driven by a timer interrupt that has a default period. On every timer interrupt the kernel checks whether scheduled timers have expired and, if so, it performs a callback to the function associated with the timer. Timer coalescing allows applications and drivers to set a tolerable delay for the expiration of their scheduled timers. The kernel then uses this delay to adjust when the timer expires in order to maximize the coincidence of timer expiration. Two approaches are provided through which developers can take advantage of timer coalescing. In the case of applications, a new user-mode function is provided that allows developers to set the period after which a timer should expire and its tolerable delay[1]. In the case of drivers, this information may be specified through an equivalent kernel-mode function[2].

Intelligent timer tick distribution: In systems with multiple logical processors, timer interrupts are mirrored on every processor and callbacks for the corresponding expired timers are performed. The intelligent timer tick distribution (ITTD) [11] reduces the amount of timer interrupts by only waking application processors from low-power states when software timers expire or non-timer hardware interrupts occur. Application processors (AP) are any processors in the system other than the base service processor (BSP). ITTD provides maximum benefits when combined with timer coalescing. After timer coalescing has grouped callbacks in time, it is more likely to find timer interrupts that have no work to do and thus ITTD can remove them. While ITTD does not affect the BSP, the extension of idle periods and thus energy savings are significant for APs.

Discussion: The energy-aware processing approach of the Windows 7 kernel is similar to that of Cryogenic. Both approaches aim to conserve energy by maximizing idle periods and reducing hardware wake-ups. However, the scope of Cryogenic is different. While the Windows 7 kernel extensions focus on minimizing processor energy consumption, Cryogenic focuses on reducing the energy consumption of devices such as hard disks and network cards. In terms of deployment effort, both systems require the installation of a new kernel module. Considering developer overhead, both Cryogenic and timer coalescing require similar effort to adapt applications through use of the new APIs, while ITTD is completely transparent to the developer and requires no modifications to applications.

Timer coalescence has subsequently been adopted by Apple in OS X Mavericks to reduce the amount of background work that is performed while the machine is running on battery power [12]. The Linux kernel also introduced a new configuration parameter[3] as of version 2.6.21 that allowed CPUs in lower-power states to remain in this state longer. This method was named *tickless kernel* [13] and has effects similar to ITTD.

[1] http://msdn.microsoft.com/en-us/library/windows/desktop/dd405521(v=vs.85).aspx

[2] http://msdn.microsoft.com/en-us/library/windows/hardware/ff553249(v=vs.85).aspx

[3] NO_HZ: http://lxr.hpcs.cs.tsukuba.ac.jp/linux/kernel/time/Kconfig

C. *big.LITTLE Processing*

big.LITTLE Processing [14], [15], [16] is an energy saving mechanism for mobile platforms developed by ARM. big.LITTLE takes advantage of the different usage patterns observed on smartphones and tablets, wherein periods of high processing activity, such as gaming or web browsing, alternate with longer periods of low intensity tasks like e-mail or audio. Using this mechanism, software execution is dynamically transitioned to the appropriate CPU depending on performance needs. big.LITTLE achieves this by combining two processors that implement the same instruction set, but with different performance and energy profiles. Therefore, all software instructions execute in an architecturally consistent way on both the big and the LITTLE processor. The energy consumption and performance of each processor differs due to their micro-architectures. The main difference being that the LITTLE processor has a pipeline length between 8 and 10 stages, whereas the pipeline of the big processor has a length between 15 and 24 stages; this is critical as the energy consumed by the execution of an instruction is related to the number of pipeline stages it must traverse [17]. big.LITTLE supports two approaches: *Migration* and *Multi-Processor (MP)*.

big.LITTLE Migration: In this model, the kernel scheduler is unaware of the big and LITTLE cores and migration of software is managed by a power management subsystem in kernel space. When the LITTLE processor is executing at capacity and more perfomance is demanded, a migration is executed and both the operating system and the applications, move to the big processor. big.LITTLE supports two classes of migration: *CPU migration* and *cluster migration*. Cluster migration transfers the context between all LITTLE CPUs and the same number of big CPUs. Therefore, only one cluster can be used at the same time. This is inefficient when the computational load is asymmetrically distributed accross cores. CPU migration addresses this problem by pairing each LITTLE CPU with a big CPU. In this case, only one CPU per processor pair can be used at once. Unused processors are switched off in either case.

big.LITTLE MP: In this case, the performance requirements of running tasks determine whether a big processor must be powered on. If demanding tasks must be executed, a big processor is powered on to run them while non-demanding tasks can keep executing on a LITTLE processor. As in the previous model, any processors that are not being used can be powered off. MP thus permits the execution of applications on the processing resource that is most appropriate to their requirements.

Discussion: The big.LITTLE approach is quite different to Cryogenic as it requires a specific hardware architecture to reduce power consumption in addition to software support. big.LITTLE is therefore a platform-specific rather than a generic approach to energy conservation. Furthermore, big.LITTLE is only concerned with processor energy consumption, whereas Cryogenic considers all devices. In principle, systems with big.LITTLE processing could also use Cryogenic as software migration would not interfere with its operation. However, further study is required to identify and eliminate problematic interactions between the two systems. For example, if Cryogenic forces many tasks to defer their execution, processing will occur in a clustered fashion. This increased demand could force a processor migration and therefore reduce power savings.

III. Design and implementation of Cryogenic

The main goal of Cryogenic is to reduce the number of hardware wake-ups such that the duration of idle periods is lengthened and thus devices can go into sleep for longer, which is then expected to reduce overall power consumption.

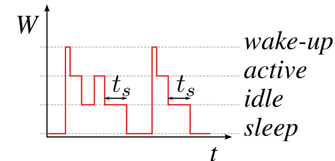


Fig. 1: Energy consumption model

Figure 1 provides a simplified model of device power consumption which we will use to illustrate our design. Devices can operate in four states that differ in their energy consumption rate. These states are: *active*, *idle*, *sleep* and *wake-up*. When a device is actually working, it remains in the active state until it has completed its tasks. When this happens, the device enters the idle state, that has a lower consumption. At this point, two state transitions are possible: it can become active again if it is requested to perform more tasks or otherwise it enters the sleep state, which has the lowest power consumption. This can only happen if the device is idle enough time to reach the sleep timeout t_s. Once the device is sleeping it can be requested to work again and thus it must change from the sleep to the active state. This transition requires the device to cycle through the *wake-up* state, that has the highest power consumption.

Although in the figure the increase of consumption from one state to the next one with higher consumption is proportional for all transitions, this is only a representation and the real increases will heavily depend on the specific device. Similarly, durations of the periods and especially the sleep timeout will likely be different from one device to another.

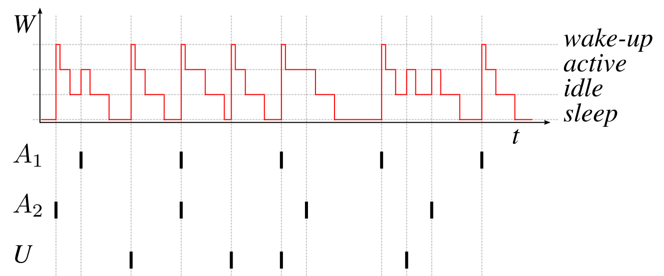


Fig. 2: Illustration of power consumption with uncoordinated tasks.

Figure 2 shows a hypothetical scenario where Cryogenic would be beneficial. There are two applications, A_1 and A_2, that execute tasks periodically, each one with a different fixed period. Furthermore, there are jobs executed due to user interaction, U; these have no predictable pattern. Assuming that all tasks want to make use of the same device, e.g. the network card, the figure illustrates the resulting power consumption for this device. As we can see, every time tasks need to use the device when it is currently sleeping, there is an overhead power consumption caused by the transition from sleeping to active. Then comes the active period and once tasks are finished, the device becomes idle. In this example, the frequency of execution forces the device to switch from idle to active many times, preventing it from going to sleep.

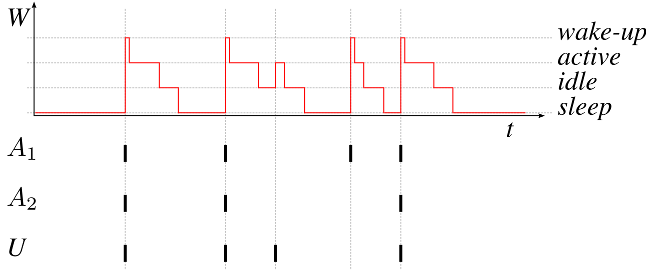


Fig. 3: Power consumption with coordinated tasks A_1 and A_2.

Assuming that the tasks A_1 and A_2 are not urgent, they could be deferred until other tasks interact with the device. This way, the network card is already active when A_1 or A_2 need to use it, and the peak consumption caused by the state transition is eliminated. Naturally, a task not being urgent does not imply that it should not be performed at all. For Cryogenic, we assume that — instead of a simple wake-up time — tasks specify a *time interval* during which they should be run. Figure 3 illustrates the resulting behaviour. Here, the number of wake-ups has been reduced from 7 to 4. Note that the user interaction, which is considered urgent and is thus not coordinated by Cryogenic, acts as the main trigger for waiting tasks. As a result, most tasks are clustered to be executed right after the user's tasks, which allows sleep periods to become longer.

Cryogenic is implemented as a Linux kernel module that enables the notification of processes that want to coordinate their activities to reduce power consumption. Using the standard POSIX APIs, processes can specify a time interval and a device which they want to use. They are then woken up during that time interval, either because some other process is using the device or because they have reached the end of the time interval. We will now describe how the Cryogenic kernel module operates, and then give a real-world example for how application developers can use this new functionality.

A. Overview of the Cryogenic kernel module

The whole implementation of Cryogenic is embedded in a kernel module that works as a character device driver [18, Ch. 1]. Character device drivers are the most common class of Linux drivers and offer direct, unbuffered access from user processes to devices.

When Cryogenic is loaded, a set of character devices is created and a subset of system calls is defined. The system calls handle the character devices through the device nodes created under `/dev/cryogenic/`. These device files are used by developers to manage the interaction between applications and hardware devices.

Nodes corresponding to SCSI devices are named after the device serial number. For network devices, the default interface name is used. Figure 4 shows the list of files in a system that has attached three hard drives, two ethernet cards and a wireless LAN card.

```
1  # ls -l /dev/cryogenic/
2  total 0
3  crw------- 1 root root 247, 0 Dec  9 16:03 9VP26KSV
4  crw------- 1 root root 247, 2 Dec  9 16:03 eth0
5  crw------- 1 root root 247, 3 Dec  9 16:03 eth1
6  crw------- 1 root root 247, 5 Dec  9 16:04 WD-WCAU46069319
7  crw------- 1 root root 247, 1 Dec  9 16:03 WD-WCAV90469334
8  crw------- 1 root root 247, 4 Dec  9 16:05 wlan0
```

Fig. 4: Sample list of device files in `/dev/cryogenic/`

B. Cryogenic system calls

As a device driver, Cryogenic must provide the necessary system calls to manage the character devices it creates. In addition to pretty canonical implementations of `open` and `release` (which corresponds to `close`), Cryogenic provides implementations for `poll` (which corresponds to the `select`-family of system calls) and `ioctl`. We will now discuss these two operations in more detail.

1) ioctl: Cryogenic uses the `ioctl` [18, Ch. 6] system call to allow applications to specify the desired time interval during which an operation should be executed. Cryogenic defines the `ioctl` request `SET_DELAY_AND_TIMEOUT`, which takes an argument of type `struct pm_times`. This structure is provided by Cryogenic and its definition is given in Code 1.

Code 1: Definition of `pm_times`

```
struct pm_times {
    unsigned long delay_msecs;
    unsigned long timeout_msecs;
};
```

When an application calls `ioctl` on a Cryogenic handle, the respective delay and timeout values are simply associated in the kernel with the respective handle. From the application developers perspective, both values specified in `struct pm_times` represent relative time in milliseconds.

2) poll: The `poll` [18, Ch. 6] system call determines whether a task is allowed to perform an I/O operation or if it must wait for some event to happen. Cryogenic's implementation of `poll` first checks if the device was removed. If it was, it returns a value indicating that the device is "ready". The application will then attempt to use the removed device, resulting in an error that is handled in the standard manner. Otherwise, `poll_wait` [18, Ch. 6] is called to queue the task on an event queue for the corresponding device, and a timer is added to wake up when the respective timeout is reached.

C. Hotplugging

Cryogenic detects when a hardware device is plugged or unplugged and dynamically add or remove the corresponding devices and structures. The handling for SCSI devices is special in that Cryogenic can detect if a SCSI device is unplugged and then read. In this case, applications do not have to reopen the respective `/dev/cryogenic/` file handle, as Cryogenic will reuse the same minor number and thereby preserve the association.

In contrast, the configuration of network devices is likely to change after a reconnection. Thus, if network devices are reconnected, the userspace application must open a fresh handle. It should be noted that this is a design decision we made, and it is possible to revise this decision in the future.

D. Waking up

Two events may resume the activity of waiting tasks: an I/O operation on a device performed by other tasks, or the expiration of their timer. Furthermore, when a hardware device is unplugged or its device driver is unloaded, some actions must be enacted in order to keep the system's consistency and avoid future errors. In particular, removal of a device will also cause Cryogenic to wake up applications waiting for I/O on that device.

In all of these cases, Cryogenic causes the system calls that are currently blocked on `poll` to be woken up. If the minimum delay has expired (or the hardware device was removed), `poll` will return a value indicating that the operation is ready to proceed. If the device was removed, the userspace application still most likely try to perform an I/O operation, and will then receive an error indicating that the device is no longer present in the system. This way, error handling is delegated to those calls that need to deal with such errors regardless of whether Cryogenic is running.

E. Application migration

To demonstrate how existing applications can easily be transformed to use the Cryogenic API, we added support for Cryogenic to GNUnet, a framework for secure, decentralized peer-to-peer networking. Specifically, we modified one particular aspect of the GNUnet system, namely the discovery of neighbours in the LAN. We will now briefly describe the existing functionality in the GNUnet system, and then show the key modifications that were made to introduce the Cryogenic API into the system.

Peers in the GNUnet overlay network obtain address information of the other peers through UDP neighbour discovery in LANs. Every five minutes, the broadcast addresses of each IPv4 interface of the local system are gathered and a so-called "HELLO" message is sent to each of these addresses. Similarly, a "HELLO" message is multicast on each IPv6-capable interface. As these periodic broadcasts are not urgent, they represent a perfect opportunity for the introduction of Cryogenic. As a preparatory step before introducing any Cryogenic-specific code, the existing logic was slightly modified to use a separate task for each transmission. This step was necessary, as we want to trigger transmissions for each network interface independently.

The first modification was to add two additional fields to the `struct BroadcastAddress`, which will be used to hold the file descriptor of the corresponding character device and the delay and timeout for the transmissions. The new fields are illustrated in Code 2.

Code 2: New fields in `struct BroadcastAddress`

```
#if LINUX
  /** Cryogenic handle. */
  struct GNUNET_DISK_FileHandle *cryogenic_fd;

  /** Timeout for Cryogenic. */
  struct pm_times cryogenic_times;
#endif
```

The `struct GNUNET_DISK_FileHandle` is provided by the GNUnet API as a generic container for open files on different operating systems. Under GNU/Linux, it is simply a wrapper around an `int`. The second new field is an instance of the structure supplied by Cryogenic that we presented in Section III-B1. We note that all of our modifications are written in a way that ensures that the original application logic functions correctly in the absence of Cryogenic.

Next, the device node corresponding to each interface is opened if possible. This is done by extending the `iface_proc` function, which contains the per-interface initialization logic (Code 3). Our modification uses the pre-existing `GNUNET_DISK_file_open` function, which is a GNUnet wrapper around the `open` system call.

Code 3: Opening the character device for an IPv4 interface

```
#if LINUX
  char filename[128];

  sprintf (filename,
           "/dev/cryogenic/%s",
           name);
  if (0 == access (name, R_OK)) {
     ba->cryogenic_fd
       = GNUNET_DISK_file_open (filename,
            GNUNET_DISK_OPEN_WRITE,
            GNUNET_DISK_PERM_NONE);
  }
#endif
```

The next modification is about calculating and setting the delay and the timeout for the transmission using `ioctl` and scheduling the transmission job using `select`. Code 4 shows the new logic that was added to the functions that schedule the transmission of a message (which happens in the `udp_ipv4_broadcast_send` function).

Code 4: Setting the delay and timeout and calling `select`

```
#if LINUX
  if (NULL != baddr->cryogenic_fd) {
    baddr->cryogenic_times.delay_msecs
      = (plugin->broadcast_interval.
          rel_value_us/1000.0)*0.5;
    baddr->cryogenic_times.timeout_msecs
      = (plugin->broadcast_interval.
          rel_value_us/1000.0)*1.5;
    ioctl(baddr->cryogenic_fd->fd,
         PM_SET_DELAY_AND_TIMEOUT,
         &baddr->cryogenic_times);
    GNUNET_SCHEDULER_add_write_file (
        GNUNET_TIME_UNIT_FOREVER_REL,
         baddr->cryogenic_fd,
         &udp_ipv4_broadcast_send, baddr);
  }
  else
#endif
  /* existing logic (without cryogenic) */
  baddr->broadcast_task =
    GNUNET_SCHEDULER_add_delayed (plugin->
        broadcast_interval,
        &udp_ipv4_broadcast_send, baddr);
```

In order to calculate the delay and the timeout we make use of the broadcast interval, which is already stored in the `plugin` structure that is passed to the functions as a parameter. As we want to place the new tolerance window in a symmetrical position with respect to the current transmission period, the delay is set to the broadcast interval minus 50%, and the timeout is set to the broadcast interval plus 50%.

GNUnet does not define a wrapper function to call `ioctl`, thus the call is performed directly on the file handle. Here, we omitted the error handling for the sake of brevity.

In contrast to `ioctl`, the call to `select` is performed indirectly using GNUnet's event loop API. The call adds the given file handle to the respective `select` set and calls the given function if `select` determines that the descriptor is ready.

Finally, all of the open file descriptors should be closed , which in this case was done by adding the code from Code 5 in the appropriate place.

Code 5: Closing the character device descriptor

```
#if LINUX
  GNUNET_DISK_file_close(p->cryogenic_fd);
#endif
```

All modifications we made were limited to the file `plugin_transport_udp_broadcasting.c`, which is part of the GNUnet transport subsystem. The code shown includes the modifications for IPv4; equivalent changes were made for IPv6.

The code samples shown above represent all of the significant changes. We thus conclude that modifying existing applications to support Cryogenic typically requires simple and localized changes. The main change relates to the scheduling logic of tasks, which are typically already scheduled with an artificial delay. The required changes are also small because the implementation of Cryogenic — by means of a module and its redefinition of the POSIX system calls — allows the developer to reuse existing APIs in most cases. As a result, the developer does not need to invest in building or supporting significant changes to existing abstractions or APIs in the application.

IV. Experimentation

We will now present experiments we performed to evaluate Cryogenic's operation on the Raspberry Pi. To ensure the reproducibility of our results, we have evaluated Cyrogenic at two independent sites: TU München (TUM) and KU Leuven (KUL). Each site evaluated Cryogenic using a different energy measurement methodology, a different networking device and different Linux distributions. We first present the experimental setup and then discuss the experimental results.

A. Test programs

We created three test programs to simulate applications causing network traffic. Each program sends fixed-size UDP packets at particular times, and outputs the time of transmission to `stdout`. The first test program **(F)** sends packets at a specified frequency. The second program **(R)** randomizes the delay between transmissions within a specified range, and the third program **(C)** uses Cryogenic to send packets within a specified time interval, trying to align its transmissions with those of the other processes. In our experiments, we first compare the energy consumption of running four **(F)** processes in parallel using fixed, prime-numbered frequencies against running two **(F)** processes and two **(C)** Cryogenic processes using equivalent frequencies. This corresponds to a system with four periodic background tasks, where two have been modified to use Cryogenic.

We also evaluated the effect of converting one **(F)** process into a randomized **(R)** process, in effect replacing one background process with unpredictable, high-priority user interactions. The exact test scripts and client applications are available from the project's website.[4]

B. Configuration of the Raspberry Pi

The *TUM site* used the latest version of the Debian-based distribution Raspbian, which at the time of writing is *wheezy* and can be downloaded from http://www.raspberrypi.org/Downloads. The *KUL site* used ArchLinux version 2014.01.08, which can be downloaded from http://archlinuxarm.org/platforms/armv6/raspberry-pi.

In the case of Raspbian, the Cryogenic kernel modules were cross compiled on a general purpose PC before being loaded on the Raspberry Pi, while in the case of ArchLinux the kernel modules were compiled locally on the Raspberry Pi. Detailed instructions for how to compile and load the Cryogenic module can be found in [19]. Furthermore, in order to manage the general purpose input/output (GPIO) pins, the

[4] https://gnunet.org/cryogenic

`bcm2835` library had to be installed on the Raspberry Pi. We used version 1.32 for our experiments.

C. Circuit assembly

The assembly of the circuits in Figure 6 and Figure 7 is achieved through the pin header embedded in the Raspberry Pi. Figure 5 shows the pin header layout.

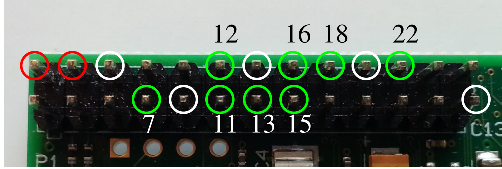


Fig. 5: Pin header layout

The pins surrounded by red circles are 5V power pins, the ones surrounded by white circles are ground pins, and the ones surrounded by green circles are GPIO pins. Thus, we connect the positive pole of the power supply to one of the 5V pins, and the negative pole to one of the ground pins.

The number next to each GPIO pin in Figure 5 is its identifier inside the pin header and is the number used to control the behaviour of the pin. The scripts that drive the experiments call C programs that set the corresponding GPIO pin to its high and low values when they start and finish their execution respectively. This allows measurement data to be precisely aligned with the actions performed by the applications.

Both sites used an oscilloscope to perform their measurements, wherein one voltage probe was connected to the GPIO pin used for signaling and the ground clamp of the probe was connected to one of the ground pins. Current draw was measured differently at each site.

At TUM, power draw was measured using a current probe placed around one of the cables that connect the Pi to the power supply, taking care to match the current direction indicator with the polarity of the circuit. If the probe were incorrectly connected, there would be no danger of damage, but the signal of the sample would be inverted. This is shown in Figure 6.

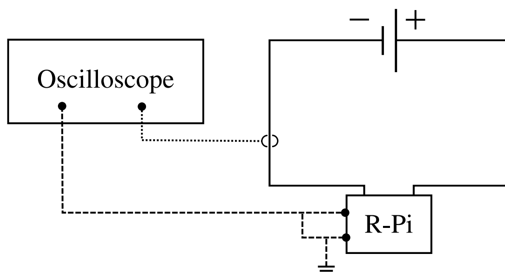


Fig. 6: Circuit diagram for measurements using a current clamp.

At KUL, current draw was measured by applying Ohm's law to the voltage drop accross a shunt resistor connected in series with the power supply of the Raspberry Pi, as shown in Figure 7. KUL used a high precision 1.5 Ohm resistor with a maximum relative error of 0.1%.

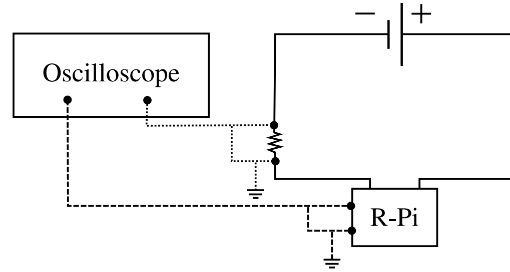


Fig. 7: Circuit diagram for measurements using a shunt resistor.

D. Experimental results

In this section we present the results of our experiments. Each experiment consists of a batch run of the scripts presented in Section IV-A. In order to obtain statistically significant results, each script was executed several times for every experiment and then the mean and the standard deviation of each set of results were calculated.

At both sites, experiments were run with a HDMI monitor and a wired USB keyboard connected to the Pi. At TUM, Cryogenic was tested with an *Edimax EW-7811Un* wireless WiFi USB adapter. At KUL, Cryogenic was tested with an *Option IconXY* 3G Modem.

1) Baseline consumption: In order to isolate the power savings arising from the use of Cryogenic, we measured the baseline power consumption of the Pi with the WiFi and 3G devices plugged in but idle. Table I illustrates the results of these measurements. For the following experiments, we subtracted the average baseline consumption from the total energy consumed to calculate the power consumption of the network devices (and the minimal additional CPU time required to prepare the transmissions); we added the standard deviation obtained to the standard deviation of every experiment.

As shown in Table I, the experiment run times differ between both experiments. The 3G experiments run longer because the embedded power saving strategy of the 3G modem spans longer time intervals than the WiFi modem. In order to properly capture this, all 3G experiments have a run time of five minutes, while for the WiFi dongle one minutes suffices. These experiment run times are propagated in all results discussed in this section. Table I also includes a time independent average power draw to facilitate direct comparisons between both baseline consumptions.

TABLE I: Baseline power consumptions.

	WiFi	3G
Total energy	113.50 J	686.34 J
Std. dev.	0.57 J	3.38 J
Experiment run time	60 s	300 s
Average power	1.89 W	2.29 W

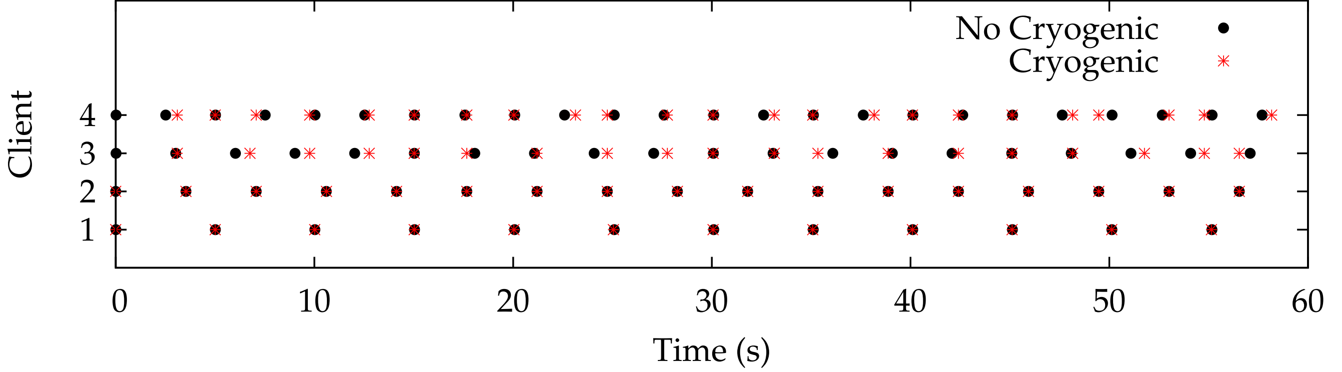


Fig. 8: Packet transmission times for non-randomized test programs. Clients 1 and 2 did not offer any scheduling flexibility to Cryogenic, while Clients 3 and 4 allowed Cryogenic limited scheduling flexibility in the pass where Cryogenic was enabled. As a result, execution times are then shifted slightly in an attempt to cluster network activities. The power-savings reported in the paper were observed on the basis of these limited shifts for Clients 3 and 4.

2) Packet transmission time: To illustrate how Cryogenic modifies the transmission time of packets, we plotted the times when packets are sent by the periodic test programs with and without Cryogenic (Figure 8). The figure shows that the transmissions of client 1 and 2 (type **(F)**) remain identical between both runs, while the transmissions of clients 3 and 4 shift when moving from type **(F)** (without Cryogenic) to type **(C)** (with Cryogenic). The figure also shows that the shifted transmissions of the **(C)** clients are now aligned with the remaining **(F)** clients.

3) Current draw due to packet transmission: Figure 9a and Figure 9b show the current draw when sending packets without Cryogenic over WiFi and 3G respectively. Figure 9c and Figure 9d show the current draw for the same time span with Cryogenic enabled. The timescale of each set of figures is quite different. In the case of WiFi, packet transmissions are separated by a few milliseconds, while in the case of 3G packet transmission are separated by a several seconds. It can be seen that the number of current peaks (caused by the wake-up of transmission hardware) decreases in both cases, reducing the overall amount of energy consumed.

When we compare the WiFi test runs on Figure 9a and Figure 9c, the packet sent by client 3 defers its transmission, while the transmission of the packet that belongs to client 4 is anticipated, so that both packets are sent almost simultaneously with the packet that belongs to client 2, which does not use Cryogenic. This results in one wake-up instead of three.

The 3G test runs shown in Figure 9b and Figure 9d exhibit a more complex behaviour. The packet transmissions in both figure do not always line up with peaks in current draw. This is because the 3G modem is cycling between a high power mode and a low power mode and is able to send and receive packets in both modes. The specific logic behind the power saving modes are proprietary and specific to each 3G modem, but from the figures we can observe that by rescheduling packets in a clustered fashion more time is spent in the low power mode. This validates that the power savings achieved by Cryogenic are truly generic.

4) WiFi Measurements at TUM: In this experiment, we connected the Pi to a protected WiFi network that was used only for the experiment. Nevertheless, we could not control interference from other WiFi networks and other devices in the vicinity operating in the ISM* frequency range.

Table II illustrates the period and tolerance values used for the experiment. Note that we use prime numbers to minimize the chance of tasks transmitting at the same time. The tolerance used for Cryogenic tasks is set to about 50% of the transmission period.

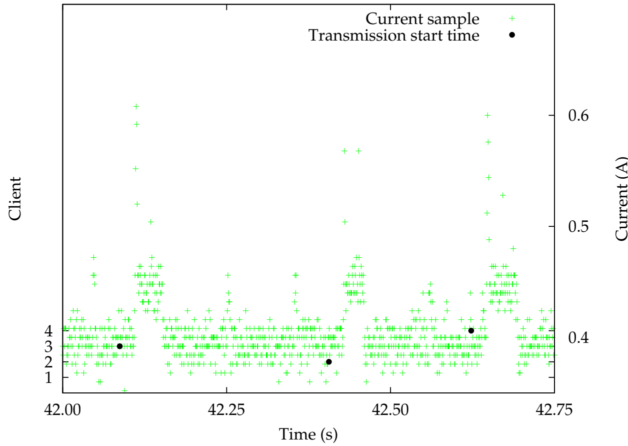
TABLE II: Programs, transmission periods and tolerances (in milliseconds) for the WiFi experiment.

No Randomization				Randomization			
Without Cryogenic **(P1)**		With Cryogenic **(P2)**		Without Cryogenic **(P3)**		With Cryogenic **(P4)**	
(F)	2503	**(F)**	2503	**(R)**	2503	**(R)**	2503
(F)	1747	**(F)**	1747	**(F)**	1747	**(F)**	1747
(F)	1499	**(C)**	1499±751	**(F)**	1499	**(C)**	1499±751
(F)	1249	**(C)**	1249±631	**(F)**	1249	**(C)**	1249±631

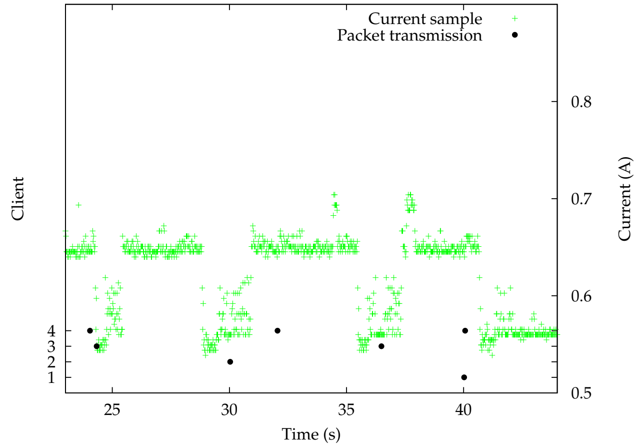
As can be seen from Table II, the use of Cryogenic introduces significant variability (of up to 0.75 seconds) into the transmission timings for the low-priority programs P2 and P4, this allows their transmissions to be synchronized with those of the high-priority programs P1 and P3.

Since Cryogenic may defer the transmission of some packets, it may slightly reduce the total number of packets sent during the course of an experiment (by 1–3% compared to non-Cryogenic runs). To ensure a fair comparison, we counted the number of packets sent during each experiment and report the energy consumption per packet transmitted using WiFi. The results are given in Table III.

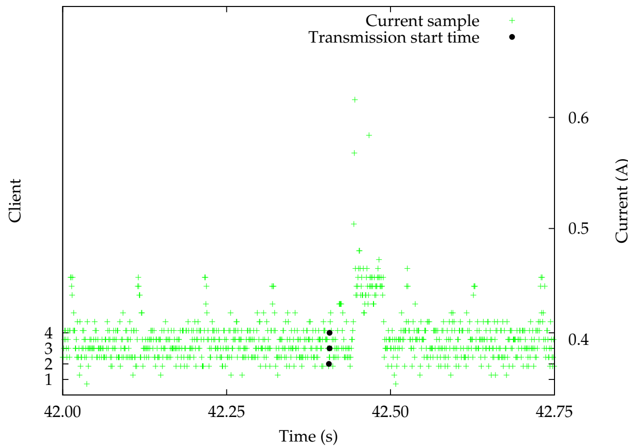
Table III shows that for WiFi transmissions, Cryogenic achieves power savings in all scenarios (from 2.3% to 6.8%). It should be noted that the results described above are a



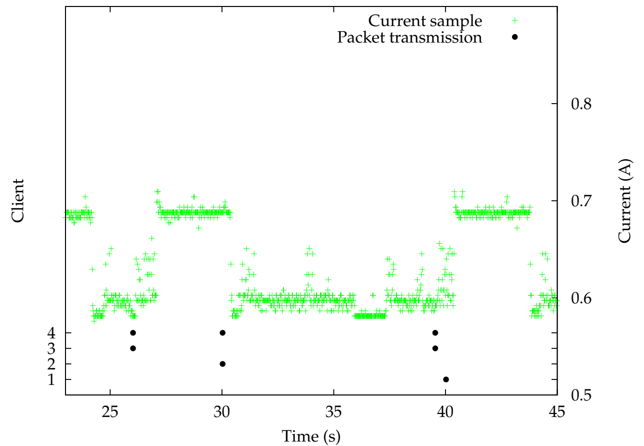
(a) Current draw of WiFi non-Cryogenic test run.



(b) Current draw of 3G non-Cryogenic test run.



(c) Current draw of WiFi Cryogenic test run.



(d) Current draw of 3G Cryogenic test run.

Fig. 9: Current draw of test runs.

TABLE III: Power consumption for the WiFi experiment.

	No Randomization		Randomization	
	Without Cryogenic	With Cryogenic	Without Cryogenic	With Cryogenic
Total	123.51 J	122.83 J	123.83 J	123.46 J
- Baseline	10.01 J	9.33 J	10.32 J	9.95 J
Std. dev.	0.92 J	1.02 J	1.05 J	0.99 J
Savings	**6.76%**		**3.58%**	
Per Packet	0.068 J	0.064 J	0.068 J	0.067 J
Std. dev.	0.006 J	0.007 J	0.007 J	0.007 J
Reduction	**5.29%**		**2.34%**	

TABLE IV: Programs, transmission periods and tolerances (in milliseconds) for the 3G experiment.

No Randomization				Randomization			
Without Cryogenic (**P1**)		With Cryogenic (**P2**)		Without Cryogenic (**P3**)		With Cryogenic (**P4**)	
(F)	20011	**(F)**	20011	**(R)**	20011	**(R)**	20011
(F)	15013	**(F)**	15013	**(F)**	15013	**(F)**	15013
(F)	12161	**(C)**	12161±6080	**(F)**	12161	**(C)**	12161±6080
(F)	8009	**(C)**	8009±4004	**(F)**	8009	**(C)**	8009±4004

representative subset of a larger series of experiments, which are described in [19].

5) 3G Measurements at KUL: In this experiment, we connected the Pi to the public BASE cellular network. As with the WiFi experiment, the tolerance used for Cryogenic tasks is set to about 50% of the transmission period and prime numbers are used to minimize the likelihood of simultaneous transmissions Table IV presents the period and tolerance values used for the experiment, while Table V reports the energy consumption per packet transmitted using 3G.

As can be seen from Table IV, Cryogenic introduces a variability of up to 6 seconds into the transmission timings for the low-priority programs P2 and P4, which allows their transmissions to be synchronized with those of the high-priority programs P1 and P3.

As with WiFi, Table V shows that for 3G transmissions, Cryogenic achieves power savings in all test scenarios.

The experiments reported in this paper should not be seen as the best-case energy savings that can be achieved with Cryogenic, but rather as a challenging and yet representative

TABLE V: Power consumption for the 3G experiment.

	No Randomization		Randomization	
	Without Cryogenic	With Cryogenic	Without Cryogenic	With Cryogenic
Total	861.56 J	848.95 J	858.41 J	852.40 J
- Baseline	175.22 J	162.61 J	172.07 J	166.06 J
Std. dev.	8.5013 J	5.2165 J	9.4671 J	8.6167 J
Savings	**7.20%**		**3.49%**	
Per Packet	1.788 J	1.694 J	1.654 J	1.628 J
Std. dev.	0.006 J	0.007 J	0.0910 J	0.0845 J
Reduction	**5.26%**		**1.57%**	

case-study. Reflecting upon the low-priority applications that we discussed in the introduction, such as Google Latitude [1] and Google Calendar [2], the delays introduced by Cryogenic (under 6.1 seconds in the worst case) are inconsequential. We therefore believe that Cryogenic offers generic energy savings for the large class of low-priority applications that can tolerate the necessary transmission delays.

V. Conclusion and Future Work

For the experiments in this paper two independent research groups have validated the performance of Cryogenic. Each site used a different energy monitoring methodology, network device and version of the operating system. This approach suggests that our results are repeatable and that the Cryogenic approach to power conservation will generalise.

The main limitation of the Cryogenic approach is that it can only provide benefits for applications that can defer operations and thus can leave some scheduling flexibility to the operating system. However, this class of applications is large and growing, as the intermittent connectivity afforded by mobile devices already requires that developers build applications in a way that can tolerate transmission delays.

While the specific hardware and software stack of each platform has a significant impact on power consumption, Cryogenic reduces the energy consumed of networking in all cases, achieving power savings of between 2.3% and 6.8% for WiFi and between 1.6% and 7.2% for 3G. The reductions achieved are moderate, but the modifications required to the original application are simple and localized, as discussed in Section III-E. It is important to note that: (i.) these savings are achieved while performing the same amount of work and (ii.) the Cryogenic test programs were not completely flexible to obtain savings, since Cryogenic was only allowed to defer about half of the transmissions. This evaluation is therefore far from the best case for Cryogenic.

Our future work will proceed on three fronts. First, we will evaluate the power savings that Cryogenic achieves for mass storage devices. Second we will integrate Cryogenic with a broader range of existing applications. Third, for integration into the Linux mainline kernel, Hans Peter Anvin (a Linux co-maintainer) suggested on the Linux Kernel Mailinglist (LKML) to add a Cryogenic-style timeout `fcntl` for all file descriptors instead of the device-specific `/dev/cryogenic/` file descriptors. This would further simplify the API.

Acknowledgments

This work was funded by the Deutsche Forschungsgemeinschaft (DFG) under ENP GR 3688/1-1 and the Agency for Innovation by Science and Technology in Flanders (IWT).

References

[1] "Google latitude," http://en.wikipedia.org/wiki/Google_Latitude, February 2014.

[2] "Google calendar," http://en.wikipedia.org/wiki/Google_Calendar, February 2014.

[3] A. Mahesri and V. Vardhan, "Power Consumption Breakdown on a Modern Laptop," in *Proceedings of the 4th International Conference on Power-Aware Computer Systems*, 2005.

[4] A. Hylick, R. Sohan, A. Rice, and B. Jones, "An Analysis of Hard Drive Energy Consumption," in *IEEE International Symposium on Modelling, Analysis and Simulation of Computers and Telecommunication Systems*, 2008.

[5] R. Kravets and P. Krishnan, "Power Management Techniques for Mobile Communication," in *Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, 1998.

[6] P. Reviriego, J. A. Hernández, D. Larrabeiti, and J. A. Maestro, "Performance Evaluation of Energy Efficient Ethernet," in *IEEE Communications Letters*, 2009.

[7] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich, "Energy Management in Mobile Devices with the Cinder Operating System," in *Sixth Conference on Computer Systems*, 2011.

[8] S. M. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich, "Apprehending Joule Thieves with Cinder," in *1st ACM Workshop on Networking, Systems and Applications for Mobile Handhelds*, 2009.

[9] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making Information Flow Explicit in HiStar," in *7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[10] Microsoft Corporation, "Windows Timer Coalescing," 2009.

[11] Microsoft Corporation, "Timers, Timer Resolution and Development of Efficient Code," 2010.

[12] Apple Inc., "OS X Mavericks Core Technologies Overview," 2013.

[13] D. Domingo, R. Landmann, and J. Reed, "Red Hat Enterprise Linux 6 Power Management Guide," 2010.

[14] P. Greenhalgh, "big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7," 2011.

[15] B. Jeff, "Advances in big.LIITLE Technology for Power and Energy Savings," 2012.

[16] R. Randhawa, "Software Techniques for ARM big.LITTLE Systems," 2013.

[17] S. Nikolaidis, N. Kavvadias, T. Laopoulos, L. Bisdounis, and S. Blionas, "Instruction Level Energy Modeling for Pipelined Processors," in *International Workshop on Power And Timing Modeling, Optimization and Simulation*, 2003.

[18] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[19] A. Morales, "Cryogenic: Enabling Power-Aware Applications on Linux," Master's thesis, Technische Universität München, 2014.