CELF PM Requirements 2006 (draft)

4/9/2006

Mark Gross mark.gross@intel.com CELF power management working group chair.

Table of Contents

Forward

The goal of this document is to capture the items from the Wiki page http://tree.celinuxforum.org/CelfPubWiki/CELFPmRequirements2006 that are of most importance to CELF or sufficiently well defined that actual work can be defined.

Please see this wiki for additional ideas that may not be captured in this document.

Introduction

This document presents the requirements for Linux Power Management from the perspective of Consumer Electronics device developers. The requirements vary from well defined feature requests to calls for analysis, benchmarking tools and methods.

CE products have different, power management needs than laptop computers. These requirements attempt to capture and communicate some of these. Additionally there is a good chance that the features needed for CE power management today will become PM features for tomorrows desktop and laptop systems.

Introduction to CE Power management

Power management for CE devices is different from power management for desktop, servers and laptop computers, even though there can be some overlap in power management methods. The differences tend to be application specific and platform support.

- CE devices tend to be application specific. They are designed to do only a few things well, such as play back video or provide the UI for a PDA or cell phone. They are constrained to running only a few core applications at all costs to the point that if the device cannot execute any of the core functions, then it should power off completely.

- There is also a huge difference in control logic for clock/voltage scaling between PCs and CE devices. In PCs CPU frequency is often scaled independently from the rest of the system. In an embedded world this is hardly the case. CPU frequency is often synchronous with the bus frequency. A number of peripherals connected to the bus can derive frequencies from the bus clock and therefore may need to be reprogrammed in case the bus frequency is changed (all of this as part of the CPU frequency change). Besides there can be several masters on the bus (CPUs, DMAs, etc). In such systems scaling a CPU clock has huge influence on the system performance and should be used with care. Such decisions should be based on much more information than simply idle time on one of the CPUs. Apart from CPU frequency scaling, embedded system often scale the bus frequency to squeeze the energy consumption even further. In practice, there are numerous dependencies between different clocks and voltages in the system, which are typically known to the system designer only.

- CE devices can have thermal constraints not common to other platforms. Hand held devices cannot allow high skin temperatures. It should be kept in mind that some computer components, e.g. CPU's, can get hot very quickly but take a long time to cool off.

- CE devices can have different performance value systems compared to laptops, desktops and servers. For instance the time it takes to launch a non-critical task may have latencies that would be unacceptable on a laptop or server, and yet is not a problem for a CE application. Conversely latencies that would be acceptable for a

laptop operation could be way out of operational or usability limits for a CE application. It all depends on the application the CE device is implementing.

Partitioning of requirements

The requirements have fallen into the following categories:

- Interface (kernel and user mode)
- Platform Throttling
- Process / OS Throttling
- Low power kernel processing
- Sleep state support
- System load prediction
- Measurement and benchmark

The following sections will explore each of these categories and provide specific requirements or issues that could be investigated further.

Interface (kernel and user mode)

It is important to bring structure in to power management designs for Linux based CE devices by providing a generalizing API/framework. Sorting out the system dependencies and expressing them in a generalized way is one of the most critical requirements to PM in CE domain.

Kernel and user mode API's ACPI independent

Most CE platforms do not include ACPI platform firmware interfaces. It is important that efforts be taken to make sure that power management API's do not explicitly or implicitly assume ACPI support or behaviors.

Without this it becomes difficult to reuse power management solutions across platforms and architectures. The result will be a forking of power management designs across architectures.

Throttling Power Parameter API's Outside CPUFREQ

More than CPUFREQ is needed to control the power parameters for todays CE platforms. PowerOP creates a new machine-level API that manages arbitrary hardware power/performance parameters. This API can be used by both CPUFREQ and by other power management mechanisms that wish to explicitly manage additional parameters, usually for embedded systems. CPUFREQ would then call the PowerOP layer to effect changes to hardware registers, etc. in response to changes in the "cpu speed" abstraction that CPUFREQ manages. Embedded power policy stacks, such as DPM, would also call this layer in response to changes in interfaces used by those stacks (usually lower-level abstractions, perhaps directly exposing the hardware registers).

This would allow both CPUFREQ and other power management software to share the same hardware-specific code, assuming there continues to be a place for both types of PM interfaces, one for desktop/laptop/server systems and one for embedded systems.

The linux-pm community has also discussed adding comprehensive power policy management frameworks that could subsume the functionality that was discussed for PowerOP.

More metric APIs

Today we only sample idle time, what other things could be sampled and used as control input for policies?

- more platform metric API's
    - fan speed
    - thermal static
    - thermal rate of change
    - power load
    - battery static
    - battery rate of change
- more system / OS load metric API's
    - fork latency
    - average time spent in TASK_UNINTERRUPTIBLE for specific tasks
    - application WFI (Wait For Interrupt) scheduling latency

- scheduler load
- lock contention
- dead line headroom
- other stuff?

Platform Throttling

Today we have basic CPU throttling. CPUFREQ is an OK framework for this but we need more and analogous infrastructure for throttling other parts of the platform. Additionally we only have 4 basic governors, user, max performance, min performance, and 2 idle time controlled CPU frequency switching policies.

Throttling on non-PC platforms is more complex and requires more platform knowledge to be exported to the governor than currently exists. For instance, to implement core voltage scaling from within a CPUFREQ governor core voltage state will need to be exported via the cpufreq_driver structure.

The implementation of the platform capabilities within ACPI on CE hardware is non-trivial. However, without adequate interfaces to the architecture-dependent code there will not be much opportunity for a standardized PM solution for CE classes of devices.

CPUFREQ needs more governor options

Beside the fact that CPUFREQ fails to export the needed data and control interface to the architecture-independent code for Frequency and Voltage throttling, there is a lot more that could be done with what is currently available. There should be more governors.

- RT deadline governor: This would limit the aggressiveness of the throttling based on periodic feedback from some process. Think of a media player providing decoding heart beats to the the governor such that the cpu is not throttled past the point where frames are dropped.
- UI responsiveness governor: this is similar to the above, only for limiting the throttling based on the latency of some selected tasks to become scheduled in response to an interrupt.
- Thermal: this would throttle the system such that temperatures do not exceed some value.
- Fan control: this would throttle based on keeping the fans off.
- core voltage: if you want to maximize power use on PXA and other platforms you simply must throttle core voltage and cpu frequencies together. There are a number of

power management policies that are blocked from implementation on mainline Linux kernels without this. For example:
http://www.intel.com/design/pca/applicationsprocessors/whitepapers/300577.htm

CPUFREQ extended to core voltage control

Today CPUFREQ is very clock-speed centric. For many systems one can control both core frequency and voltage. For some systems changing core voltage is a more expensive operation than changing frequency. To enable effective CPU frequency and core voltage control we need to extend the design of CPUFREQ to include the notion of target voltage.

Temperature based throttling

Many portable devices are made for human touch. Usability needs include preventing the device from overheating and preventing burning the user's hands. Platform vendors will be including temperature sensors in these devices, and the OS and application will need to have some policies in place to use it.

Non-ACPI architecture specific power control

For platforms that cannot afford the overhead of adding ACPI platform firmware / BIOS support for power management, there is a need for the architecture and sub architecture support of more or less equivalent capabilities.

The task of throttling the CPU frequency on some platforms without breaking the serial port connection, violating some constraints with memory controllers or interface specification to other parts on the CE platform can be a significant challenge.

ACPI and platform firmware / BIOS take care of this type of stuff for developers underneath the OS on most laptop, desktop and server platforms from Intel. However, for CE devices there is very little support available in the arch and sub-arch kernel trees.

Platform vendors and sub-architecture maintainers need to add support for controlling the power parameters for their platforms. As multiple architectures define implementations, then a common set of control interfaces needs to evolve.

Process / OS Throttling

Sometimes it is not enough to throttle the platform. Throttling the work load is the next level of control. Support for process level throttling is needed to in order to take power management to the next level and provide systems that will maintain thermal or battery constraints.

The OS should be aware of trigger points for HW protection, where the HW will shut off power and loose user data. If the platform is approaching one of these triggers, and the OS should throttle itself to avoid loss of data.

Dual core / SMP CPU throttling

After you have throttled the hardware back and you have more than one core under utilized. It may be useful to change the idle processing on those cores such that they enter higher latency idle states by not scheduling tasks / sending interrupts to those cores.

Sometimes you cannot just shut off CPU's. But you can not schedule work on them, and use specific CPU instructions that are hi-latency and lower power than the normal idle process.

Some multi-core systems must throttle both cores symmetrically. This means that if your work load is low enough to not need the other core, the most you can do is to simply not send any interrupts to it and to avoid scheduling any tasks to run on it.

Doing this effectively from a PM governor (say a CPUFREQ governor) has some interesting challenges in coupling scheduler behavior to PM design.

Process scheduling control based on power states

The idea is to provide a background thread policy that would keep tasks out of the run state as a function of PM state. For instance, a laptop will sometimes runs updatedb and other cron jobs when it's on battery, and it would be better if updatedb and similar process would sleep until the system is re-tethered to the wall.

This type of scheduler based throttling could be applied to thermal system constraints as well. Many laptops get quite hot when some web pages opened (flash banner adds burning mips). It would be interesting to throttle such threads in the scheduler.

There needs to be structure and support for developing scheduler based power management implementations enabling the following:

- Policies for tethered vs. un-tethered
- Policies for thermal control (e.g. keep the fan from starting up by not running some cron job.)
- Scheduler policy classes that schedule tasks based on some TBD policy manager Run levels tied to PM policy

Asymmetric Suspend Resume

The ability to suspend a system under one power state and resume it under different conditions and selectively resume the suspended processes is needed. One user scenario is

the following: the system is tethered running at full power and more or less operating assuming it had unlimited battery. The user suspends, and takes the device to a coffee shop or airplane and resumes. This time under battery power. It would be desirable to selectively resume processes that are required to run when under battery. The selection criterion for resume could include IP address, or MAC of current DHCP server, or powered state (AC / Battery).

Lower power kernel processing

The kernel does a lot of bookkeeping and processing on its own as a result of design choices for things like process accounting, scheduler design for CPU bound multi-processing, and other things. This area of PM requirements focuses on reducing the number of instructions the kernel does over time, as well as things that happen in the kernel that get in the way of putting the CE platform into a possible lower power state for longer times.

One thing to keep in mind is that for CE platforms and applications it is sometimes acceptable to violate POSIX.

Low power idle

Many platforms provide hardware support for different types of idle states. It important to have standardized ways for extending the amount of time the cpu doesn't execute instructions dynamically. Idle processing functions could be sorted by latency and the specific idle function used by the idle loop controlled by some PM policy interfaces.

A number of CPUs and ASSPs currently used in mobile devices support different sleep states, although often the software support to exploit these in not available. Typically, a CPU has a low power mode that can be entered when the OS idle loop/idle task runs and that is exited by an interrupt. The latency associated with this "wait for interrupt" (WFI) state is low; no state saving is required and the OS can resume from where it left off. More recent CPUs have included support for other sleep states that are in-between WFI and the "system suspend" state in which power to the CPU is removed and external RAM is used to save/restore state. For example, the ARM11 family introduced a "dormant" power state, in which power is removed from the CPU core but not its cache RAMs, enabling a faster "warm start". TI's OMAP family of ASSPs include a number of sleep states, where progressively more and more of the device is powered down. ASSPs often include a "System Controller" that manages clocks and power gating, providing a number of clocking regimes, for example:

- Off - system suspended
- Crystal oscillator - CPU runs, limited range of peripherals available,
- (probably) no SDRAM available
- Main oscillator - CPU, more peripherals, SDRAM available

- PLL - maximum performance, system devices available

The system may also permit frequency/voltage scaling within one of these clocking regimes - this is the area that CPUfreq and other proprietary software is aiming to tackle.

Runtime selection of idle states

Today we can change what type of idle processing happens at boot up time. It would be useful to be able to change the latency of the idle processing in support of lower power idle processing.

For ACPI platforms this can be implemented using C states, but for non-ACPI platforms the lower power idle states need to be entered by the OS explicitly, by explicit machine instructions.

A policy framework for controlling which platform idle states are entered from idle is needed for choosing what type of processing to use in idle.

Tick-less idle

When in a high latency lower power idle state its important to avoid timer interrupts waking up the CPU to do nothing useful when it is in an otherwise deep idle state.

It is noted that Variable Scheduling Timeout (VST) is a more general tick-less idle solution as it goes farther to remove the periodic timer ticks from the scheduler design all together. It is mentioned here for completeness.

Reducing Tick Overhead

The timer tick processing is bloated for CE applications where each instruction results in power loss. Efforts to minimize the work done by the kernel when processing timer ticks would help a lot for CE and embedded applications.

It would be acceptable to the CE and embedded application developers to sacrifice some POSIX compatibility for this. Of course such an implementation would require compile time switches.

Sleep state support

- Need both suspend to RAM and suspend to disk
- Generalized suspend /resume implementations that can be easily extended to suspend to flash.

- Need to define quality metric and test cases for suspend resume and drive community toward such goals
- Video recovery (VGA platforms only / set top box)
- Better debug logging for suspend / resume success and failure
- Low latency suspend / resume control
    - Resume takes a long time to resume, and then even longer before the OS stops thrashing the VM / page cache.
    - The resume thrash costs battery (measurement data is needed for this)
    - The resume thrash breaks audio play back for a significant amount to time.

Suspend to Flash

Sony is interested in this, and has done some work in this area. It would give all the benefits of Suspend to RAM with the added advantage of being able to remove power from external RAM too. A project within Samsung has looked at using NAND Flash as a swap device, to make use of existing Suspend to Disk code.

Suspend to RAM

There is support for this in some platforms - for example, Intel PXA and TI OMAP family parts. Power can be removed from the CPU when it is in a sleep mode because its state has previously been saved to external memory (usually SDRAM). The SDRAM itself is in a self-refresh mode. Exit from sleep is via a warm reset.

All suspend mechanisms rely on the Linux power management framework (see kernel/power/pm.c) with architecture- and platform-specific code underneath.

System load prediction

The ability to predict system throughput and latency capability and needs is one of the largest gaps for the implementation of good power management designs today.

The system designer is responsible for estimating the workload and performance requirements of main applications on the system components. As this knowledge is not explicitly present in the system, the PM results in guessing strategies (such as idle-time monitoring). Whereas for PC with many applications, heuristic strategies are the only way to go for PM. In embedded CE applications the fine-tuning of applications for PM is a normal practice.

The application could explicitly share information with help PM subsystem to improve the "guessing" accuracy. The requirement to provide interfaces from PM framework to applications to export their performance requirements and/or monitor their real-time activity is very important.

For instance MPlayer could tell the PM subsystem about the video playback performance directly to the PM subsystem over a standard interface to maintain optimum PM while not dropping frames.

Today we only use the kstat data to monitor idle. We could do better if we knew:

- Process contention for locks
- Size of task list
- CPU event counters
- Number of tasks in run state
- Number of tasks waiting on IO
- Number of niced tasks
- Number of RT tasks
- RT processing deadline times
- Latency from interrupt to process schedule (UI event)
- Latency of wake from interrupt from idle.
- Interrupt rates
- average time of key tasks spent in TASK_UNINTERRUPTIBLE state

Need more ideas to be discussed.

Measurement and benchmark

This requirement is for the development of well defined methodologies for measurement of power consumption and battery life as well as representative work loads and power management scenarios.

PM measuring methodology definition

How to measure power

- Understand the trade-offs WRT where power is measured
- Warn about issues with some test configurations.
- How to measure battery life
- Understanding of differences in battery technologies

PM benchmark work loads

- well defined target platforms
- well defined work loads
- well defined PM scenarios and use cases highlighting the need for multiple power management techniques
- Good PM robustness benchmark
- Suspend/resume latency benchmark
- Regular PM benchmark reporting