CHAPTER 6

Power Consumption by Video Applications

After discussing video compression, quality, and performance aspects in the previous chapters, in this chapter we turn our attention to another dimension of video application tuning: power consumption. Power consumption needs to be considered together with those other dimensions; tradeoffs are often made in favor of tuning one of these dimensions, based on the needs of the application and with a view toward providing the best user experience. Therefore, we first introduce the concept of power consumption and view its limits on typical modern devices, then we follow with a discussion of common media workloads and usages on consumer platforms. After that, we briefly introduce various criteria for power-aware platform designs.

Within this general setup, the chapter deals with three major topics: power management, power optimization, and power measurement considerations. In regard to power management, we present the standards and management approaches used by the operating system and by the processor. For discussion of power optimization, we present various approaches, including architectural, algorithmic, and system integration optimization. The third topic, power measurement, takes us into the realm of measurement methodologies and considerations.

Besides these three main topics, this chapter also briefly introduces several power measurement tools and applications, along with their advantages and limitations.

Power Consumption and Its Limits

In today's mobile world order, we face ever-increasing desires for compelling user experiences, wearable interfaces, wireless connectivity, all-day computing, and--most critical--higher performance. At the same time, there's high demand for decreasing form factor, lower weight, and quieter battery-powered devices. Such seemingly contradictory requirements present unique challenges: not only do newer mobile devices need extensive battery lives but they also are harder to cool, as they cannot afford bulky fans.

Most important, they need to operate in a limited power envelope. The main concern from the consumer's point of view, however, is having a system with better battery life, which is cost-effective over the device's life. Therefore, power limits are a fundamental consideration in the design of modern computing devices.

Power limits are usually expressed in terms of *thermal design power* (TDP), which is the maximum amount of heat generated for which the cooling requirement is accounted for in the design. For example, TDP is the maximum allowed power dissipation for a platform. The TDP is often broken down into the power consumption of individual components, such as the CPU, the GPU, and so on. Table 6-1 lists typical TDPs for various processor models:

Table 6-1. *Typical Power Envelopes*

Type	TDP (Watts)	Comment
Desktop server/ workstation	47W–120W	High-performance workstations and servers
All-in-one	47W-65W	Common usage
Laptop	35W	
Ultrabook class	17W	Recent areas of focus to decrease TDP
Tablet/Phablet	8W	
Smartphone	<4W	

Although it is important to reduce cost and conserve power for the desktop workstations and servers, these platforms essentially are not limited by the availability of power. Consumer devices and platforms such as portable tablets, phablets, and smartphones, on the other hand, use size-constrained batteries for their power supply.

A major drawback of these devices is that batteries in tablets often drain down before an 8- or 9-hour cross-Atlantic flight is over, and in the case of smartphones, often need a recharge every day. As such, today's market demands over 10 hours of battery life for a tablet to enable users to enjoy a long flight and more than 24 hours for a smartphone for active use. Additionally, the users of these devices and platforms may choose to use them in many different ways, some of which may not be power-efficient. We need, therefore, to understand various power-saving aspects for media applications on these typical consumer platforms.

Power is the amount of energy consumed per unit time, and it is typically expressed in terms of Joules per second, or watts. The switching power dissipated by a chip using static CMOS gates, such as the power consumed by the CPU of a mobile computing device with a capacitance C_{dyn}, running at a frequency f and at a voltage V, is approximately as follows:

$$P = AC_{dyn}V^2 f + P_s \qquad \text{(Equation 6-1)}$$

Here, P_S is the static power component introduced mainly due to leakage, and A is an activity constant related to whether or not the processor is active or asleep, or under a gating condition such as clock gating. For a given processor, C_{dyn} is a fixed value; however, *V* and *f* can vary considerably. The formula is not perfect because practical devices as CPUs are not manufactured with 100 percent CMOS and there is special circuitry involved. Also, the static leakage current is not always the same, resulting in variations in the latter part of the equation, which become significant for low-power devices.

Despite the imprecision of the equation, it is still useful for showing how altering the system design will affect power. Running the processor of a device at a higher clock frequency results in better performance; however, as Equation 6-1 implies, at a lower frequency it results in less heat dissipation and consequently lower power consumption. In other words, power consumption not only dictates the performance, it also impacts the battery life.

In today's technology, the power or energy supply for various electronic devices and platforms usually comes from one of three major sources:

- An electrical outlet, commonly known as the "AC power source"
- A so-called *SMPS* unit, commonly known as the "DC power source"
- A rechargeable battery

A switch mode power supply (SMPS) unit rectifies and filters the AC mains input so as to obtain DC voltage, which is then switched on and off at a high frequency—speed in the order of hundreds of KHz to 1 MHz.

The high-frequency switching enables the use of inexpensive and lightweight transformers, inductors, and capacitors circuitry for a subsequent voltage step-down, rectification, and filtering to output a clean and stable DC power supply. Typically, an SMPS is used as a computer power supply.

For mobile usage, rechargeable batteries supply energy to an increasing number of electronic devices, including almost all multimedia devices and platforms. However, because of the change in internal resistance during charging and discharging, rechargeable batteries degrade over time. The lifetime of a rechargeable battery, aka "the battery life," depends on the number of cycles of charge/discharge, until eventually the battery can no longer hold an effective charge.

Batteries are rated in watt-hours (or ampere-hours multiplied by the voltage). Measuring system power consumption in watts gives a good idea of how many hours a battery will work before needing a recharge. This measure of battery life is usually important to consumers of today's electronic devices.

Media Workloads on Consumer Platforms

One of the main goals in designing a typical consumer electronic device or platform is to make it as user-friendly as possible. This implies that an important design consideration is how such devices are to be used. Nowadays, rapid integration of multimedia functionalities in modern mobile devices has become commonplace.

For example, a smartphone is expected to work not only as a wireless phone and a communication device but also should accommodate applications such as calendars, clocks, and calculators, combining to function as a productivity device. It should also serve as a navigation device with a compass, a GPS, and maps; and it should function as an entertainment device, with games and multimedia applications. In addition to these usages, as an educational platform the device is used for digital storytelling or a virtual classroom.

Human interaction with these devices calls for high-resolution cameras, high-speed wireless connection to the Internet, and voice, touch, and gesture input. On the output side, high-fidelity speakers, high-resolution displays, fast processing, and low power consumption are common expectations.

However, supporting multiple high-end functionalities often conflicts with the need to save power. Increases in battery capacity only partially address the problem, as that increase is not sufficient to keep up with the expansion of multimedia integration and enhanced user experience. Analyzing and understanding the nature of these multimedia workloads will help toward achieving the performance and power optimizations within the constraints just mentioned.

In popular consumer electronics press reviews, power data is often measured and analyzed using non-multimedia benchmark workloads, such as Kraken, Sunspider, and Octane Javascript benchmarks. Usually these benchmark applications focus on usage of the device as a computing platform, leaving unexamined the power consumption of the device's other usages. Yet, often these other applications are not optimized for power, or may be optimized to achieve higher performance only on certain platforms and operating systems. This fails to recognize the impact of task migration and resource sharing between the processing units. With the increasing availability of integrated processor graphics platforms, it becomes necessary to include media usages and applications in such analyses.

In the following section, we discuss some of the common media usages and applications.

Media Usages

Multimedia applications are characteristically power-hungry. With the demand for more and more features, requirements for power consumption are increasingly raised to higher levels. Moreover, some usages may need additional instances of an application, or more than one application running at a time.

Mobile devices used as entertainment platforms have typically run two main types of applications: gaming and media. The 2D and 3D video games are the most popular, but many other media applications are also in demand on these devices. Among them, the following are notable:

- Still image capture
- Still image preview/view finder
- Wireless display or Miracast: clone mode or extended mode
- Browser-based video streaming
- Video recording and dual video recording
- Video playback
- Audio playback
- Internet browsing
- Videophone and video chat
- Video conferencing

- Video transcoding
- Video email and multimedia messaging
- Video upload to Internet
- Video editing
- Augmented reality
- Productivity applications

Most of these usages are implemented via special software applications, and some may benefit from hardware acceleration if supported by the platform.

Intel processors are noteworthy for supporting such hardware acceleration through the integrated processor graphics, both serving as a general-purpose computing platform and fulfilling needs for special-purpose applications. The integration of graphics units into the central processor allows mobile devices to eliminate bigger video cards and customized video processors, thereby maintaining a small size suitable for mobile usage.

On many mobile devices, some combinations of multimedia applications are used simultaneously. For example, audio playback may continue when a user is browsing the Internet, or video playback may be complemented with simultaneous video recording. Some of these applications use common hardware blocks for the hardware acceleration of video codec and processing tasks; simultaneous operation of such hardware blocks is interesting from a system resource scheduling and utilization point of view.

One example of such complex system behavior is multi-party video conferencing; another is video delivery over Miracast Wireless Display. Wi-Fi certified Miracast is an industry-standard solution for seamlessly displaying multimedia between devices, without needing cables or a network connection. It enables users to view pictures from a smartphone on a big screen television, or to watch live programs from a home cable box on a tablet. The ability to connect is within the device using Wi-Fi Direct, so separate Wi-Fi access is not necessary.[1]

Figure 6-1 shows a usage model of the Miracast application.

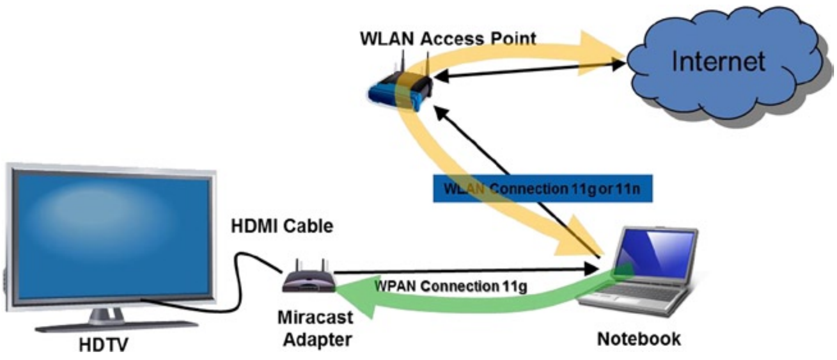


Figure 6-1. *Video delivery over Miracast*

[1]For details, see www.wi-fi.org/discover-wi-fi/wi-fi-certified-miracast.

To better understand the power consumption by video applications, let us analyze one of the multimedia usages in detail: video delivery over Miracast Wireless Display.[2]

Wireless Display (WiDi) is an Intel technology originally developed to achieve the same goals as Miracast. Now that Miracast has become the industry standard, it has been supported in Intel (R) Wireless Display (TM) since version 3.5. The goal of this application is to provide a premium-content capable wireless display solution that allows a PC user, or a handheld device user, to remotely display audiovisual content over a wireless link to a remote display. In other words, the notebook or smartphone receives a video from the Internet via the wireless local area network or captures a video using the local camera. The video is played on the device using a local playback application. A special firmware for Miracast Wireless Display then captures the screen of the device and performs hardware-accelerated video encoding so as to send the compressed bit stream via Wi-Fi data exchange technology to a Miracast adapter. The adapter performs a decoding of the bit stream to HDMI format and sends it to the display device through an HDMI cable connection. The end-to-end block diagram is shown in Figure 6-2.

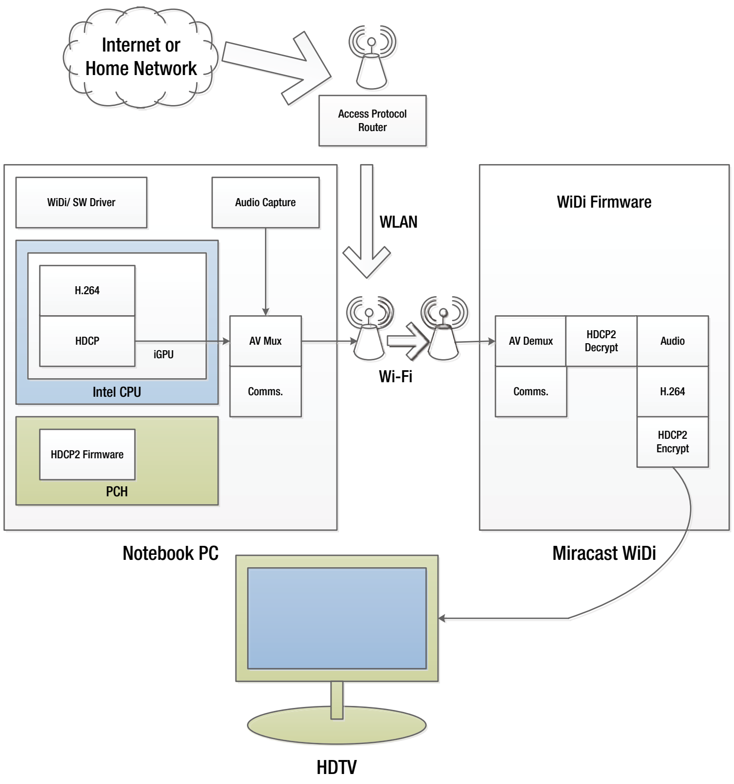


Figure 6-2. *Miracast Wireless Display end to-end block diagram*

[2]For details, see www-ssl.intel.com/content/www/us/en/architecture-and-technology/intel-wireless-display.html.

In Figure 6-2, the major power-consuming hardware modules are the CPU, the PCH, the video codec in the integrated GPU, the hardware-accelerated content protection module, the memory, the local display of the notebook, and the remote HDTV display. The Miracast Wireless Display adapter mainly runs the wireless display firmware and consumes a smaller amount of power. The typical distribution of power consumption in this example is shown in Figure 6-3.

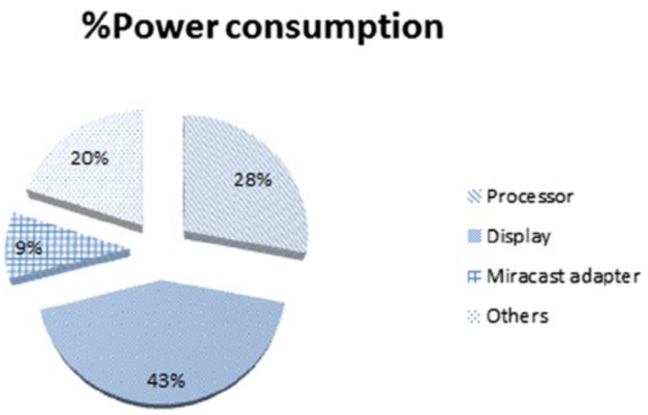


Figure 6-3. *Typical distribution of power consumption by components in a Miracast Wireless Display application*

As can be seen in Figure 6-3, usually the bulk of the power is consumed by the display, which in this example consumes about 1.5 times as much power as the processor. The Miracast adapter itself consumes a moderate amount—in this example, approximately 9 percent of the total power consumed by the application. Due to the complex nature of this application, a careful balance should be maintained between performance needs and power consumption, so that the appropriate optimizations and tradeoffs can be made so as to obtain a satisfactory user experience.

Another common multimedia application is video playback along with associated audio. A detailed analysis of this application is provided in Agrawal et al.[3] Here, we just note that by performing hardware acceleration of the media playback, the overall power consumption is reduced from ~20W to ~5W, while the power consumption profile is also changed significantly. Various tasks within the media playback pipeline get a performance boost from hardware acceleration, as these are offloaded from the CPU to the special-purpose fixed-function hardware with better power-performance characteristics.

Analysis of these applications enables identification of the modules that are prime candidates for power optimization. For example, some power optimization can be achieved by migrating tasks like color-space conversion from the display unit to the GPU. Some Intel platforms are capable of such features, achieving a high level of power optimization. Various power optimization techniques are discussed in detail later in this chapter.

[3]A. Agrawal, T. Huff, S. Potluri, W. Cheung, A. Thakur, J. Holland, and V. Degalahal, "Power Efficient Multimedia Playback on Mobile Platform," *Intel Technology Journal* 15, no. 2 (2011): 82–100.

Power-Aware Designs

Power consumption is a function of both hardware and software efficiency. Therefore, performance gains or power savings increasingly depend on improving that efficiency. This is done typically in terms of *performance per watt*, which eventually translates to *performance per dollar*. Performance per watt is the quantity of computation that can be delivered by a computing system for every watt of power consumed. Today's platforms aim to achieve high scores in this measure by incorporating "power awareness" in the design process, as significant consideration is given to the cost of energy in computing environments.

In power-aware designs, typically the power savings are achieved by employing a divide-and-conquer policy: the system is divided into several independent power domains, and only the active domains are supplied with power. Depending on the active state of each domain, intelligent management of power achieves the optimum power solutions. Also, optimization within each domain is done with a view to gaining an edge in power saving and value proposition of a system.

Power awareness is important not only in the hardware design but also in the applications, so as to maximize the performance per dollar. Toward this end, most operating systems provide power-management features. As applications know the utilization pattern of various hardware resources and tasks, better power management can be achieved if the applications can provide appropriate "hints" to the power-management units. Furthermore, power awareness of those applications can yield software-level optimizations, such as context-aware power optimizations, complexity reduction, and memory transfer reduction. These techniques are discussed later in the chapter in regard to power optimization.

Power-Management Considerations

The goal of power management in both computers and computer peripherals, such as monitors and printers, is to turn off the power or switch the system to a low-power state when it is inactive. Power management in computing platforms provides many benefits, including increased battery life, lower heat emission, lower carbon footprint, and prolonged life of devices such as display panels and hard disk drivers.[4]

Power management happens on various constituent hardware devices that may be available in a computer system (aka "the system"); among them are the BIOS, central processing unit (CPU), hard disk drive (HDD), graphics controller, universal serial bus (USB), network, and display. It is also possible to monitor and manage the power use to various parts of memory, such as dynamic random access memory (DRAM) and non-volatile flash memory, but this is more complex and less common. Some examples of power management are listed in Table 6-2; some of these are discussed in subsequent sections of this chapter.

[4]M. Vats and I. Verma, *Linux Power Management: IEGD Considerations* (Intel Corporation, 2010). Available at `www.intel.com/content/dam/www/public/us/en/documents/white-papers/linux-power-mgmt-paper.pdf`.

Table 6-2. *Power Management Features*

Device/ Component	Power Management Features
BIOS	CPU settings (e.g., CPU states enabled, CPU fan throttling), platform settings (e.g., thermal high/low watermarks, chassis fan throttling, etc.)
CPU	HLT (halt instruction in x86 for CPU to halt until next external interrupt is fired), Stop clock, Intel SpeedStep (aka dynamic frequency scaling)
Display	Blanking, dimming, power saver mode, efficient energy use as specified in the Energy Star international standard
Graphics Controller	Power down to intermediate state, power shutoff
Hard drive/ CD-ROM	Spin down
Network/ NIC	Wake on LAN
USB	Power state transition of devices such as mouse, USB drives, etc.; wake on access (e.g., mouse movement)

There may be special power-management hardware or software available. Typically, hardware power management in the processor involves management of various CPU states (aka *C*-states), such as the core *C*-states, the module *C*-states, the package *C*-states, and so on. (Details of the *C*-states are described in the next section.) On the other hand, software power management in the operating system or in the driver involves tasks like CPU core offline, CPU core shielding, CPU load balancing, interrupt load balancing, CPU frequency governing etc. With the introduction of the integrated graphics processing units (iGPU), primarily in Intel platforms, various GPU states are also important considerations for power management.

ACPI and Power Management

The Advanced Configuration and Power Interface (ACPI) specification is an open standard adopted in the industry for system-level configuration and management of I/O devices and resources by the operating system, including power management. Originally proposed by Intel, Microsoft, and Toshiba in 1996, the specification effort was later joined by HP and Phoenix. The latest ACPI specification version 5 was published in 2011.

With wider adoption in the industry, it became necessary to support many operating systems and processor architectures. Toward this end, in 2013, the standards body agreed to merge future developments with the Unified Extensible Firmware Interface (UEFI) forum, which is an alliance of leading technology companies, including the original ACPI participants and major industry players like AMD, Apple, Dell, IBM, and Lenovo. All recent computers and portable computing devices have ACPI support.

ACPI Power States

To the user, a computer system appears as either ON or OFF. However, the system may support multiple power states, as defined in the ACPI specification. ACPI compliance indicates that the system supports the defined power management states, but such compliance does not promise the most power-efficient design. In addition, a system can have power-management features and tuning capabilities without being ACPI compliant.

According to the ACPI specification, the devices of a computer system are exposed to the operating system in a consistent manner. As such, for system-level power management, the ACPI defines power draw states for the individual devices (known as the device states), as well as the overall computer system (known as the global states or the system sleep states). The operating system and the devices can query and set these states. Important system buses such as the peripheral component interconnect (PCI) bus, may take advantage of these states.

Global States

The ACPI specification defines four possible global "Gx" states and six possible sleep "Sx" states for an ACPI-compliant computer-system (Table 6-3). However, some systems or devices may not be capable of all states.

Table 6-3. *ACPI Global States*

State	Gx	Sx	Description
Active	G0	S0	The system is fully usable. CPUs are active. Devices may or may not be active, and can possibly enter a lower power state. There is a subset of S0, called "Away mode," where monitor is off, but background tasks are running.
Sleep	G1	S1	The system appears to be off. Power consumption is reduced. All the processor caches are flushed, and the CPU(s) stops executing instructions. The power to the CPU(s) and RAM is maintained. Nonessential devices may be powered off. This state is rarely used.
		S2	The system appears to be off. CPU is powered off. Dirty cache is flushed to RAM. Similar to S1, the S2 state is also rarely used.
		S3	Commonly known as *standby*, *sleep,* or *Suspend-to-RAM* (STR). The system appears to be off. System context is maintained on the system DRAM. All power is shut to the noncritical circuits, but RAM power is retained. Transition to S0 takes longer than S2 and S1, respectively.

(*contiuned*)

Table 6-3. (*contiuned*)

State	Gx	Sx	Description
Hibernation		S4	Known as *hibernation* or *Suspend-to-Disk* (STD). The system appears to be off. Power consumption is reduced to the lowest level. System context is maintained on the disk, preserving the state of the OS, applications, and open documents. Contents of the main memory are saved to non-volatile memory such as a hard drive, and the system is powered down, except for the logic to resume.
Soft Off	G2	S5	The system appears to be off. System context is not maintained. Some components may remain powered, so the computer can wake from input from a keyboard, mouse, LAN, or USB device. The working context can be restored if it is stored on nonvolatile memory. All power is shut, except for the logic required to restart. Full boot is required to restart.
Mechanical Off	G3		The system is completely off and consumes no power. A full reboot is required for the system to return to the active state.

Device States

The power capabilities of all device hardware are not the same. For example, the LAN adapters will have the capability to wake the system; the audio hardware might permit streaming while the system is in standby mode, and so on. Some devices can be subdivided into functional units with independent power control. Furthermore, some devices, such as the keyboard, mice, modems, and LAN adapters, have the capability to wake up the system from a sleep state, while the devices are asleep themselves. Such capability is possible by the fact that the hardware for these devices must draw a small seeping current and be equipped to detect the external wake event.

The ACPI specification defines the device-dependent D-states as shown in Table 6-4.

Table 6-4. *Device States*

Dx	Subset of Dx	Description
D0		Fully ON and operating.
D1-D2		Intermediate power states. Definition varies by device.
D3	Hot	Device is off and unresponsive to bus, but the system is ON. Device is still connected to power. *D3 Hot* has auxiliary power enabling a higher power state. A transition from D0 to D3 implies D3 Hot.
	Cold	No power to the device—both the device and system are OFF. It is possible for the device to consume *trickle* power, but a wake event is needed to move the device and the system back to D0 and/or S0 states.

The ACPI defines the states D0 through D3, and provides a subdivision of D3 into D3 hot and D3 cold. Some devices or operating systems don't distinguish between D3 hot and cold, and treats D3 as having no power to the device. However, Windows 8 explicitly tracks D3 hot and D3 cold. The D1 and D2 states are optional, but if they are used properly, they would provide better cleanliness when the device is idle.

Power Management by the Operating System

Modern operating systems customarily offer many power management features. Linux, Windows, OS X, and Android all support more intelligent power management using the newer ACPI standard, rather than the old BIOS controlled Advanced Power Management (APM). However, all major operating systems have been providing stable support for basic power-management features such as notification of power events to the user space—for example, battery status indication, suspend the CPU when idle, and so on.

In the following sections, we discuss power management by the Linux and the Windows operating systems. In the context of Linux power management, three important components are mentioned: the X Window, the Window Manager, and the Intel Embedded Graphics Driver (IEGD). The Windows power management discourse includes the Windows power requirements, power policy, the Windows driver model, and the Windows driver framework. There's also a brief description of device power management under Windows 8, followed by a discussion on how to deal with power requests.

Linux Power Management

Linux supports both the older APM and the newer ACPI power management implementations. APM focuses on basic system and OS power management, with much of the power-management policy controlled at the BIOS level; whereas an APM driver acts as an interface between the BIOS and the Linux OS, as power-management events pass between the BIOS and OS. Devices are notified of these events so they can respond appropriately.

The ACPI provides greater flexibility in power management and platform configuration, and allows for platform independence and OS control over power-management events. In addition to the power-management policies, ACPI supports policies for responding to thermal events (e.g., fans), physical movement events (e.g., buttons or lids), CPU states, power source (e.g., battery, AC power supply), and the like.

Power-management software manages state transitions along with device drivers and applications. Device drivers are responsible for saving device states before putting them into their low-power states and then restoring the device state when the system becomes active. Generally, applications are not involved in power-management state transitions. A few specialized softwares, such as the IEGD for Linux, deal directly with some devices in order to handle state transitions. Besides the IEGD, there are a few common software technologies in Linux, including the X Window system, the Window managers, and several open-source processes such as `/sys/power/state` and `/proc/acpi/event`, which also provide some part of Linux power management.

The X Window

The X Window system is supported by many operating systems, including Linux, Solaris, and HP-UX. It provides graphics capabilities to the OS and supports user-level, system-level, and/or critical standby, suspend, and resume. In the APM implementation, the X-server controls the power-management events. In ACPI implementation, the X Window system handles the graphics messages as a user process, but the system-wide power-management events like suspend/resume are handled by a kernel mode driver.

Window Managers

Window managers on Linux are user-level processes that provide the graphical user interface and also deliver reliable power management of the operating system. Among the many supported windows managers in Linux are two popular window managers, GNOME and KDE. In GNOME, power management uses the hardware abstraction layer (HAL) and involves open-source platform power management built on an open-source messaging interface called DBUS, while KDE3 provides a proprietary power-management solution named KPowersave.

Intel Embedded Graphics Driver

In the Intel Embedded Graphics Driver (IEGD) power management, a kernel mode driver helps the Linux kernel manage the power. It is also responsible for graphics device initialization and resource allocation. In order for you to clearly understand the flow of a power event, here are the main parts of the Suspend to RAM example.[5]

The Suspend to RAM starts when a power-management *event* occurs in the platform, such as when a button is pressed or a window manager option is triggered, and the operating system is notified of the event. Usually the Linux operating system employs a protocol to communicate an event between a software component and the Linux kernel. Using such protocols, the OS (typically via the Window manager) commands the kernel to go to a lower power state, at which point the Linux kernel starts the suspend procedure by notifying the X-Server driver.

The X display must switch to console mode before going into a lower power state. With ACPI implemented in the Linux kernel, this switch happens by the X-Server driver's calling the *Leave virtual terminal* function, when the IEGD process saves the graphics state and registers information. The Linux kernel then freezes all user processes, including the X Window process. Now the kernel is ready to check which devices are ready for the suspend operation, and it calls the suspend function of each device driver (if implemented) in order to put the device clocks to D3 mode--effectively putting all devices into a lower power state. At this point only the Linux kernel code is running, which freezes all other active processors except the one where the code is running.

[5]Ibid.

Following execution of the kernel-side suspend code, two ACPI methods--namely, PTS (*Prepare-to-Sleep*) and GTS (*Going-to-Sleep*) are executed, the results of which may not be apparent to the Linux kernel. However, before actually going to sleep, the kernel writes the address of the kernel wakeup code to a location in the Fixed ACPI Description Table (FADT). This enables the kernel to properly wake up upon receiving the restore command.

The restore command usually results from a user event, such as a keystroke, mouse movement, or pressing the power button, which turns the system on. Once on, the system jumps to the BIOS start address, performs housekeeping tasks such as setting up the memory controller, and then scans the ACPI status register to get the indication to RAM that the system was previously suspended. If *video repost* is supported, during resume operation the BIOS also calls this function to re-execute the video BIOS (vBIOS) code, thereby providing a full restart of the vBIOS.

The system then jumps to the address programmed earlier, as indicated by the ACPI register's status and the FADT. The wakeup address leads to the kernel code execution, putting the CPU back into protected mode and restoring the register states. From this point, the rest of the wakeup process traverses the reverse path of the suspend process. The ACPI WAK method is called, all the drivers are resumed, and user space is restarted. If running, the X-server driver calls the *Enter virtual terminal* function, and the IEGD restores the graphics device state and register information. After saving the console mode, the X-server driver re-enters the GUI, thereby completing a successful wakeup.

Windows Power Management

Power management in the Windows operating system, particularly Windows 8, has significant improvements in this area compared to previous Windows versions.

Power Requirements

In versions earlier than Windows 8, the power requirements involved supporting the common ACPI states, such as the S3 and S4 states, mainly on mobile personal computer platforms. However, Windows 8 aimed to standardize on a single power requirement model across all platforms including desktop, server, mobile laptops, tablets, and phones. While one of the goals was to improve the battery life of portable platforms, Windows 8 applies the smartphone power model to all platforms for quick standby-to-ready transitions, and ensures that the hidden applications consume minimal or no resources.

To this end, Windows 8 defines the requirements listed in Table 6-5.[6]

[6]J. Lozano, *Windows 8 Power Management.* (StarJourney Training and Seminars, 2013.)

Table 6-5. *Windows 8 Power Requirements*

Requirement Type	Requirements
System Power Requirements	1. Maximum battery life should be achieved with minimum energy consumption. 2. The delay for startup and shutdown should be minimal. 3. Power decisions should be intelligently made—for example, a device that is not in a best position to change the system power state should not do so. 4. Capabilities should be available to adjust fans or driver motors on-demand for quiet operation. 5. All requirements should be met in a platform independent manner.
Device Power Requirements	6. Devices, especially for portable systems, must be extremely power conscious. 7. Devices should be aggressive in power savings: a. Should provide just-in-time capabilities. b. Low transition latency to higher states. c. When possible, the device logic should be partitioned into separate power buses so that portions of a device can be turned off as needed. d. Should support connected standby as appropriate for quick connection.
Windows Hardware Certification Requirements	8. The Windows HCK tests require that all devices must support S3 and S4 without refusing system sleep request. 9. Standby and connected standby must last for days. 10. Device must queue up and not lose the I/O request while in D1-D3 states.

Power Policy

Power policies (also known as *power plans* or *power schemes*) are preferences defined by the operating system for the choice of system and BIOS settings that affect energy consumption. For each power policy, two different settings are usually set by the operating system by default, one with battery power supply, the other with AC power supply. In order to preserve battery as much as possible, the settings with the battery power supply are geared toward saving power more aggressively. Windows defines three power policies, by default:

- **Performance mode:** In performance mode, the system attempts to deliver maximum performance without regard to power consumption.
- **Balanced mode:** In this mode, the operating system attempts to reach a balance between performance and power.
- **Power saver mode:** In this mode, the operating system attempts to save maximum power in order to preserve battery life, even sacrificing some performance.

Users may create or modify the default plans but the power policies are protected by the access control list. Systems administrators may override a user's selection of power policies. On Windows, a user may use an applet called "powercfg.cpl" to view and edit a power policy. A console version of the applet, called "powercfg.exe," is also available, which permits changing the access control list permissions.

Application software can obtain a notification of the power policy by registering for the power plan and can use power policies in various ways:

- Tune the application behavior based on the user's current power policy.
- Modify the application behavior in response to a change in power policy.
- Move to a different power policy as required by the application.

The Windows Driver Model

In the Windows Drive Model (WDM), the operating system sends *requests* to the drivers to order the devices to a higher or lower power state. Upon receiving such requests, a driver only saves or restores the state, keeping track of the current and next power states of the device, while a structure called Physical Device Object (PDO) performs the work of actually increasing or lowering the power to the device.

However, this arrangement is problematic, as the model requires the drivers to implement a state machine to handle the power IRPs (I/O request packets), and may result in unwanted complexity due to the time needed to perform a power state transition. For example, for a *power down* request, the driver saves the state of the device in memory, and then passes the request down to the PDO, which subsequently removes power from the device; only then can it mark the request as *completed*. However, during a *power up* request that may follow, the driver must first pass the request to the PDO, which then restores the power *before* restoring the device state, and informs the driver to mark the request as *completed*. To overcome this difficulty, the Windows driver framework (WDF) was proposed.

The Windows Driver Framework

In order to simplify power management within a driver, Windows introduced the concept of *events* in the latest Windows Driver Framework (WDF) driver model. In this model, there are optional *event handler* functions in the driver, whereby the framework calls the event handlers at the appropriate time to handle a power transition, thereby eliminating the need for a complex state machine.

Windows 8 offers a new, more granular way to address the power needs of functions on multifunction devices in the form of a *power framework* called *PoFx*. Additionally, it introduces the concept of *connected standby*, allowing a powered-off device to occasionally connect to outside world and refresh state or data for various applications. The primary benefit is a quick recovery from standby state to ON state, as if the system had been awake the whole time. At the same time, the power cost is low enough to allow the system to be in standby state for days.

Device Power Management in Windows 8

In Windows 8, in response to a query from the plug-and-play (PnP) manager, the device drivers announce their device's power capabilities. A data structure called `DEVICE_CAPABILITIES` is programmed by the driver, indicating the information as shown in Table 6-6.

Table 6-6. *Device Capabilities Structure*

Field	Function
Device D1 and D2	Indicates whether the device supports D1, or D2, or both.
Wake from Dx	Indicates whether the device supports waking from a Dx state.
Device state	Defines the Dx state corresponding to each Sx state.
DxLatency	Nominal transition time to D0.

There is latency for the devices when moving from Dx to D0, as the devices require a small period of time before they can become operational again. The latency is longer for the higher Dx states, so that a transition from D3 to D0 would take the longest time. Furthermore, a device needs to be operational before it can respond to new requests—for example, a hard disk must spin up before it can be slowed down again. The latency is announced by the driver via the `DEVICE_CAPABILITIES` data structure.

■ **Note** A device state transition may not be worthwhile if sufficient time is not spent in the lower power state, as it is possible that the transition itself would consume more power than when the device had been left in a particular state.

In order to manage the device power, the Windows Power Manager needs to know the transition latency of each device, which can vary for different invocations even for the same device. Therefore, only a nominal value is indicated by the driver. The Windows OS controls the time gap between a query and a set power request, during which time the device sleeps. A high value for this time gap would increase the sleep time, while a low value would cause the OS to give up on powering down the device.

Dealing with Power Requests

There are kernel mode data structures called *I/O request packets* (IRPs) that are used by the Windows Driver Model (WDM) and the Windows NT device drivers to communicate with the operating system and between each other. IRPs are typically created by the I/O Manager in response to I/O requests from the user mode. In Windows 2000, two new managers were added: the plug-and-play (PnP) and Power manager, which also create IRPs. Furthermore, IRPs can be created by drivers and then passed to other drivers.

In Windows 8, the Power Manager sends *requests*--that is, the power IRPs--to the device drivers, ordering them to change the power state of the relevant devices. Power IRPs use the major IRP data structure `IRP_MJ_POWER`, with the following four possible minor codes:

- `IRP_MN_QUERY_POWER`: A query to determine the capability of the device to safely enter a new requested Dx or Sx state, or a shutdown or restart of the device. If the device is capable of the transition at a given time, the driver should queue any further request that is contrary to the transition before announcing the capability, as a `SET` request typically follows a `QUERY` request.
- `IRP_MN_SET_POWER`: An order to move the device to a new Dx state or respond to a new Sx state. Generally, device drivers carry out a `SET` request without fail; the exception is bus drivers such as USB drivers, which may return a failure if the device is in the process of being removed. Drivers serve a `SET` request by requesting appropriate change to the device power state, saving context when moving to a lower power state, and restoring context when transitioning to a higher power state.
- `IRP_MN_WAIT_WAKE`: A request to the device driver to enable the device hardware so that an external wake event can awaken the entire system. One such request may be kept in a pending state at any given time until the external event occurs; upon occurrence of the event, the driver returns a success. If the device can no longer wake the system, the driver returns a failure and the Power Manager cancels the request.
- `IRP_MN_POWER_SEQUENCE`: A query for the D1-D3 counters--that is, the number of times the device has actually been in a lower power state. The difference between the count before and the count after a sleep request would tell the Power Manager whether the device did get a chance to go to a lower power state, or if it was prohibited by a long latency, so that the Power Manager can take appropriate action and possibly not issue a sleep request for the device.

For driver developers, one of the difficulties in calling the various power IRPs is determining when to call them. The Kernel Mode Driver Framework (KMDF) in Windows 8 implements numerous state machines and event handlers, including those for power management. It simplifies the task of power management by calling the event handlers at the appropriate time. Typical power event handlers include: D0 entry, D0 exit, device power state change, device arm wake from S0/Sx, and device power policy state change.

Power Management by the Processor

For fine-grained power management, modern Intel processors support several partitions of *voltage islands* created through on-die power switches. The Intel Smart Power Technology (Intel SPT) and Intel Smart Idle Technology (Intel SIT) software determine the most power efficient state for the platform, and provide guidance to turn ON or OFF different voltage islands on the processor at any given time. Upon receiving a direction to go into a lower power state, the processor waits for all partitions with shared voltage to reach a safe point before making the requested state change.

CPU States (*C*-states)

The CPU is not always active. Some applications need inputs from the system or the user, during which the CPU gets an opportunity to wait and become idle. While the CPU is idle or running low-intensity applications, it is not necessary to keep all the cores of the CPU powered up. The CPU operating states (*C*-states) are the capability of an idle processor to turn off unused components to save power.

For multi-core processors, the *C*-states can be applied at a package level or at a core level. For example, when a single threaded application is run on a quad-core processor, only one core is busy and the other three cores can be in low-power, deeper *C*-states. When the task is completed, no core is busy and the entire package can enter a low-power state.

The ACPI specification defines several low-power idle states for the processor core. When a processor runs in the *C0* state, it is working. A processor running in any other *C*-state is idle. Higher *C*-state numbers represent deeper CPU sleep states. At numerically higher *C*-states, more power-saving actions, such as stopping the processor clock, stopping interrupts, and so on, are taken. However, higher *C*-states also have the disadvantage of longer exit and entry latencies, resulting in slower wakeup times. For a deeper understanding, see the brief descriptions of various *C*-states of an Intel Atom processor, given in Table 6-7.

Table 6-7. *Cx State Definitions, Intel Atom Processor Z2760*

State	Function	Description
C0	Full ON	This is the only state that runs software. All clocks are running and the processor core is active. The processor can service *snoops* and maintain cache coherency in this state. All power management for interfaces, clock gating, etc., are controlled at the unit level.
C1	Auto Halt	The first level of power reduction occurs when the core processor executes an Auto-Halt instruction. This stops the execution of the instruction stream and greatly reduces the core processor's power consumption. The core processor can service snoops and maintain cache coherency in this state. The processor's North Complex logic does not explicitly distinguish *C1* from *C0*.

(*continued*)

Table 6-7. (*continued*)

State	Function	Description
C2	Stop Grant	The next level of power reduction occurs when the core processor is placed into the *Stop Grant* state. The core processor can service *snoops* and maintain cache coherency in this state. The North Complex only supports receiving a single *Stop Grant*. Entry into the *C2* state will occur after the core processor requests *C2* (or deeper). Upon detection of a break event, *C2* state will be exited, entering the *C0* state. Processor must ensure that the PLLs are awake and the memory will be out of self-refresh at this point.
C4	Deeper Sleep	In this state, the core processor shuts down its PLL and cannot handle *snoop* requests. The core processor voltage regulator is also told to reduce the processor's voltage. During the *C4* state, the North Complex continues to handle traffic to memory so long as this traffic does not require a *snoop* (i.e., no coherent traffic requests are serviced). The *C4* state is entered by receiving a *C4* request from the core processor/OS. The exit from *C4* occurs when the North Complex detects a *snoop*-able event or a break event, which would cause it to wake up the core processor and initiate the sequence to return to the *C0* state.
C6	Deep Power Down	Prior to entering the *C6* state, the core processor flushes its cache and saves its core context to a special on-die *SRAM* on a different power plane. Once the *C6* entry sequence has completed, the core processor's voltage can be completely shut off. The key difference for the North Complex logic between the *C4* state and the *C6* state is that since the core processor's cache is empty, there is no need to perform *snoops* on the internal front side bus (FSB). This means that bus master events (which would cause a popup from the *C4* state to the *C2* state) can be allowed to flow unhindered during the *C6* state. However, the core processor must still be returned to the *C0* state to service interrupts. A residency counter is read by the core processor to enable an intelligent promotion/demotion based on energy awareness of transitions and history of residencies/transitions.

Source: Data Sheet, Intel Corporation, October 2012. *www.intel.com/content/dam/www/public/us/en/documents/product-briefs/atom-z2760-datasheet.pdf*.

Performance States (*P*-states)

Processor designers have realized that running the CPU at a fixed frequency and voltage setting is not efficient for all applications; in fact, some applications do not need to run at the operating point defined by the highest rated frequency and voltage settings. For such applications there is a power-saving opportunity by moving to a lower operating point.

Processor performance states (*P*-states) are the capability of a processor to switch between different supported operating frequencies and voltages to modulate power consumption. The ACPI defines several processor-specific *P*-states for power management, in order to configure the system to react to system workloads in a power-efficient manner. Numerically higher *P*-states represent slower processor speeds, as well as lower power consumption. For example, a processor in *P3* state will run more slowly and use less power than a processor running at *P1* state.

While a device or processor is in operation and not idling (D0 and C0, respectively), it can be in one of several *P*-states. *P0* is always the highest-performance state, while *P1* to *Pn* are successively lower-performance states, where *n* can be up to 16 depending on implementation.

P-states are used on the principles of dynamic voltage or frequency scaling, and are available in the market as the SpeedStep for Intel processors, the PowerNow! for AMD processors, and the PowerSaver for VIA processors.

P-states differ from *C*-states in that *C*-states are idle states, while *P*-states are operational states. This means that with the exception of *C0*, where the CPU is active and busy doing something, a *C*-state is an idle state; and shutting down the CPU in the higher *C*-states makes sense, since the CPU is not doing anything. On the other hand, *P*-states are operational states, meaning that the CPU can be doing useful work in any *P*-state. *C*-states and *P*-states are also orthogonal—that is, each state can vary independently of the other. When the system resumes *C0*, it returns to the operating frequency and voltage defined by that *P*-state.

Although *P*-states are related to the CPU *clock frequency*, they are not the same. The CPU clock frequency is a measure of how fast the CPU's main clock signal goes up or down, which may be a measure of performance, as higher performance would generally mean using a higher clock frequency (except for memory-bound tasks). However, this is backward looking, as you can measure the average clock frequency only after some clock cycles have passed. On the other hand, *P*-state is a performance state the OS would like to see on a certain CPU, so *P*-states are forward looking. Generally, higher clock frequency results in higher power consumption.

When a CPU is idle (i.e., at higher *C*-states), the frequency should be zero (or very low) regardless of the *P*-state the OS requests. However, note that all the cores on a current-generation CPU package share the same voltage, for practical reasons. Since it is not efficient to run different cores at different frequencies while maintaining the same voltage, all active cores will share the same clock frequency at any given time. However, some of the cores may be idle and should have zero frequency. As the OS requests a certain *P*-state for a logical processor, it is only possible to keep a core at zero frequency when none of the cores is busy and all cores are kept at the same zero frequency.

While using a global voltage supply for all the cores leads to the situation where certain *P*-state requests cannot be met, separate local voltage supplies for each core would be cost-prohibitive. A tradeoff concerning global- and local-voltage platforms is to adopt multi-core architecture with different *voltage islands*, in which several cores on a voltage island share the same but adjustable supply voltage. An example of this is available in Intel's many-core platform called the *single-chip cloud computer*. Another example is the use of a *fully integrated voltage regulator* (FIVR), available in some processors, that allows per core *P*-states so that cores can be operated at frequencies independent of each other.

Just as the integrated GPUs behave similarly to the CPU, *C*-states and *P*-states have been defined for the GPU like the CPU *C*-states and *P*-states. These are known as *render C-states* (*RC*-states) and *render P-states* (*RP*-states).

Turbo

It is possible to *over-clock* a CPU, which means running a core at a higher frequency than that specified by the manufacturer. In practice, this behavior is possible due to the existence of multiple cores and the so-called *turbo* feature resulting from the power budget. Turbo is a unique case in which the frequency and voltage are increased, so the turbo state functions as an opposite to the various *P*-states, where voltage and frequency are decreased. The CPU enters turbo state while operating below certain parameter specifications, which allows it some headroom to boost the performance without infringing on other design specifications. The parameters include the number of active cores, processor temperature, and estimated current and power consumption. If a processor operates below the limit of such parameters and the workload demands additional performance, the processor frequency will automatically dynamically increase until an upper limit is reached. In turbo state, complex algorithms concurrently manage the current, the power, and the temperature with a view toward maximize the performance and the power efficiency.

To better understand a turbo scenario, let us consider an example. Suppose a quad-core CPU has a TDP limit of 35W, so that each core has a maximum of 8.75W power budget. If three of the four cores are idle, the only operational core can utilize the whole 35W of the power budget and can "turbo" up to a much higher frequency than would be possible with a less than 9W power budget. Similarly, if two cores are active, they use the same higher frequency and share the power budget while the idle cores do not consume any energy.

A maximum turbo frequency is the highest possible frequency achievable when conditions allow the processor to enter turbo mode. The frequency of Intel Turbo Boost Technology varies depending on workload, hardware, software, and overall system configuration. A request for a *P*-state corresponding to a *turbo-range* frequency may or may not be possible to satisfy, owing to varying power characteristics. So a compromise is made because of many factors, such as what the other cores and the GPU are doing, what thermal state the processor is at, and so on. This behavior also varies over time. Therefore, as the operating frequency is time-varying and dependent on *C*-state selection policy and graphics subsystem, selecting a *P*-state value does not guarantee a particular performance state.

In general, the *P1* state corresponds to the highest guaranteed performance state that can be requested by an OS. However, the OS can request an opportunistic state, namely *P0*, with higher performance. When the power and thermal budget are available, this state allows the processor configuration where one or more cores can operate at a higher frequency than the guaranteed *P1* frequency. A processor can include multiple so-called turbo mode frequencies above the *P1* frequency.

Some processors expose a large turbo range and typically grant all cores the maximum possible turbo frequency when the cores seek to turbo. However, not all applications can effectively use increased core frequency to the same extent. Differences may arise from varying memory access patterns, from possible cache contention, or similar sources. So allowing all cores to be at a highest level of turbo mode can unnecessarily consume power. In order to combat such inefficiencies, techniques have been proposed to efficiently enable one or more cores to independently operate at a selected turbo mode frequency.[7] One of the techniques periodically analyzes all cores granted turbo mode to determine whether their frequency should be increased, decreased, or left unchanged based on whether the core has been classified as stalled or not over the observation interval.

Thermal States (*T*-States)

In order to prevent potential damage from overheating, processors usually have thermal protection mechanisms, where, in an effort to decrease the energy dissipation, the processor throttles by turning the processor clocks off and then back on according to a pre-determined duty cycle. The thermal states, or *T*-states, are defined to control such throttling in order to reduce power, and they can be applied to individual processor cores. *T*-states may ignore performance impacts, as their primary reason is to reduce power for thermal reasons. There are eight *T*-states, from 0 to 7, while the active state is T_0. These states are not commonly used for power management.

The Voltage-Frequency Curve

It is important to understand the relationship between voltage and frequency for processors, as both CPU and integrated GPU follow such relationship in order to scale. A *P*-state requested by the operating system is in fact a particular operating point on the *V-F* curve.

As can be seen in Figure 6-4, the voltage vs. frequency curve tends to have an inflection point, at which the voltage starts to scale with the frequency.

[7]M. K. Bhandaru and E. J. Dehaemer, U. S. Patent No. US 20130346774, A1, 2013. *Providing energy efficient turbo operation of a processor.* Available at `www.google.com/patents/WO2013137859A1?cl=en`.

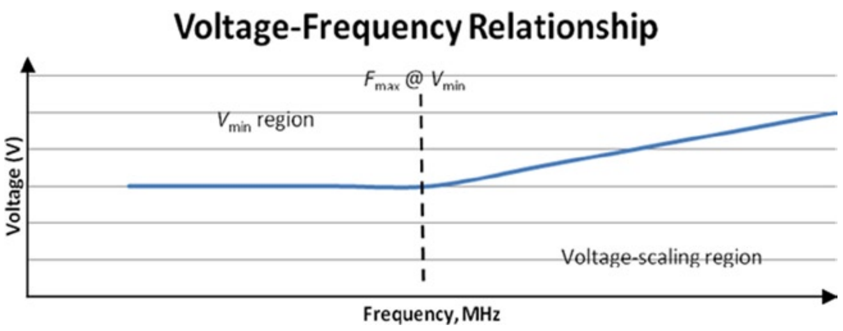


Figure 6-4. *Voltage-frequency relationship of a typical processor*

Up to this point, a minimum voltage, $V_{min,}$ is required to make the circuit operational regardless of the frequency change. The maximum frequency F_{max} at the minimum voltage V_{min} is the highest frequency at which the processor part can operate at V_{min}. This is the point where power efficiency is the best, as we see from the power-frequency relationship shown in Figure 6-5. Increasing the frequency beyond F_{max} requires increased voltage supply for the circuit to be operational. At the voltage-scaling region, the required voltage scales are reasonably linear with frequency. This region offers power-reduction opportunities, as discussed later.

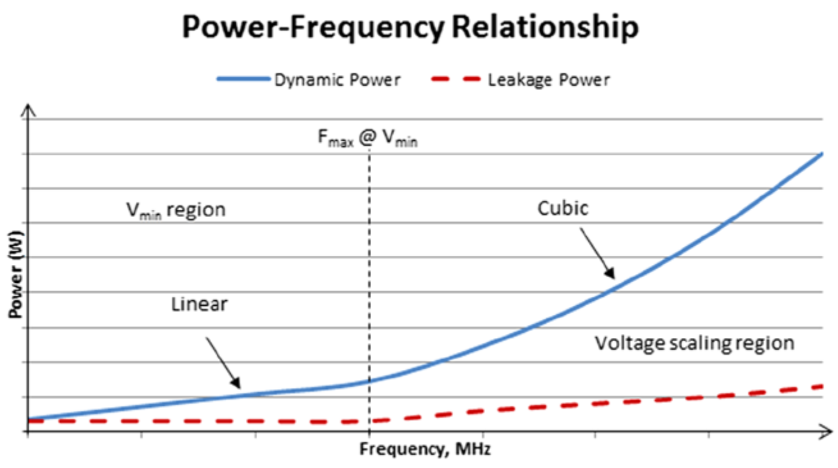


Figure 6-5. *Power-frequency relationship and optimization opportunities*

Figure 6-5 shows the power vs. frequency relationship and power optimization opportunities in a typical processor.

In the V_{min} region, power does not fall as fast as frequency; as dynamic power falls with frequency, the leakage power stays constant. On the other hand, in the voltage-scaling region, power increases much faster than frequency, as voltage scales are roughly linear with frequency; dynamic power goes up as V^2f and leakage goes up roughly as V^3. The leakage power depends only on the small amount of leakage current, and it follows almost the same pattern as the voltage curve with respect to frequency.

Power Optimizations

Let's recall the power equation:

$$\textit{Total power} = \textit{leakage power} + A\, C_{dyn}\, V^2 f, \qquad \text{(Equation 6-2)}$$

Here, A is the activity, C_{dyn} is the dynamic capacitance, V is the voltage, and f is the operating frequency. From this equation it is easy to see that, in order to reduce power, the following approaches can be taken:

- Reduce voltage and frequency
- Reduce activity and C_{dyn}

As voltage increases approximately linearly with frequency in the voltage-scaling region, the term V^2f implies a cubic relationship for the power with respect to the frequency (see Figure 6-5). Therefore, reducing the voltage and/or the frequency results in a dramatic reduction in the power. The power that is conserved in such a manner can be given to other parts of the system so that the overall system operation can benefit.

However, the frequency reduction cannot help below F_{max} at V_{min} (below which voltage cannot be reduced, as there is a minimum voltage necessary for operation of the circuit). In the V_{min} region, the voltage stays constant, so reduction in frequency can yield very little power savings. At this point, only activity and C_{dyn} reduction can provide further power optimization. This calls for more efficient algorithms and micro-architecture design, as well as dynamically turning off unused portions of the circuit.

The above power-reduction considerations have given birth to new ideas and approaches of power optimization, including various *gating* optimizations and use of special-purpose heterogeneous hardware components, such as the integration of a GPU capable of multimedia processing, camera image processing, and so on. Overall, it is not a hardware-only problem; it requires careful consideration at the software micro-architecture level as well.

In general, good power optimization requires incorporation of many approaches, all working in harmony toward the goal of saving power and increasing battery life. From a systems engineering point of view, optimizations can be made in various individual power domains within the system by selectively turning off power to certain idle parts of the system.

Power optimizations can be done at various levels of the system. Typically, power optimization is done at the following levels:

- Architectural optimization
- Algorithmic optimization
- System integration optimization
- Application level optimization

In general, power optimization combines all of these approaches. Architectural optimizations deal with the optimization opportunities at the processor hardware level and try to obtain a suitable hardware-software partitioning. Algorithmic optimizations look for power-saving opportunities in system and application algorithms. For example, above the hardware and hardware abstraction layer, the graphics execution stack includes hierarchical layers of the application, the middleware, the operating system, and the graphics driver. Opportunities to save power exist within each layer and are exploited using algorithmic optimization.

Inter-layer optimization opportunities, however, are more complex and addresses inefficiencies by employing optimization at the system integration level. For example, efficiency can be improved by choosing to use fewer layers and by redefining the boundaries of the layers in order to find the most power-efficient places for the optimization. Furthermore, at the application level, load sharing between the CPU and the integrated GPU may be considered for reuse of power in one unit that is saved from the other, by running a task on the most power-efficient device. Discussions of these optimization techniques are follows in more detail.

Architectural Optimization

The techniques for optimizing power efficiency at the processor architecture level include:

- Hardware-software partitioning
- Dynamic voltage and frequency scaling
- Power gating
- Clock gating
- Slice gating
- Use of low-level cache

Hardware-Software Partitioning

There has been a paradigm shift in the approach to optimizing hardware and software interaction. The earlier philosophy was to obtain performance by removing execution bottlenecks. For example, if the CPU was the bottleneck in a graphics application, then the main power and performance tuning approach was to use a better CPU or to tune the CPU code to maximize graphics performance. However, processor architects soon realized that removing execution bottlenecks alone is not sufficient; it is also prohibitive from a power-consumption perspective to run all subparts of the system simultaneously at maximum performance, as various components compete for their share of the power envelope.

This realization opened two optimization opportunities: (a) power saved in one subpart can be applied to another; and (b) unused power can be applied to turbo behavior. Accordingly, considerations of power management for the overall system and shifting power between the CPU, graphics, and other subsystem are taken into account. As such, the new philosophy of hardware-software interaction aims not only to eliminate performance bottlenecks but also to continue tuning to increase efficiency and save power as well. For example, focus is now given to design goals including:

- Reducing CPU processing
- Optimizing driver codes to use the fewest CPU instructions to accomplish a task
- Simplifying the device driver interface to match the hardware interface to minimize the command transformation costs
- Using special-purpose hardware for some tasks with a balanced approach for task execution

Fixed-purpose hardware is often implemented with a minimum number of gates that switch states or toggle between states to perform certain specific tasks. As dynamic power consumption is a function of the number of gates that are switching, and as less switching means less dynamic power consumption, it is beneficial to perform the same task on the special-purpose fixed function hardware, as opposed to general-purpose hardware that may not use the optimum number of switching gates for that particular task. Obviously, if the nature of the task changes, the special-purpose hardware cannot be used, as it is often not flexible enough to accommodate changes in how it is used. In this case, power saving may be achieved by sacrificing flexibility of tasks, and often by migrating workloads from general-purpose hardware to fixed-function hardware. Careful design of hardware-software partitioning is necessary to save power in this manner, and non-programmable tasks may be migrated from general-purpose execution units to fixed-purpose hardware. For example, video processing algorithms that are run using GPU hardware designed explicitly for that task typically consume less power than running those same algorithms as software running on the CPU.

Dynamic Voltage and Frequency Scaling

To decrease power consumption, the CPU core voltage, the clock rate, or both can be altered, at the price of potentially lower performance, using dynamic voltage and/or frequency scaling. Alternatively, higher performance can be achieved at the expense of higher power consumption. However, as mentioned in the *P*-state discussion earlier, with the advancement of generations of CPU technology, this process is becoming increasingly complex, and there are many contributing factors, such as the load balancing among the multiple CPU cores and the GPU, thermal states, and so on. On the other hand, new techniques beyond dynamic voltage and frequency scaling are emerging to combat the challenges.

Power Gating

Processors can selectively power off internal circuitry by not supplying current to the parts of the circuitry that are not in use, and thereby reduce power consumption. This can be accomplished either by hardware or software. Examples of this technique include Intel Core and AMD CoolCore, where in a multi-processor environment only certain core processors (or part of the circuit in those processors) are active at a given time.

Power gating generally affects the design more than clock gating, and may introduce longer entry and exit latency from a gated state. Architectural tradeoffs are generally considered between the amount of power saved and the latency involved. Another important consideration is the area used for power gating circuitry if implemented in hardware. For example, in fine-grained power gating, switching transistors may be incorporated into the standard cell logic, but it still has a large area penalty and difficult independent voltage control per cell. On the other hand, in coarse-grained power gating, grid style sleep transistors drive cells locally through shared virtual power networks, and save area at the expense of sensitivity.

For quick wakeup from a power gated state, sometimes *retention* registers may be used for critical applications. These registers are always powered up, but they have special low-leakage circuits as they hold data of the main register of the power gated block, enabling quick reactivation.

Clock Gating

Clock gating is a popular technique for reducing dynamic power dissipation by using less switching logic and by turning off unnecessary clock circuitry, thereby saving power needed to switch states that are not useful at a given time. Clock gating can be implemented in RTL code or can be manually inserted into the design.

There are several forms of clock gating, ranging from manual to fully automated, that may be applied together or separately, depending on the optimization. On the one hand, there is the manual clock gating performed by driver software, where a driver manages and enables the various clocks used by an idle controller as needed. On the other hand, in automatic clock gating, the hardware may detect idle or no-workload states, and turn off a given clock if it is not needed. For example, on a particular board, an internal bus might use automatic clock gating so that it is temporarily gated off until the processor or a DMA engine needs to use it, while other peripherals on that bus might be permanently gated off if they are unused or unsupported on that board.

Slice Gating

Current Intel processors such as the fourth-generation core processor architecture or later have integrated graphics processing units that have arrays of programmable execution units (EUs) in addition to fixed-function hardware for specific tasks. The EUs, along with media samplers, are further arranged in slices. For example, some fourth-generation core SKUs have 40 EUs distributed between two equivalent slices, each containing 20 EUs and located in two different power domains. Figure 6-6 shows the slice structure of typical Intel fourth-generation core processor graphics execution units.

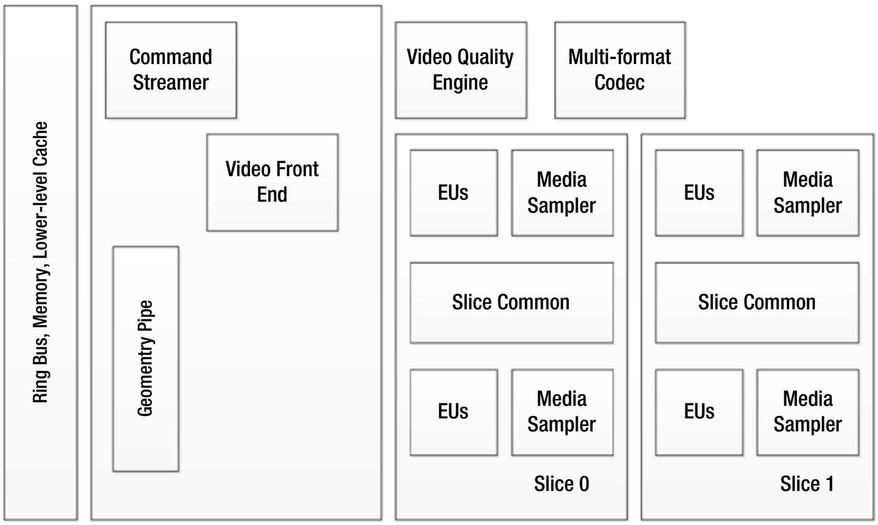


Figure 6-6. *Intel fourth-generation core processor graphics execution unit slice structure*

As hardware-accelerated multimedia tasks require many different assets in the graphics hardware, certain media tasks may place different demands on the media assets inside the slices, such as the EUs or the media samplers, and on the assets that are outside the slices, such as the Video Front End or Video Quality Engine. For some media workloads that require relatively little work from slice-based assets, the processor can shut down one slice to save leakage power. For example, for some media workloads, fewer than 20 EUs are needed, whereupon the driver software may power off one slice without affecting the performance. This is slice gating, also known as slice shutdown. The advantage of slice gating is that it maximizes power efficiency across a broad range of tasks.

Use of Low-level Cache

Memory power can be significantly reduced by using low-level caches and by designing algorithms to utilize these caches in an efficient manner. Video applications are typically compute-bound and not memory-bound, unless a memory-restricted system is used. Therefore, algorithms can take advantage of memory bandwidth reduction approaches, and thereby lower power consumption. For example, in the Intel core architecture, the cache is arranged in hierarchical levels, where both a low-level cache and a level-three cache are used. This enables power optimization, owing to the lower cost of memory access.

Algorithmic Optimization

The goal of algorithmic optimization is to reduce execution time by running the tasks fast and turning off processing units whenever they are not necessary. This can be achieved in many ways, including:

- As power consumption is proportional to execution residency, running less code in the CPU translates to less power consumption. So, performing code optimization of key software modules contributes to algorithmic optimization.
- Processing tasks can be offloaded to dedicated power-efficient fixed-function media hardware blocks as supported by the platform.
- In order to perform various stages in a pipeline of tasks for a given usage, it is generally necessary to expand the data into some intermediate representation within a stage. Storing such data requires a much larger bandwidth to memory and caches. The cost of memory transactions in terms of power consumption can be reduced by minimizing the memory bandwidth. Bandwidth reduction techniques are, therefore, important considerations for algorithmic optimization.
- The concurrency available among various stages or substages of the pipeline may be explored and appropriate parallelization approaches may be made to reduce the execution time.
- The I/O operations can be optimized by appropriate buffering to enable the packing of larger amounts of data followed by longer idle periods, as frequent short transfers do not give the modules a chance to power down for idle periods. Also, disk access latency and fragmentation in files should be taken into account for I/O optimization, as they may have significant impact in power consumption.
- Appropriate scheduling and coalescing of interrupts provide the opportunity to maximize idle time.
- All active tasks can be overlapped in all parts of the platform—for example, the CPU, the GPU, the I/O communication, and the storage.

Algorithmic optimization should be made with the power, performance, and quality tradeoffs in mind. Depending on the requirements of an application, while attempting to save power, attention should be paid to maintaining the performance and/or visual quality. A few common algorithmic optimization techniques are described in the following sections.

Computational Complexity Reduction

A computing device or system consumes very little power when it is not actively computing, as only the display engine needs to be awake; other compute engines may be temporarily in a sleeping state. The idea behind reducing computational complexity is to keep the system in a high power or busy state only as long as necessary, and to allow the system to return to idle state as often as possible. Improving the performance of an application can easily achieve power savings, as it allows the system to go back to idle state earlier because the work is done faster.

There are several approaches to computational complexity reduction, including algorithmic efficiency, active-duty cycle reduction, minimizing overheads such as busy-wait locks and synchronization, reducing the time spent in privileged mode, and improving the efficiency of I/O processing. We discuss some of these approaches next, but for a thorough treatment of them, see *Energy Aware Computing*.[8]

Selecting Efficient Data types

It is possible to optimize an algorithm that is heavy in floating point calculations by using integer arithmetic instead. For example, the calculation of discrete wavelet transforms using the lifting scheme usually involves a number of floating point operations. But the lifting coefficients can be implemented by rational numbers that are powers of 2, so that the floating point units in the data path can be replaced by integer arithmetic units.[9] This leads to power savings, as the hardware complexity is reduced.

Similarly, rearranging the code in a way suitable to take advantage of compiler optimization, or in a way where certain data dependency allows a computation to be done before entering a loop instead of inside the loop, can yield significant performance gain and thereby power savings. In an audio application example,[10] show some sine and cosine functions being repeatedly called on fixed values inside a busy loop; as the values are fixed, the computation can be made before entering the loop. This optimization yields about a 30 percent performance gain and also saves power.

In another example, motion vector and discrete cosine transform calculations were done on a vector of pixels instead of using each pixel separately,[11] which not only gives a 5 percent overall performance improvement in a software-only H.263 video encoder, but also provides power saving in two ways: by doing the computation faster, and by using improved memory access and cache coherency.

[8] B. Steigerwald, C. D. Lucero, C. Akella, and A. R. Agrawal, *Energy Aware Computing* (Intel Press, 2012).
[9] P. P. Dang and P. M. Chau, "Design of Low-Power Lifting Based Co-processor for Mobile Multimedia Applications," *Proceedings of SPIE* 5022 (2003): 733–44.
[10] Steigerwald et al., *Energy Aware Computing.*
[11] S. M. Akramullah, I. Ahmad, and M. L. Liou, "Optimization of H.263 Video Encoding Using a Single Processor Computer: Performance Tradeoffs and Benchmarking," *IEEE Transactions on Circuits and Systems for Video Technology* 11, no. 8 (August 2001): 901–15.

Code Parallelization and Optimization

Removing run-time inefficiency is also the goal of code parallelization and optimization. Multithreading, pipelining, vectorization, reducing the time spent in a privileged mode, and avoiding polling constructs are common techniques for code parallelization and optimization.

A properly threaded application that uses all available resources usually completes earlier than a single-threaded counterpart, and it is more likely to provide performance and power benefits. In this context, selecting the right synchronization primitives is also very important. Some applications, especially media applications, are particularly amenable to improvement using multithreading in a multi-core or multi-processor platform. In a multithreaded media playback example mentioned by Steigerwald et al.,[12] while almost linear performance scaling was achieved, the power consumption was also halved on a four-core processor at the same time, as all the four cores were busy running a balanced workload.

Similarly, the same operation on different data can be efficiently performed by using vector operations such as single-instruction multiple-data (SIMD) in the same clock cycle on a vector of data. Most modern processors support SIMD operations. The Intel Automatic Vectorizing Extensions (AVX) support eight 32-bit floating-point simultaneous operations in a single processor clock cycle. As Steigerwald et al.[13] claims, for media playback applications, use of such SIMD operations can result in approximately 30 percent less power consumption.

In Listing 6-1, note the following Direct3D query structure and the polling construct that only burns CPU cycles, resulting in wasted power.

Listing 6-1. Polling Example with Direct3D Query Structure (Power Inefficient)

```
        while ( S_OK != pDeviceContext->GetData( pQuery, &queryData,
sizeof(UINT64), 0 ) )
        {
                sleep (0); // wait until data is available
        }
```

It is better to use blocking constructs to suspend the CPU thread. However, Windows 7 DirectX is nonblocking. Although a blocking solution using the OS primitives would avoid the busy-wait loop, this approach would also add latency and performance penalty, and may not be appropriate for some applications. Instead, a software work-around may be used, where a heuristic algorithm detects the `GetData()` call in a loop. In an example of such work around,[14] up to 3.7W power was reduced without performance degradation. Listing 6-2 shows the concept of the workaround:

[12] Steigerwald et al., *Energy Aware Computing.*
[13]Ibid.
[14]D. Blythe, "Technology Insight: Building Power Efficient Graphics Software," Intel Developer Forum, 2012.

Listing 6-2. Example of an Alternative to the Polling Construct

```
INT32 numClocksBetweenCalls = 0;
INT32 averageClocks = 0;
INT32 count = 0;

// Begin Detect Application Spin-Loop
// ... ...
UINT64 clocksBefore = GetClocks();
if ( S_OK != pDeviceContext->GetData( pQuery, &queryData, sizeof(UINT64),
0 ) ) {
        numClocksBetweenCalls = GetClocks() - clocksBefore;
        averageClocks += numClocksBetweenCalls;
        count++;

        if ( numClocksBetweenCalls < CLOCK_THRESHOLD )
        {
                averageClocks /=count;
                if ( averageClocks < AVERAGE_THRESHOLD )
                {
                        WaitOnDMAEvent( pQuery, &queryData, sizeof(UINT64) );
                        return queryData;
                }
                else
                {
                        return queryBusy;
                }
        }
        else
        {
                return queryBusy;
        }
}
else
{
        return queryData;
}
// End Detect Application Spin-Loop
```

Memory Transfer Reduction

Limiting data movement and efficient data processing lead to better performance and lower power consumption. In this connection, it is more efficient to keep data as close to processing elements as possible by using the memory and cache hierarchy, and to minimize data transfer from main memory.

Reduction of memory transfer can curtail the power consumption, owing to the reduced number of memory accesses, even at the expense of a moderate increase in computational complexity.[15] Bourge and Jung proposed to reduce memory transfer by using embedded compression for the predictive pictures in the encoding feedback loop. It is possible to use an embedded coding scheme that would keep the reference frame in the frame memory in compressed format so as to use about a third of the memory compared to regular uncompressed coding method. If a lossless compression is used, then the required memory would be halved instead.

However, by using block-based memory access and by carefully managing the computational complexity of the embedded coding scheme, Bourge and Jung show that an overall power saving is possible.[16] They achieve this by imposing some restrictions on the coding scheme, which is a lossy scheme and is capable of obtaining better compression ratio and corresponding power saving than a lossless scheme. The restrictions include coding each block independently, fixing the compression ratio for each block, and jointly storing the luminance and chrominance blocks in memory. The end result is that even with an increase in computational complexity, the memory transfer, which dominates power consumption, is saved by 55 percent.

Although this particular scheme resulted in visual quality degradation at higher bitrates, using an appropriate lossless scheme may bring about overall power savings due to less memory transfer. Most important, owing to such reduction in memory transfer, a smaller memory embedded closer to the CPU can be used, leading to less cable dissipation during access. In some hardware implementations, it is possible to use low-cost on-chip memory instead of off-chip SDRAM.

System Integration Optimization

The interaction between various layers in the software stack can be optimized during system integration to yield a more power-efficient solution. The operating system, the graphics drivers, the middleware such as Intel media software development kit (SDK), and the applications can cooperate in such optimization. As these layers are typically developed by different companies, it is natural to expect inefficiencies resulting from such interactions. To improve the inter-layer efficiency, the following approaches to system integration optimization may be considered:

- Reducing the number of layers.
- Improving the understanding of the authors of various layers regarding each other's capabilities and limitations.
- Redefining the boundaries of the layers.

However, lacking such radical approaches, and until these become available, system integration optimization can still be done at various levels, some of which are as follows.

[15]A. Bourge and J. Jung, "Low-Power H.264 Video Decoder with Graceful Degradation," *Proceedings of SPIE* 5308 (2004): 372–83.
[16]Ibid.

System Operating Point on the P-F Curve

Figure 6-7 shows typical system operating points on the power-frequency curve, compared to the minimum operating point (F_{max} at V_{min}) and the maximum operating point (running at turbo frequency).

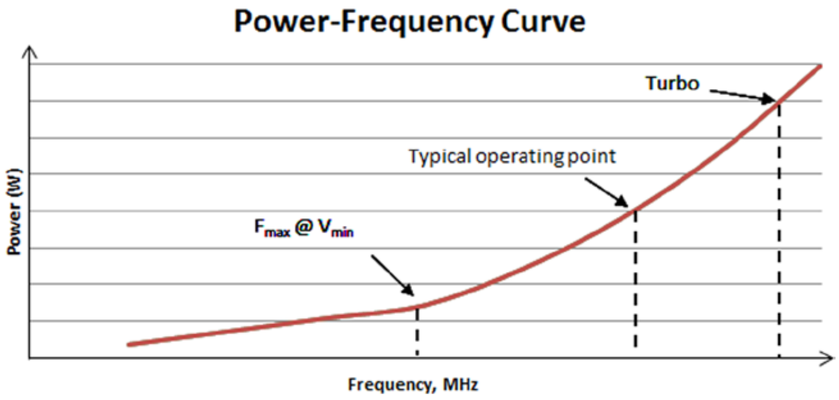


Figure 6-7. *Typical system operating point*

As seen in Figure 6-7, in the voltage-scaling region of the power curve, tuning the system's operating frequency is important for power saving. It is possible to occasionally run the system at a lower frequency and save power as long as performance requirements are met. From the power consumption point of view, the best operating point is F_{max} at V_{min}; however, this frequency may not be sufficient for some applications. On the other hand, from the performance point of view, the best operating point is in the turbo frequency region. Based on the resource utilization profile, it is possible for power-aware graphics drivers to determine how to tune the frequency of the processor, and it is possible to dynamically move between turbo and regular operating frequency.

As the operating system manages power, some systems offer various power policies ranging from low-power usage with low performance to high-power usage with high performance. In addition, the BIOS provide some flexibility to set the system frequency. End-users may take advantage of these power policies to adjust the system operating point to appropriate levels; for example, using the *power-saver* policy can lower the operating frequency and thereby save power.

Intelligent Scheduling

The levels of hardware-software partitioning are generally in the scope of architectural optimization. However, system-level optimization should also carefully consider the power-saving opportunities that are not covered by architectural design alone. For example, scheduling and migrating tasks between a software layer and special-purpose hardware units is a way such power-saving opportunities may be made available.

The operating system performs the scheduling of tasks for the CPU, while graphics drivers can schedule and manage the tasks for the GPU. Intelligent scheduling and load sharing between the CPU and the GPU is an active area of research, for which the middleware and the application layer may also make significant contributions. It is important, then, to find the most efficient place to do the processing; for instance, it may not be sufficient to simply multithread a CPU work, and it may be less efficient in terms of Joules per operation than operations per second.

Accomplishing migration of such a task from the CPU to a more power-efficient dedicated hardware module requires cooperation from all layers of the execution stack. To facilitate the scheduling, sometimes it is necessary to partition a piece of the system into several smaller chunks. For example, a shared user mode driver (UMD) that would interact with three run-time environments, such as OpenGL run-time, Direct3D 11 run-time, and Direct3D 9 run-time, may be redefined and divided into three components: OpenGL UMD, D3D 11 UMD, and D3D 9 UMD. This would facilitate both specific hardware access and interaction with the run-time environments; and it would make the system more amenable to power gating.

Similarly, some fixed work repeatedly done by the kernel mode driver for every invocation may be moved to the hardware itself. Examples of such system-level optimization can be found in the Intel fourth-generation core processor architecture, where using such system-level optimizations achieves a 2.25W decrease in CPU power for a popular 3D game application.[17]

Duty Cycle Reduction

By parallelizing the essential active tasks in a system—for example, tasks in the CPU, the GPU, the memory, and the I/O subsystem—the overall *uncore duty cycle* can be minimized. This would keep the related power subdomains active only for the required operations as needed and only for the minimum period of time, turning them off otherwise. The power subdomains include the various sensors, the PLLs, the memory interface interconnect buses, and so on, which can be separately controlled to minimize power consumption.

Furthermore, in order to run at a more efficient operating point, the duty cycle of the processor can be reduced by moving along the voltage-frequency curve, and using a higher frequency and higher power consumption for a shorter period of time, before going to an idle state for a relatively longer period of time. For the overall duration, this would typically result in lower power consumption. Conversely, for the same frequency, power can be saved with a lower voltage setting, as power is proportional to the square of the voltage. Duty cycle reduction is usually done at the system integration optimization level by the graphics kernel mode driver.

Figure 6-8 depicts the effect of a duty cycle reduction algorithm that focuses on using a higher frequency for a shorter period to accomplish the task of a video application, while the CPU is idle for longer period of time. In this example, the CPU utilization is reduced by approximately 20 percent.

[17]Blythe, "Technology Insight."

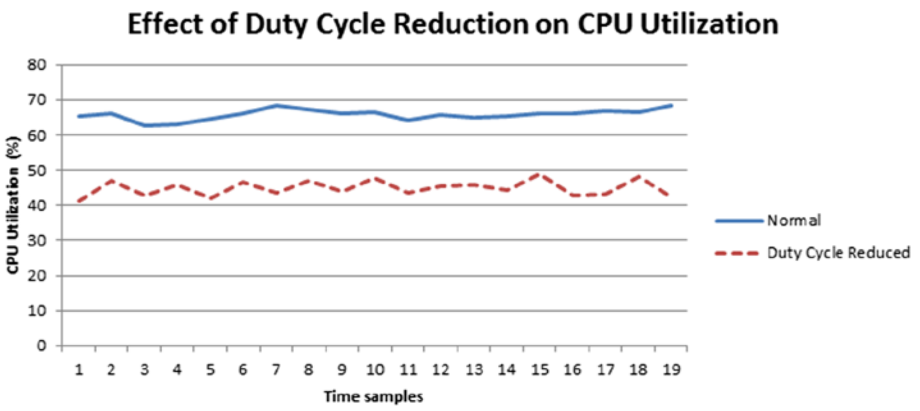


Figure 6-8. *Effect of duty cycle reduction on CPU utilization*

Application-Level Optimization

With the desire to support a plethora of functionalities in mobile computing devices comes the use of multiple sensors. A contemporary platform therefore includes light sensors, gyroscopes, accelerometers, GPS receivers, and near-field communications. By becoming aware of the available system resources and the user environment where multiple sensors may be active at a given time, applications can help avoid power misuse and can help users determine the priority of the sensors and features for a power-starving scenario.

Context Awareness by the Application

It is possible for a badly written application to burn power unnecessarily that could otherwise be saved. On the other hand, if an application is aware of the system resources that it runs on, and can sense a change in the system resource availability, it is possible for that application to react in a friendly manner to overall power consumption. For example, upon detecting low battery and subsequently notifying the user, an application may wait for intervention from the user before going to a lower power state. Alternatively, in a more active response, it may dim the display by default after sensing a darker ambient light condition.

It is the duty of the operating system to allocate system resources for each application, as requested by the application. The application's registering for power-related events allows the operating system to notify the application of a power event so as to enable the application to make an appropriate response. The application can also query for system state information using the APIs (application programming interfaces) provided by the operating system. For example, depending on whether the system is powered by a battery or connected to AC wall power, applications can make various power-saving decisions:

- Instead of a full system scan as done while on AC power, a virus checker may start a partial scan of the system on battery power.
- A media player may decide to trade off video quality to achieve longer playback of a Blu-ray movie.
- A gaming application may choose to sacrifice some special effects to accommodate more sections of the game.

In Windows, applications can query the operating system using a unique GUID (globally unique identifier) called `GUID_ACDC_POWER_SOURCE` to obtain the power setting information, and use this knowledge when a power event occurs. Similarly, to determine the remaining battery capacity, the `GUID_BATTERY_CAPACITY_REMAINING` can be used. And to learn about the current power policy, the `GUID_POWERSCHEME_PERSONALITY` can be used. It is also possible to use the `GUID_BACKGROUND_TASK_NOTIFICATION` to determine whether it is suitable to run a background task at the current state or it is better to wait for the *active* state so as not to perturb an idle state. In Linux, similar approaches also exist, where `CCBatteryInfo` structure can be used to determine the battery state. Furthermore, if the application switches contexts, it is possible to lower the power for the application's context that is no longer running.

Applications Seeking User Intervention

An application may invite favorable user intervention to save power. For example:

- An application can monitor battery capacity, and when the battery charge drops to a certain fraction of its capacity--say, 50 or 25 percent--the application may indicate a warning to the user interface to alert the user of the remaining battery capacity.
- An application can respond to a power source change from AC to DC by notifying the user of the change and providing an option to dim the display.
- An application can respond to ambient light level and request the user to adjust the display brightness.

Some of these actions can also be automatically taken by the system, but depending on the application, some may require user intervention. In general, user-configurable options allow the user to personalize the system, the application, and the experience. System and application designers may need to consider various tradeoffs when deciding which choices to give to the user and which to implement by default. For example, Windows provides the user with three power policies to choose from, or to define one's own settings. These options and settings drive the system-level behaviors that significantly impact the power efficiency of the platform.

Power Measurement

Now that we have covered different areas of power optimization, let us consider how to actually measure the power. In this section, we present the measurement methodology and various power-measurement considerations.

The ability to measure and account for power at various levels of the system allows system designers or users to understand existing power-management policies or to deploy optimized power-management policies as needed. Measuring power can uncover power-related problems that result in higher cost for the system. The major motivations for measuring power include:

- Understanding the impact of an application on power consumption by the system, and potentially finding optimization opportunities by tuning the application.
- Determining the effect of software changes at the user level, at the driver level, or at the kernel level; and understanding whether there is any performance or power regression owing to code changes.
- Verifying that a debug code was removed from the software.
- Determining the amount of power savings from power-management features, and verifying that such features are turned on.
- Determining the *performance per watt* in order to drive performance and power tuning, thereby obtaining the best tradeoff in practical thermally constrained environments.

However, few tools and instructions are available to measure the power consumed in a platform. Also, depending on the need for accuracy, different power-measurement methods can be used, ranging from simple and inexpensive devices to specialized data acquisition systems (DAQs). We present various approaches to power measurement.

Methodology

Within a computing system, power is measured at various system levels and at the motherboard level. In particular, this applies to the CPU package power, memory power, and display power measurement. Depending on the type of power supply, such measurement is of two types: AC power and DC power.

AC Power Measurement

For the system AC power or wall-power measurement, generally an AC power meter is connected between the power source and the system under measurement. The price for this measurement equipment may vary, depending on the accuracy and precision requirements. Simple, low-cost equipment typically has several drawbacks, including small ranges, low and imprecise sampling rates, inability to be used with other devices such as AC to DC converters or data acquisition systems, low resolution, and incongruity for measuring small power changes. On the other hand, they are easy to use and require little or no setup time.

For purposes of system-level and motherboard measurement, AC power measurement is not suitable, as these methods cannot provide insight into the system's power consumption.

DC Power Measurement

Although DC power can be measured using scopes and multi-meters, the easiest, most accurate, and most precise way of measuring DC power is by using automated *data acquisition systems* (DAQs). DAQs take analog signals as inputs and convert them to digital data sequence for further processing and analysis, using specialized software. Typically, DAQs can support several input channels, and can interface with the data-analyzing computer via standard serial or USB ports. They are capable of handling very high data rates and can measure tiny voltage differences, making them ideal for automated power measurements.

The power dissipated across a resistor can be expressed as follows:

$$P = V^2/R, \qquad \text{(Equation 6-3)}$$

where V is the voltage in volts, R is the resistance in ohms, and P is the power in watts.

The current through the circuit is determined by the ratio of V to R. To measure the power of a black box circuit, it is a common practice to add a very small *sense resistor* with a low resistance, r, in series with the black box, which has a larger resistance, R, so the total resistance of the circuit is approximately equal to R. In this case, the power needed for the black box can be approximated in a modified version of Equation 6-3:

$$\cdot P = \frac{v \times \Delta v}{R}, \qquad \text{(Equation 6-4)}$$

where ΔV is the voltage drop across the sense resistor and V is the potential of a channel input with respect to ground.

Since voltage is the potential difference between two points, for each voltage measurement two inputs are required: one to represent the ground, or reference potential, and the other to represent the non-zero voltage. In a single-ended measurement, the reference is provided by the DAQ's own ground and only the non-zero voltage is measured for an input channel voltage. Compared to using separate grounds for each channel, single-ended measurements may be less accurate, but they have the advantage of using faster sampling or more input channels.

DAQs can take as inputs the general-purpose analog signals in the form of voltage. The analog signals may have originally been captured using a sensor before being converted to the voltage form, or they may already exist in a voltage form. In the latter case, a simple low-resistance *sense resistor* can act as a sensor.

In order to measure the power of a certain system or motherboard component, typically the appropriate power rails are instrumented so that a sense resistor is connected in series on the rail. As current flows through the sense resistor, a voltage drop ΔV is created, which can be measured by the DAQ, as shown in Figure 6-9, where a very small sense resistor (e.g., 2 milliohm resistance) is used.

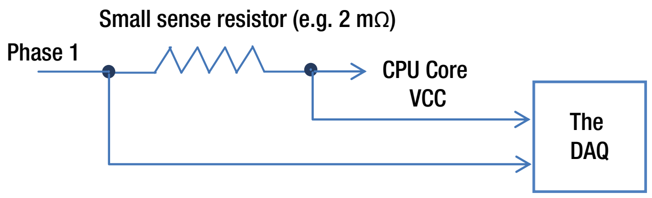


Figure 6-9. *Power measurement setup in a power rail*

The data can be analyzed and aggregated to give the measured power over a period of time, using special software accompanying the DAQ, such as the National Instrument LabView.

Considerations in Power Measurement

The following factors are generally taken into account while measuring power:

- The TDP of the processor part under measurement.
- The accuracy and precision of the data acquisition system; The ability of the DAQ and associated software for real-time conversion of analog voltage signals to digital data sequence, and for subsequent processing and analysis.
- Ambient temperature, heat dissipation, and cooling variations from one set of measurements to another; to hedge against run-to-run variation from environmental factors, a three-run set of measurements is usually taken and the median measured value is considered.
- Separate annotation of appropriate power rails for associated power savings, while recording the power consumption on all power rails at a typical sampling rate of 1 kHz (i.e., one sample every one millisecond), with a thermally relevant measurement window between one and five seconds as the moving average.
- Recognition of operating system background tasks and power policy; for example, when no media workload is running and the processor is apparently idle, the CPU may still be busy running background tasks; in addition, the power-saving policy of the operating system may have adjusted the high-frequency limit of the CPU, which needs to be carefully considered.
- Consideration of average power over a period of time in order to eliminate the sudden spikes in power transients, and consideration only of steady-state power consumption behavior.

- Benchmarks included for both synthetic settings and common usage scenarios; appropriate workloads considered for high-end usages so that various parts of the system get a chance to reach their potential limits.
- Consideration of using the latest available graphics driver and media SDK versions, as there may be power optimizations available in driver and middleware level; also, there is a risk of power or performance regression with a new graphics driver such as potential changes to the GPU core, memory, PLL, voltage regulator settings, and over-clock (turbo) settings.

Tools and Applications

Power-measurement tools include both specialized and accurate measurement systems such as DAQs, as well as less accurate software-based tools and applications. We consider a specialized DAQ system and introduce several software tools with varying capabilities.

An Example DC Power-Measurement System

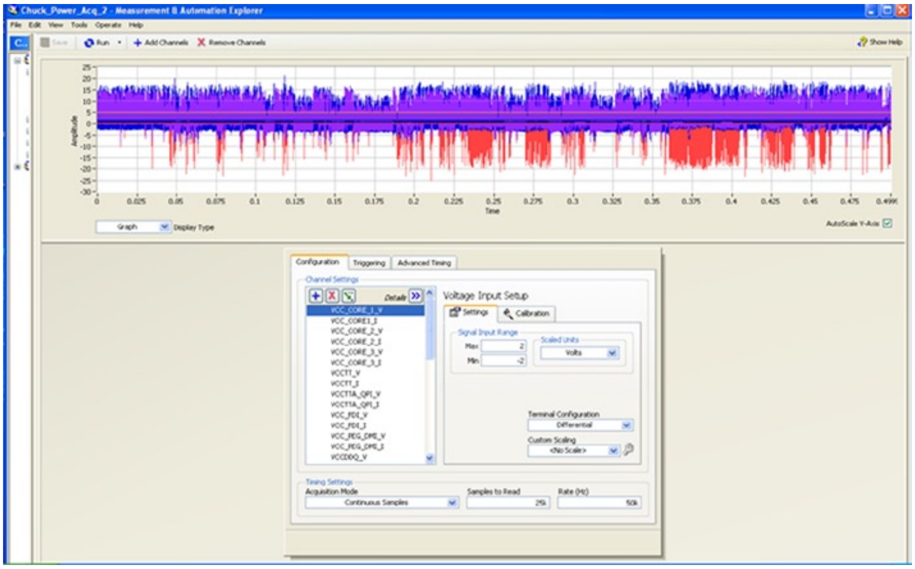
An example DC power measurement system is based on the National Instruments* PXIe 6363 PCI-express based DAQ and the associated LabView Signal Express software application for signal analysis. The PXIe 6363 has a signal capture bandwidth of 1.25 million samples per second and an A/D conversion resolution of 16 bits on every voltage input channel. This input voltage is programmable down to ±1V, so that it is easy to zoom into the low-voltage signals. Similarly, for today's low-power devices, newer versions of PCIe DAQs with higher-precision input voltages are also available.

Typically a 2 milli-ohm current sense resistor is used in series with all power rails of interest—for example, the CPU package, the memory DIMMs, and the display, for which the peak, the average, and the minimum DC power consumption are measured. Also, the run-time CPU and GPU frequencies are monitored to determine proper turbo operation. The power setup is calibrated automatically on each run for sense resistor and test harness variations that may occur due to ambient temperature.

To capture and compute power in watts, it is necessary to measure both voltage and current for each power rail. This is accomplished by using current sense resistors in series with the incoming power supply on each voltage rail. The voltage drop across the current sense resistor is a small amplitude signal that directly correlates to the amount of current flowing through the sense resistor. The voltage for each power rail (positive and negative wire), and the output of the current sense resistor (positive and negative wire), connects directly to the PXIe 6363 via the removable TB-2706 terminal block analog input modules.

The measured power is logged and plotted using the LabView Signal Express to produce a detailed and comprehensive power-performance profile. This application captures and processes the voltage and current measurements from the PXIe 6363 DAQ modules and computes the power in watts simply by multiplying the measured voltage and sense current.

This application also supports various statistical measurements, such as moving average, peak, average, and minimum power used for detailed signal analysis. Figure 6-10 depicts a sample of a LabView configuration interface for a power measurement system. In this interface, selections can be made for the voltage channels of interest. Figure 6-11 then shows an example of the LabView interface when a power measurement is in progress. The top and bottom windows show voltage, current, or power signals from all input channels and a single channel (Channel 0), respectively.



Channel Settings

VCC_CORE_1_V
VCC_CORE1_I
VCC_CORE_2_V
VCC_CORE_2_I
VCC_CORE_3_V
VCC_CORE_3_I
VCCTT_V
VCCTT_I
VCCTTA_QPI_V
VCCTTA_QPI_I
VCC_FDI_V
VCC_FDI_I
VCC_PEG_DMI_V
VCC_PEG_DMI_I
VCCDDQ_V

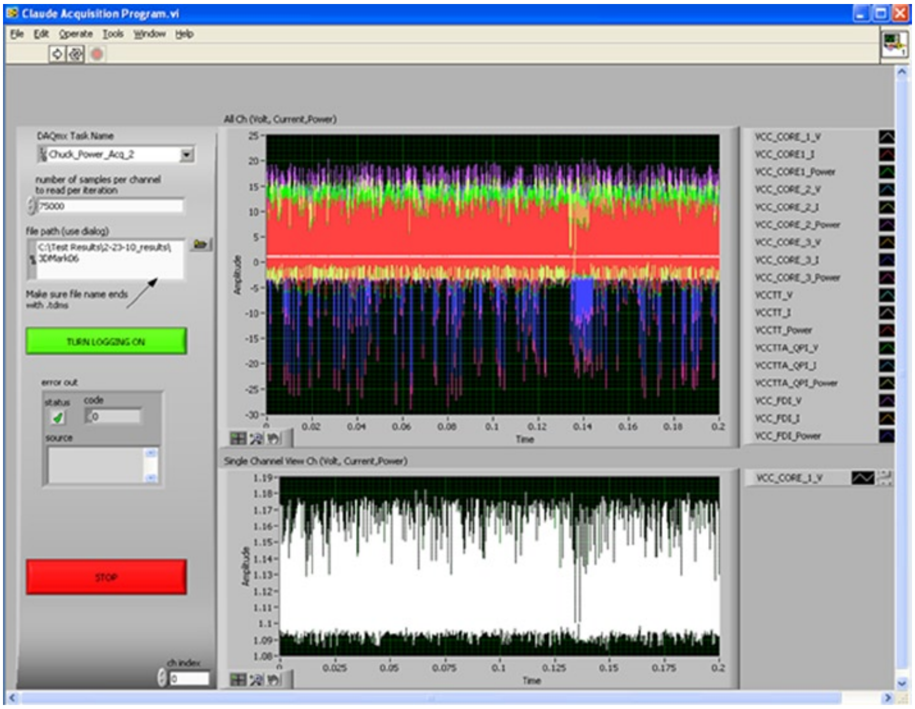
Voltage Input Setup

Signal Input Range: Max 2, Min -2
Scaled Units: Volts
Terminal Configuration: Differential
Custom Scaling: <No Scale>

Timing Settings

Acquisition Mode: Continuous Samples
Samples to Read: 25k
Rate (Hz): 50k

Figure 6-10. *LabView data acquisition setup for power measurement*



DAQmx Task Name: Chuck_Power_Acq_2

number of samples per channel to read per iteration: 75000

file path (use dialog): C:\Test Results\2-23-10_results\3DMark06

Make sure file name ends with .tdms

error out — status: ✓ code: 0 source:

ch index: 0

Figure 6-11. *Power data acquisition in progress*

Software Tools and Applications

To get the best and most accurate data on how much energy a computer platform is using during operation, a hardware power meter is needed. The Networked Data Acquisition Unit (NetDAQ) from Fluke, the National Instrument DAQ, and the Yokogawa WT210 are examples of such acquisition systems. However, these are expensive and the cost may not be justifiable to a regular consumer or an application developer who is only interested in a one-time or so power measurement. For these users it makes more sense to select a software tool or application that measures power consumption.

The tool and applications are primarily used to identify power issues, with and without workloads running, in order to optimize the system's power consumption. The issues typically encountered include:

- **CPU/Chipset Power:** Such problems are identified by examining the CPU *C*-state residency to determine whether the CPU and the chipset power are optimally managed, and to get some insight into what is causing any increase in platform power consumption. For example, high residency at deep *C*-states such as *C3* may indicate frequent *C*-state transition due to device interrupt or software activity.

- **CPU Utilization:** CPU utilization samples are commonly taken at every timer tick interrupt--i.e., every 15.6 millisecond for most media applications and some background applications. However, the timer resolution can be shortened from the default 15.6 millisecond in an attempt to capture activities within shorter periods. For multi-core CPUs, CPU utilization and power consumption depend on the active duration, while each core may only be active for a partial segment of the total duration for which the platform is active. Therefore, CPU core utilization and platform utilization should be counted separately. Logically, when the activities of two cores overlap, the CPU utilization is shown as the sum of two utilizations by most power measurement tools. Only few tools, the Intel Battery Life Analyzer among them, can actually use fine-grain process information to determine the total active duration of both the platform package and the logical CPU. By investigating the CPU utilization, inefficient software components and their hotspots can be identified, and the impact of the software component and its hotspots can be determined to find optimization opportunities.
- **CPU Activity Frequency:** Power tools can help identify software components causing frequent transition of CPU states. It is valuable to determine the frequency of the activity of each component and the number of activities that are happening in each tick period. Understanding why the frequent transitions are happening may help point to power-related issues or improvement prospects.
- **GPU Power:** On the modern processors, as most media applications run on the GPU, it is also important to understand the impact of GPU *C*-state transitions and GPU utilization. GPU utilization largely controls the power consumption of media applications. However, there are only few tools that have the ability to report GPU utilization; the Intel GPA is one such tool.

In general, there are several tools and applications available for the measurement and analysis of power consumption of various components of a computing device. Some are especially relevant for analysis of the idle system behavior, while others are suitable for media applications. In the next section, we discuss some of these tools, starting with the Linux/Android based PowerTop and going into several Windows-based tools. We then discuss specific tools for monitoring and analyzing battery life. The Power Virus is also mentioned, which is mainly used for thermal testing. However, as new tools are constantly being developed, some tools obviously are not covered.

PowerTop

PowerTop is a software utility developed by Intel and released under GPL license that is designed to measure and analyze power consumption by applications, device drivers, and kernels running on Android, Linux, or Solaris operating systems. It is helpful in identifying programs that have power issues and to pinpoint software that results in excessive power use. This is particularly useful for mobile devices as a way to prolong the battery life.

PowerCfg

PowerCfg is a command line tool in Windows that allows users control the power-management settings of the system and to view or modify the power policy. It is typically used to detect common issues in power efficiency, processor utilization, timer resolution, USB device selective suspend, power requests, and battery capacity.

PwrTest

PwrTest is a power management test tool available in the Windows Driver Kit that enables application developers and system integrators to obtain power-management information such as the various sleep states information (e.g., *C*-state and *P*-state information) and battery information from the system and record over a period of time.

Perfmon and Xperf

The Windows Perfmon provides abilities to monitor the performance counters available in Windows, including *C*-state and *P*-state residencies, which are useful in understanding CPU utilization and activity related issues.

The `Xperf` is a command-line tool that helps developers in system-wide performance analysis by monitoring system and kernel events such as context switches, interrupt service routines, and deferred procedure calls for a period of time and by generating reports for graphical review. It is useful to correlate the events with system status in scenarios where the system is idle, running web browsing, or during media applications. `Xperf` generates event trace logs that can be viewed using `Xperfview`; both of these tools are available in the Windows Performance Toolkit.

Joulemeter

Developed by Microsoft Research, Joulemeter is a modeling tool to measure the energy usage of virtual machines (VMs), computers of various form factors and power capacity, and even individual software applications running on a computer. It measures the impact of components such as the CPU, screen, memory, and storage on their total power use. One of its advantages is that it can measure the impact of software components, such as VMs, that do not have a hardware interface and therefore are not amenable to measurement by hardware power meters.

The data obtainable from Joulemeter includes the current energy usage for each component, such as the base or idle energy usage, CPU usage above the baseline idle, monitor, and hard disk. The output data is presented in watts and is updated every second. Details can be found in *Joulemeter: Computational Energy Measurement and Optimization.*[18]

Intel Power Gadget

To assist end-users, independent software vendors, original equipment manufacturers, and the application developers to precisely estimate power consumption without any hardware instrumentation of the system, Intel developed a software tool named Intel Power Gadget, which is enabled for the second- Generation Intel Core processors. Additional functions of the tool include estimation of power on multi-socket systems and externally callable APIs to extract power information within sections of the application code.

The gadget includes a Microsoft Windows sidebar gadget, driver, and libraries to monitor and estimate real-time processor package power information in watts, using the energy counters in the processor. After installation, the gadget can be simply brought up to monitor processor power usage while running a workload or when the system is idle. An "Options" pop-up window allows setting the sampling resolution in milliseconds and the maximum power in watts. The output data, notably the processor package power and frequency, is generated in real time and can be logged in a file with a comma-separated values (CSV) format. The gadget can be downloaded from Intel's website.[19]

Intel Power Checker

The Intel power or energy checker tool determines the power efficiency of a system in terms of useful work done with respect to energy consumed during that work. It is an easy way for media or game application developers to check the power efficiency of their applications on mobile platforms with Intel Core or Atom processors. This tool does not require an external power meter, and it is useful for power analysis of any application compiled for Intel processors or Java framework applications.

By default, this tool checks the system capability to provide power consumption data and whether a particular driver called `EzPwr.sys` (part of Intel Power Gadget) is installed, which would be necessary if an external power meter device is used. Typically, the tool first measures the baseline power without the target application running, while unnecessary processes such as operating system updates, Windows indexing service, virus scans, Internet browsers, and so on are turned off. In the next step, the target application is run, and power is measured again starting from a desired point of the target application's execution. Finally, it measures power again when the target application is completed and returned to an idle state. The tool provides analysis on elapsed time, energy consumption, and average *C3* state residency, and gives the platform timer duration in milliseconds. This tool is now part of the Intel Software Development Assistant.[20]

[18]Available from Microsoft Research at `research.microsoft.com/en-us/projects/joulemeter/default.aspx`.

[19]Available from `software.intel.com/en-us/articles/intel-power-gadget`.

[20]Available from `software.intel.com/en-us/isda`.

Intel Battery Life Analyzer

The Intel Battery Life Analyzer (BLA) is a software tool running on Microsoft Windows that is primarily used to monitor the activities of hardware and software platform components and determine their impact on battery life. It can identify drivers, processes, or hardware components that prevent the platform from entering low-power states. BLA has many modules to support the power analysis, including CPU *C*-state and software activity analysis.

The more time the system spends in the deep *C*-state, the less power it consumes. BLA recommends threshold values for *C*-state residencies, in particular, that the deepest *C*-state residency at idle should be greater than 95 percent for the processor package (i.e., socket) containing multiple processor cores and 98 percent per core. Also, *C0* and *C1* states for the package should be less than 5 percent at idle. There are options in the BLA tool to set the appropriate *C*-state threshold values. Copies of the BLA tool can be requested via e-mail from Intel.[21]

Intel Graphics Performance Analyzer

The Intel Graphics Performance Analyzers 2013 (Intel GPA) is a suite of three graphics analysis and optimization tools--namely, the system analyzer, the frame analyzer, and the platform analyzer—to help game and media application developers optimize their games and other graphics-intensive applications. Intel GPA supports the latest generations of Intel Core and Intel Atom processor-based platforms running Microsoft Windows 7, 8, 8.1, or the Android operating system. The system analyzer provides the CPU and the GPU performance and power metrics in real time, and allows users to quickly identify whether the workload is CPU- or GPU-bound so the user can concentrate on specific optimization efforts. The frame analyzer provides ability to analyze performance and power down to the frame level. The platform analyzer provides off-line analysis of the CPU and GPU metrics and workloads with a timeline view of tasks, threads, Microsoft DirectX, and GPU-accelerated media applications in context. The tool is also available from Intel.[22]

GPU-Z and HWiNFO

GPU-Z is a lightweight system utility from TechPowerUp, designed to provide vital information about a video card and/or the integrated graphics processor; it supports nVIDIA, ATI, and Intel graphics devices. HWiNFO is free software, available from the Internet, that combines the functionalities of CPU-Z and GPU-Z and provides the CPU, the GPU, and memory usages, along with other system information.

[21]Request for BLA tool can be made at `batterylifeanalyzer@intel.com`.
[22]Available from `software.intel.com/en-us/vcsource/tools/intel-gpa`.

Power Virus

Power virus executes specific machine code in order to reach the maximum CPU power dissipation limit—that is, the maximum thermal energy output for the CPU. This application is often used to perform integration testing and thermal testing of computer components during the design phase of a product, or for product benchmarking using synthetic benchmarks.

Summary

In modern processors, power considerations go beyond battery life and attempt to dictate performance. We reviewed the power-consumption behavior by media applications running on mainstream computing devices.

First, we discussed the requirements and limits of power consumption of typical systems, the power equation, and aspects of various sources of power supply. Then, we covered how a mobile device is expected to serve as the platform for computing, communication, productivity, navigation, entertainment, and education. We also surveyed three major topics: power management, power optimizations, and power measurement considerations.

Finally, we learned about several power-measurement tools and applications, and their advantages and limitations. In particular, we showed as an example a specific DC power measurement system using a DAQ, and several software-based power measurement tools.

While there is no single tool or application suitable for all types of power-measurement scenarios, some tools are quite capable of providing important insights into the power profiles of video applications, and are useful for this purpose.