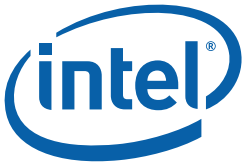


Energy-Efficient Platforms – Considerations for Application Software and Services

Whitepaper

February 2011

Revision 2.0

Document Number: 322304-002

Contents

Figures

Revision History

Document Number	Revision Number	Description	Revision Date
322304-001	1.0	• Initial Release	July 2009
322304-002	2.0	• Mobile Platform Update	February 2011

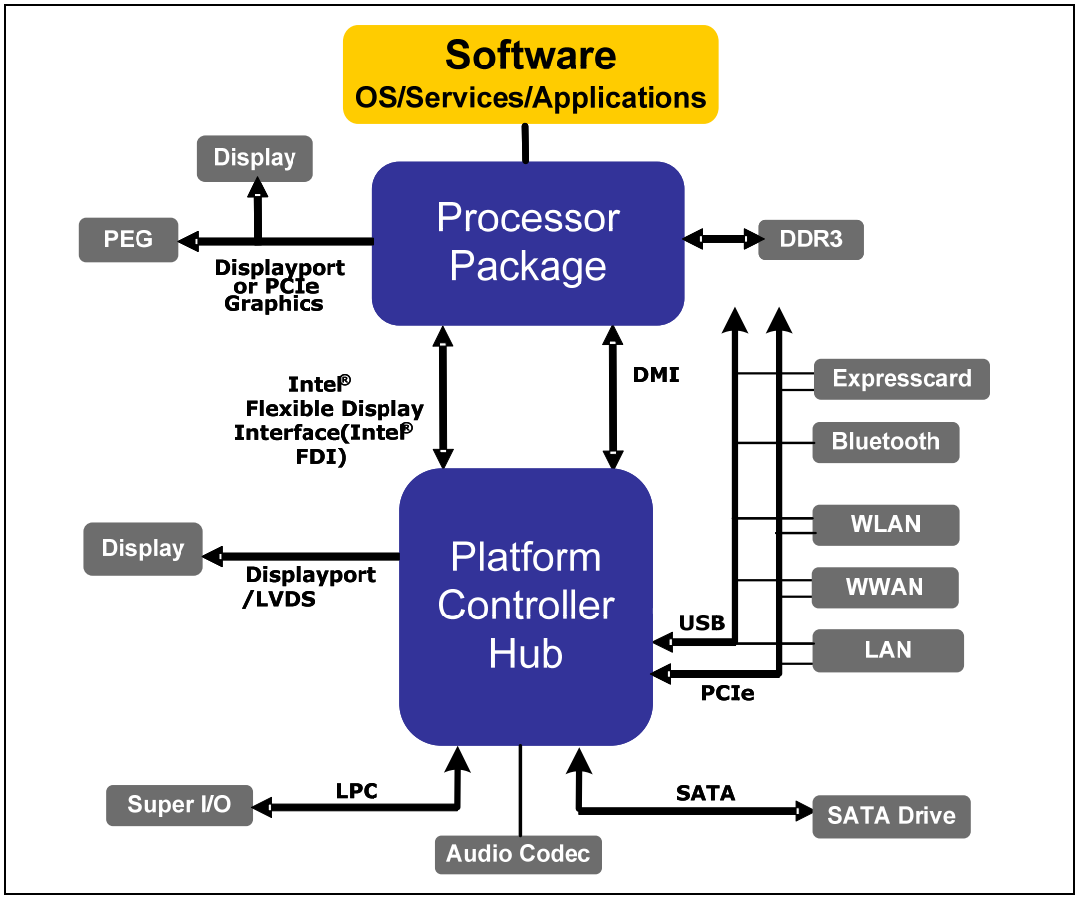
1 Introduction

1.1 Overview

Adoption of Mobile platforms of all form factors – Notebooks, Netbooks, Mobile Internet Devices, etc. have been steadily increasing and longer battery life is consistently ranked as one of the top requirements by consumers. Continuous network connectivity enhances the mobile usage model, but increases power consumption, thereby requiring improved energy efficiency. Mobile platforms must also meet the energy efficiency regulatory requirements such as the US Environmental Protection Agency (EPA) Energy Star program. To make the vision of "All day Battery Life" a reality, the average platform power consumption must go down.

Intel® Architecture (IA) platforms such as depicted in Figure 1 are open systems where operating systems, software applications and services and hardware devices are created and sold by various vendors.

Figure 1: Platform Ecosystem[1]



In spite of remarkable progress in processor power management and efforts to address the power efficiency of other platform components, a single ill-behaving device or software ingredient can impede all these benefits by preventing the platform components from residing in low power states. Platform level energy efficiency requires all components in the platform ecosystem to cooperate. Additionally, software plays an important role in the platform power ecosystem and the way software applications and services behave have a huge impact on battery life.

Both innovative performance and power management features are being added to successive generations of Intel® Architecture (IA) platforms to provide performance

[1] PEG: PCI Express* Graphics
DMI: Direct Media Interface
LPC: Low Pin Count bus
WLAN: Wireless Local Area Network
WWAN: Wireless Wide Network
LAN: Local Area Network
SATA: Serial ATA

on demand and save power at other times. Software needs to be written in a manner such that the underlying hardware consumes the lowest power for a task and higher power performance features are only turned on when the required performance cannot be achieved at lower power.

1.2 Structure of the Whitepaper

The first two chapters (Chapter 0 and Chapter 2) outline the fundamentals of the mobile platform power saving features in including how software behavior affects platform energy-efficiency. Then Chapter 3 describes how to identify software issues that negatively affects platform energy-efficiency. Finally, Chapter 4 discusses the details of how to fix and optimize software energy-efficiency.

1.3 Mobile Platform Power

What is Mobile Platform Power? Power is typically the amount of energy consumed by the platform over time. This involves understanding the workload of the platform and the energy used over a given time for that workload.

Mobile Platform Power is typically broken into three categories:

- Thermal Design Power (TDP)
- Platform Average Power – Average platform power measured over some time when a workload is executing
- Platform Idle Power – Average platform power measured over some time when no workload is executing and the system is idle

The TDP power is defined as being the hardest workload the mobile platform should ever see under normal operating conditions, and is what the mobile platform thermal cooling system is designed to handle. TDP largely defines the power level that the system should be designed to cool, and is in general, not directly related to normal platform battery life.

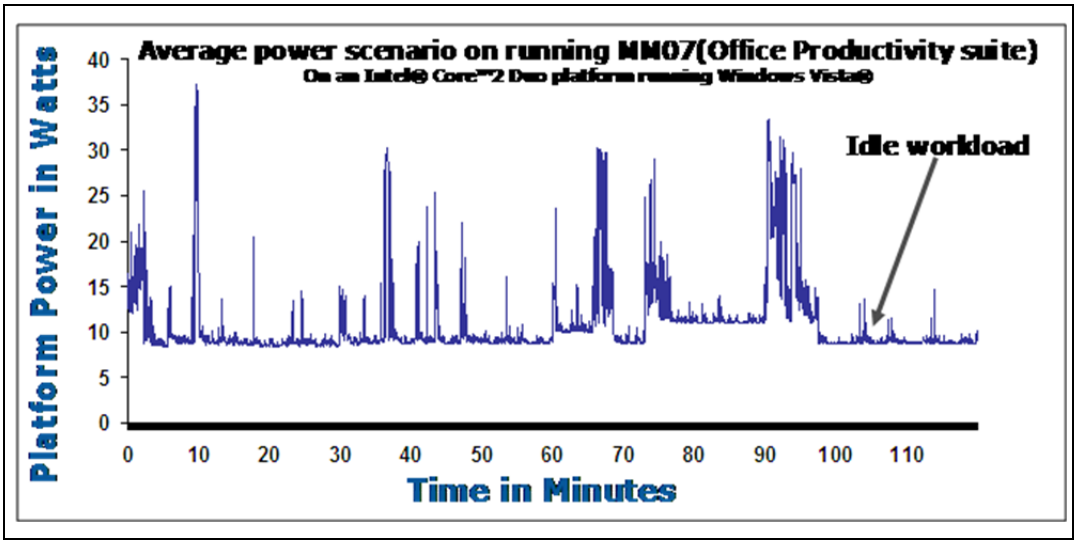
Average power for mobile platforms is defined as the average power consumption of the platform and is modeled by benchmarks such as Mobile Mark ’07 (MM07) that are representative of real end user usage patterns where the machine is idle between bursts of activity.

Idle power for mobile platforms is defined as being the power a platform would consume when the system is running in the ACPI S0 state, and software applications and services may be running but are not actively executing workloads and there is minimal background activity.

1.4 Typical Power Profile

Usage analysis has shown that a mobile platform in the ACPI S0 working state is typically idle for about 90-95% of the time as measured by the CPU C-state residency. Figure 2 below shows an example power graph over a period of time with a typical benchmark running.

Figure 2: Typical Mobile Platform Power Profile in ACPI S0 State



Since a mobile platform predominantly resides in the idle state, it is crucial to lower the platform idle power consumption for a significant increase in battery life. This also benefits the average power scenarios, and helps all but the most demanding (TDP-like) workloads. When actively executing workloads, improving computational and data efficiency so that the job can get done quickly, thereby increasing idle state residency improves platform energy-efficiency.

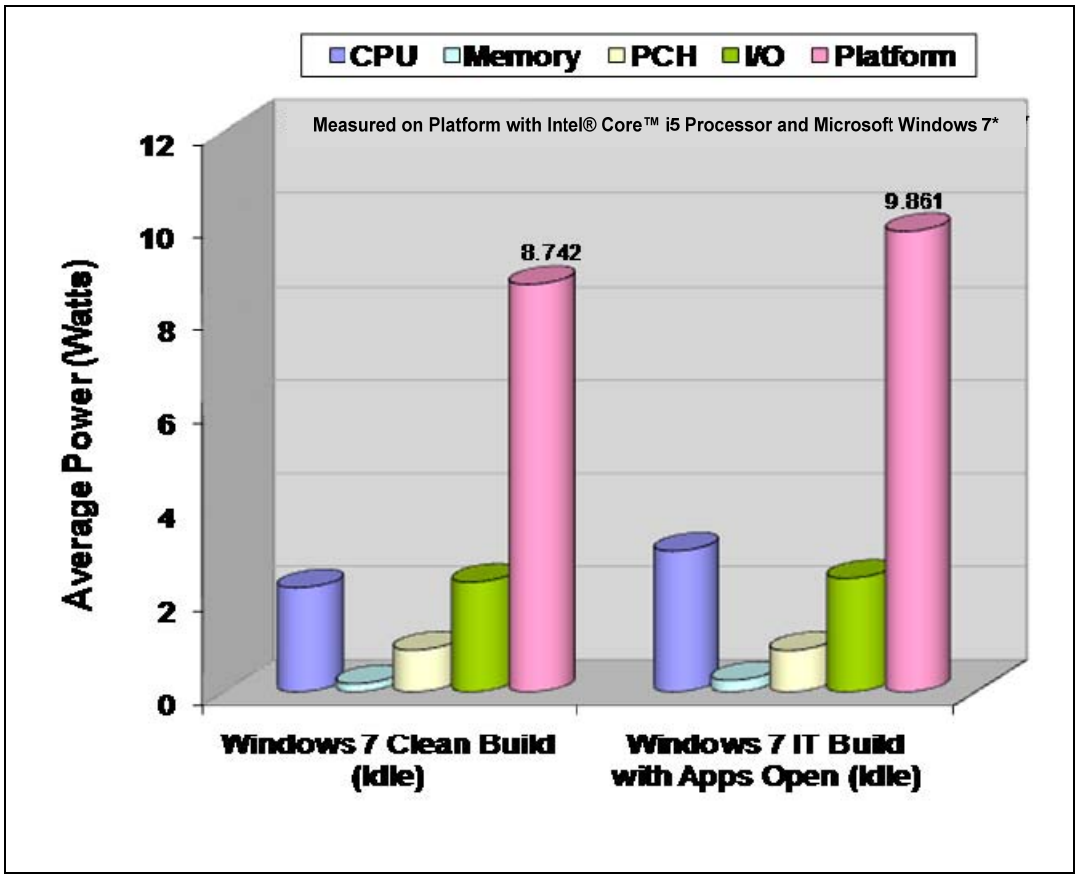
1.5 Software Impact on Platform Power

Software behavior can have a significant effect on platform power consumption and battery life. When idle, power efficient applications should have minimal effect on platform power consumption. Ideally, the platform idle power consumption when the application is started, but doing nothing meaningful should be the same as when the application is not running.

One way to evaluate the impact of application software and services on platform power is to compare the average power consumption of a platform with a clean build (just the OS and driver installation) and a platform with additional software and services installed (e.g., An IT configured enterprise notebook) and running but idle.

Figure 3 below shows an example of such a comparison. There is an impact of ~1 W even when the applications are idle. This type of application behavior can have a significant impact on battery life.

Figure 3: Comparison of Clean Build (Idle) Vs. IT Build with Apps Open (Idle)



1.6 Energy-Efficient Software

Software plays an important role in the platform energy-efficiency paradigm. Platform hardware components lay the foundation for power consumption and in every successive generation of platform, the hardware components are being designed to aggressively reduce power consumption when idle and provide performance on demand. However, a single ill-behaving software application or service can eliminate all the benefits designed into the hardware and can have an adverse impact on platform power.

In newer and future platforms with deeper power management states, there is significant power savings in these states, but the power cost of entering and exiting these states is also high. Hence, any software or service that causes very frequent

entry/exit to these states will mitigate all the savings and may actually increase power consumption.

Below are some of the high level requirements for energy-efficient software:

Idle workloads

- For idle workloads where the processor and platform can reside in the deepest idle state for long periods of time, software execution (for status polling, statistics, background bookkeeping activities, etc.) should be avoided as much as possible.
- Activity must be aligned, coalesced or batched whenever possible. Frequency of execution, especially for background services should be minimized as much as possible
- Software must ensure that it does not hinder the platform hardware components from going into the deepest low power states by generating unnecessary interrupts. Processor and Operating system policies for setting P-state and C-state are influenced by application behavior and keeping the CPU in certain P-states and C-states will not allow the platform to go into deep low power states
- Hard disks spin down to low power states when idle. Periodic registry or disk accesses by software must be avoided for idle workloads.

Active Workload Execution

- Improve computational efficiency and CPU utilization to enable longer idle periods where the hardware components can be power managed
- Reduce the number of processor C-state break events. Frequent C-state transitions will result in net energy loss
- In multi-core CPU architectures, maximize multi-core concurrent execution. This will enable processor package level resources and platform resources to be more efficiently power managed
- Reduce frequency of execution. A job with a certain CPU utilization requirement, is more energy-efficient when executed less frequently with longer periods of execution
- Applications increasing timer resolution have a huge impact on power. Use the largest timer interval possible. If smaller intervals are required, then reset the interval to a large value as soon as the task is done

2 Software Impact to Platform Energy-Efficiency

2.1 C-State Transitions

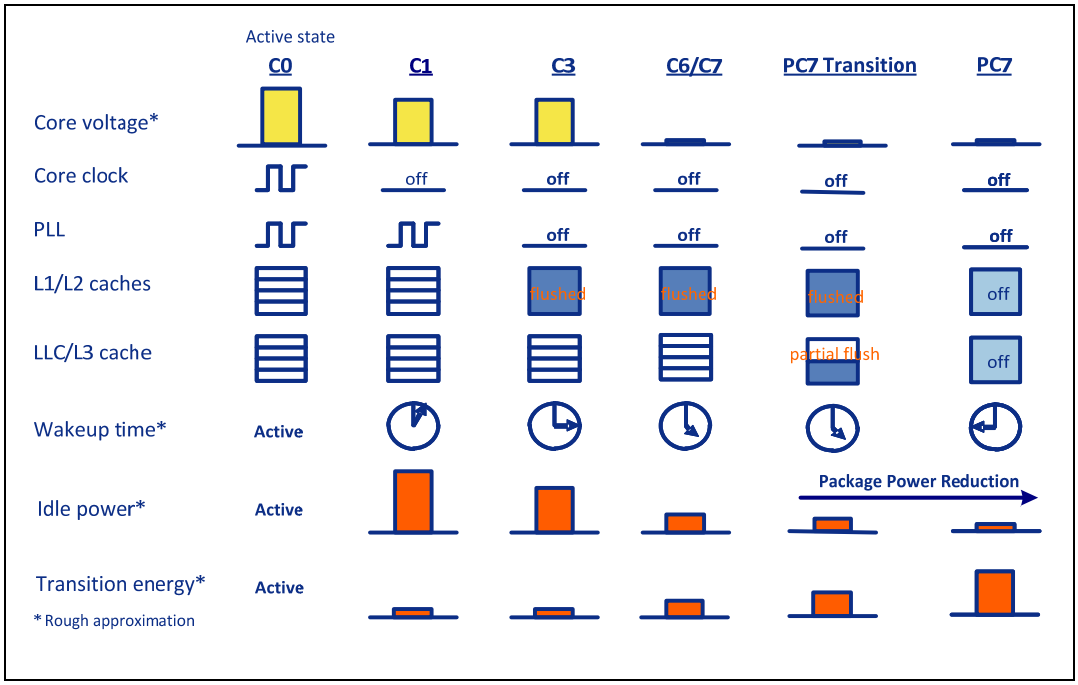
Software must be written with the objective of keeping the number of processor C-state transitions as low as possible. Very frequent C-state transitions will result in net energy loss.

2.1.1 Processor Power Management Overview

In order to save power when the processor is idle, the processor enters low power states called C-states. Intel processors support several levels of core and package (resources shared by all the cores) idle states (C-states), allowing for a flexible selection between power consumption and responsiveness.

Figure 4 below shows the typical core and package C-states supported by Intel processors. Cx states denote processor core C-states whereas PCx states denote processor package C-states. Although software applications do not directly set these C-states, their behavior greatly influences the processor residency in various C-states.

Figure 4: Flexible C-states to select Idle Power Level vs. Responsiveness



Typical Core level C-states supported are:

C0 – Active state executing code

C1 – Halted, snoops serviced

C3 - Core (L1/L2) caches flushed

C6 – Core state saved and Core voltage reduced to ~0

C7 – When last core enters C7, LLC is flushed progressively

Typical Package level states supported are:

PC0 – Active state

PC1 – Low latency state

PC3/PC6 – LLC ways valid, retention voltage

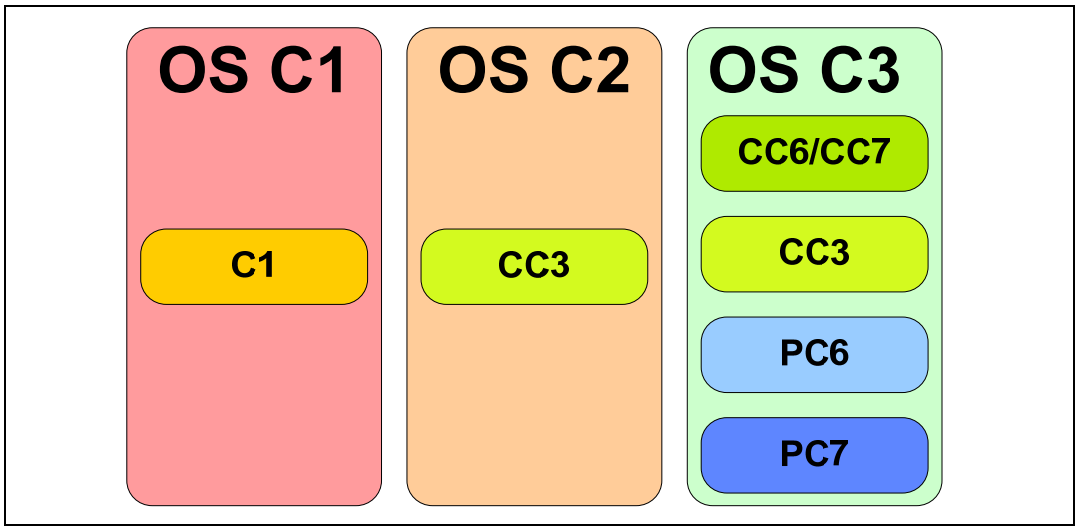
PC7 (Deep Power down) – LLC fully shrunk, No snoops, aggressive package power reduction

C2 Popup – For Bus Master Traffic

2.1.2 Software C-state to Hardware C-state Mapping

Operating Systems use the ACPI C-states C1, C2 and C3 and these are mapped to various hardware C-states as shown in Figure 5 below. The Operating System sets C-state per logical processor (processor thread). Underlying hardware coordinates between the several logical processors and also determines the package C-states.

Figure 5: OS C-state Vs Actual Hardware C-state



2.1.3 The Energy Cost of Transition Increases as Deeper C-states are Entered. Frequent Transitions to Deep C-states Will Result in a Net Energy Loss. Application Software Impact to C-state Transition

Although C-state transitions are controlled by the operating system and application programs don't have any direct control over it, the behavior of application programs can have a big influence on how effectively C-states are used.

Application programs primarily impact the frequency of C-state transitions. When a running thread terminates or blocks and no other thread is scheduled for the particular logical CPU, that CPU enters a lower power C-state and stays in that state until a ready thread is scheduled to execute on that CPU. There are many reasons why a running thread blocks; a common reason being waiting for external events (I/O completion, signal from other thread, etc.). Reducing the frequency of waiting for these events will help reduce C-state transitions.

It is important to be aware that some OS APIs use RPCs (Remote Procedure Calls) which block the thread and causes C-State transitions.

2.1.4 Impact of Excessive C-state Transitions: A Case Study

Software must aim to keep the number of C-state transitions as low as possible. Frequent C-state transitions from idle to active are not energy efficient. Activity must be coalesced or batched whenever possible to allow for higher C-state residencies and reduced C-state transitions.

Figure 6: Frequent Inter-Processor Interrupt and C-state Transitions

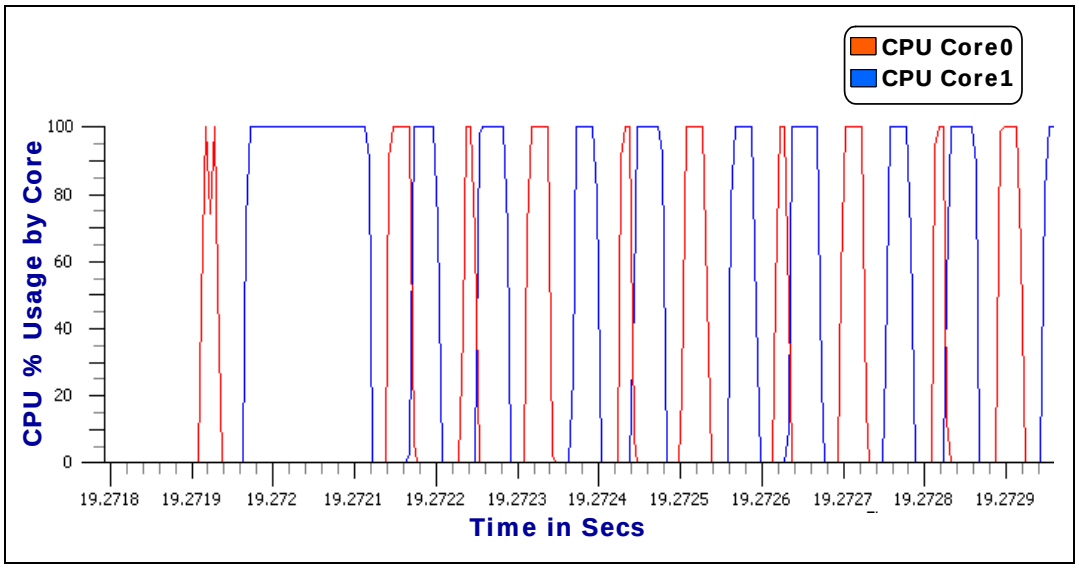
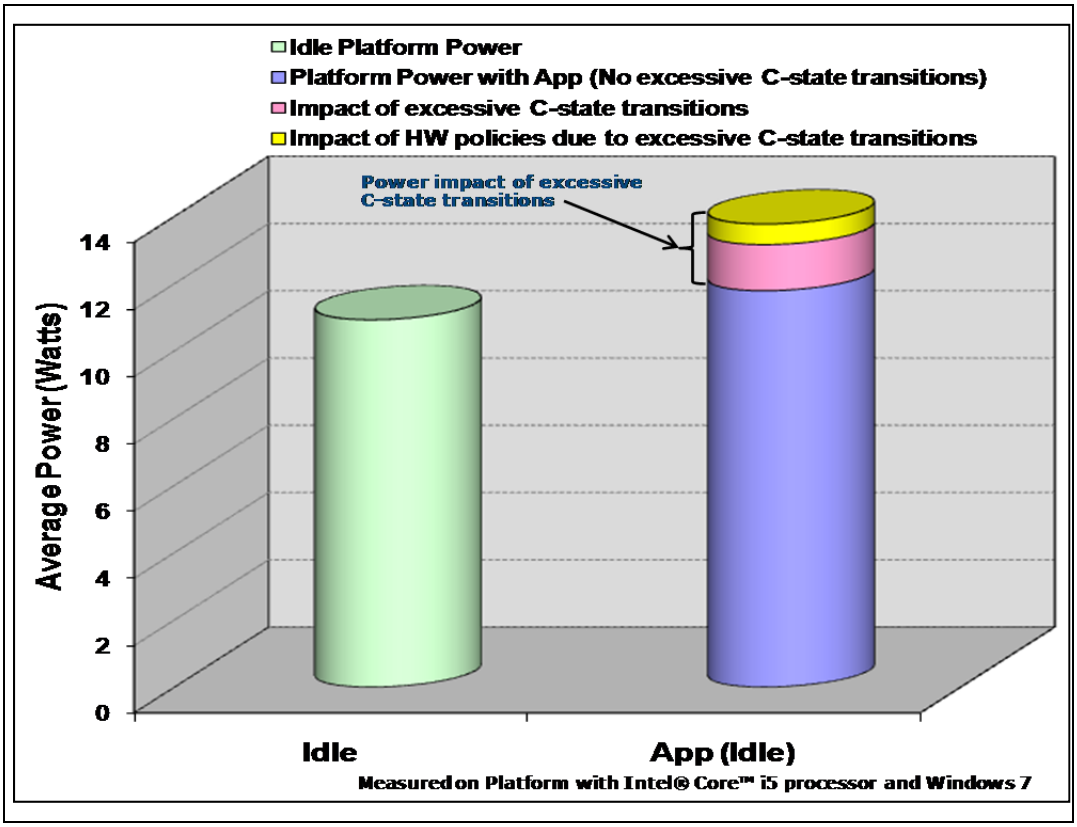


Figure 6 above shows a single-threaded application that has two processes which communicate frequently with each other while one process waits on the other process completion. Each process runs for a very small duration (~50usec). In a multi-core system, when these two processes are scheduled on two different cores, the communication between these two processes generates an Inter-Processor Interrupt (IPI). While one process is waiting for the other to complete, the core goes into a lower power C-state.

Figure 7 below shows the power impact of such behavior.

Figure 7: Power Impact of Excessive C-state Transitions



This sort of frequent C-state transition impacts power consumption in two ways:

- Energy requirements to enter/exit C-state are not trivial. When the C0 (active) duration is very small, the latency to transition in and out of the C-states in comparison is appreciable and may result in net energy loss.
- Hardware policy may demote a C-state request from OS to a shallower C-state based on heuristics. Even if the frequent C-state transition behavior occurs only for 2-3 msec in a 15.6 msec window, hardware polices may either demote core C-state or re-open package level cache and this will impact power for the remaining ~12-13 msec of idle period.

2.1.5 Software Recommendations

High level recommendations for applications and services to reduce C-state transitions are:

- Do not split a task between processes/threads unless parallel execution can occur
- If it is necessary that a task be split between processes, then coalesce work so that the number of C-state transitions can be reduced
- Coalesce activity whenever possible to increase idle period residency

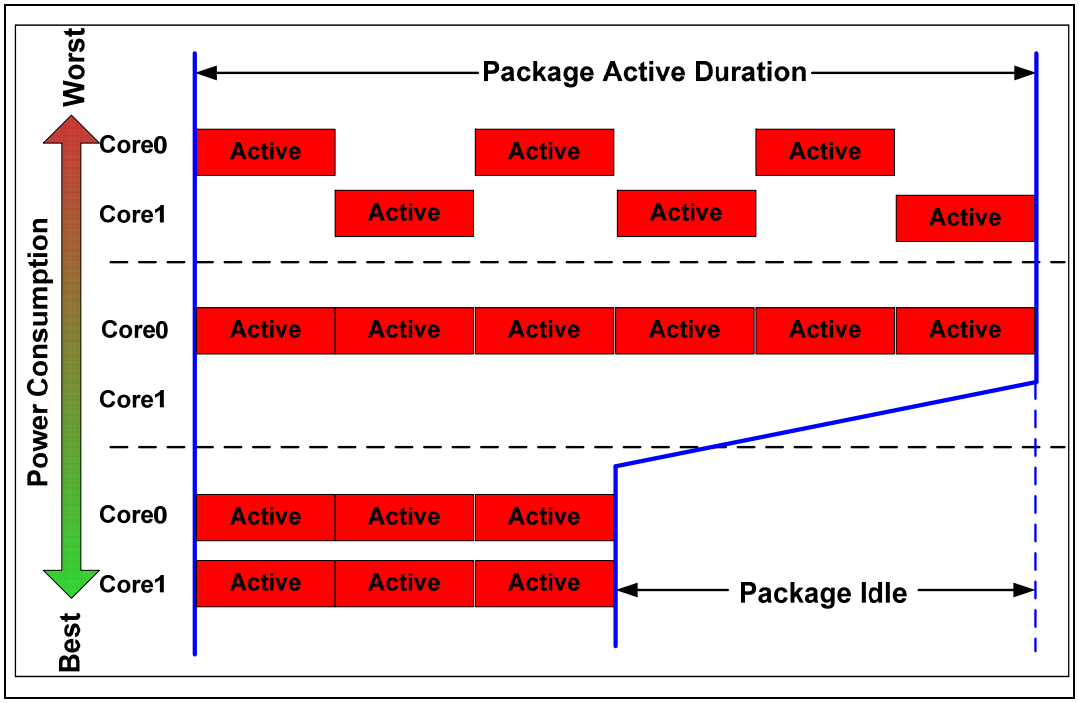
- Minimize synchronization between threads. This includes, not only explicit use of synchronization APIs, but also APIs that implicitly block the execution of the thread.
- When it is required to use APIs that block, it is recommended to coalesce and finish the use of such APIs in a short amount of time rather than spreading them over a long duration. This will help to reduce the duration of C-state demotion.

2.2 Multi-Core Scheduling

In multi-core architectures, performance increase can be achieved by parallel execution on the cores. This can also lead to power savings as the job will be finished faster and system resources and processor package level resources typically shared by the cores can be put into low power mode.

Figure 8 below shows that platform power consumption for a specific task will be highest when it is executed on multiple cores with no parallelism. There will be an increase in power consumption both due to frequent core C-state switching and due to processor package and system level resources being on for a long duration. When the task is executed on a single core, there is some reduction in power consumption as the frequency of C-state switching reduces. Platform power consumption is optimal when the task executes on both cores concurrently as this will reduce C-state transitions and significantly reduce the power consumption of processor package level resources and other platform level resources shared by the processor cores. Concurrent execution on multiple cores also results in better performance. This is a good example where better performance and better battery life can coexist.

Figure 8: Maximizing Multi-core Concurrency Reduces Power Consumption

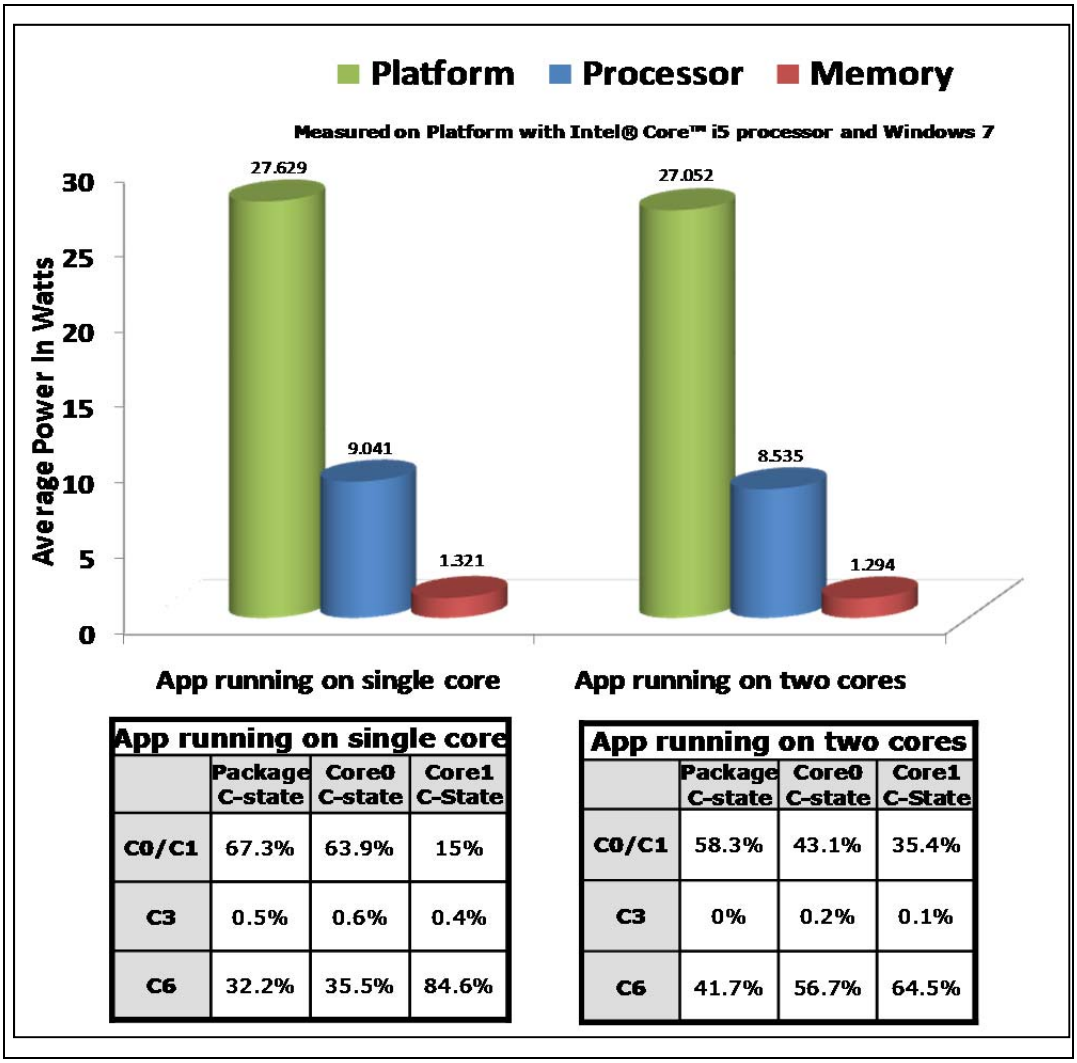


2.2.1 Impact of Inefficient Multi-core Scheduling: A Case Study

Figure 9 below shows the power impact of running a media application on a single processor core in comparison with when the application is run on two cores. For this case study, the total CPU execution requirement as defined by the CPU core C0 residency is about 79%. In the first test case, the media application runs completely on CPU Core0 taking the Core0 C0 residency to ~64%. The Core1 C0 residency is ~15% which is caused by the OS and other software components.

In the second test case, the application is allowed to run on CPU Core0 and CPU Core1 concurrently taking the Core0 C0 residency to ~43% and the Core1 C0 residency to ~35%. This means more activity overlaps between Core0 and Core1 and helps to reduce package C0 duration. In the first run, processor package C0 residency is ~67% and it drops to ~58% in the second run. The concurrent execution of the application on the two processor cores causes a ~10% increase in processor package C6 residency which leads to ~500 mW power savings in processor package power consumption.

Figure 9: Power Impact of Concurrent Execution



App running on single core	Package C-state	Core0 C-state	Core1 C-State
C0/C1	67.3%	63.9%	15%
C3	0.5%	0.6%	0.4%
C6	32.2%	35.5%	84.6%

App running on two cores	Package C-state	Core0 C-state	Core1 C-State
C0/C1	58.3%	43.1%	35.4%
C3	0%	0.2%	0.1%
C6	41.7%	56.7%	64.5%

This application is not very CPU bound. In CPU bound applications, where the percentage increase in concurrency is much larger, the power savings will be much higher.

In media type applications, like the example shown above, a certain amount of code has to be executed during a fixed time interval. It is important to note that for such applications though there is no overall increase in performance due to activity overlap between the threads, there is significant power savings as code execution within each timer interval completes faster.

Energy-efficiency not only improves when concurrency between the CPU cores is increased, but it also improves when concurrency between the CPU cores and the Graphics processing cores increases as the package level and other platform resources are also shared between the CPU cores and the graphics processing cores. When work

is being offloaded to the graphics processing cores, concurrently executing on the CPU cores instead of serializing execution will improve overall platform energy-efficiency.

2.2.2 Turbo Boost Impact of Inefficient Multi-core Scheduling

Performance States (P-states) allow the processor to switch between several discrete voltage and frequency combinations. A lower performance state is used when the processor activity is low. ACPI P0 state is the highest performance state. The turbo boost feature is hardware controlled where the processor runs faster than its base operating frequency while remaining within the power, current and temperature component specification limits in the presence of a performance demanding (P0) workload.

The turbo feature gives a performance boost when performance is required while also increasing power consumption. A single threaded CPU intensive application can keep a core in C0/P0 state for a long time enabling the turbo feature. The power impact can be much larger compared to when the same application is run on multiple cores. Applications must be written to efficiently use all the cores for power savings such that the turbo feature is only engaged when additional performance is needed, and will be done so by a combination of OSPM policy, OEM preference, and dynamic system conditions. Turbo is not under direct control of the application.

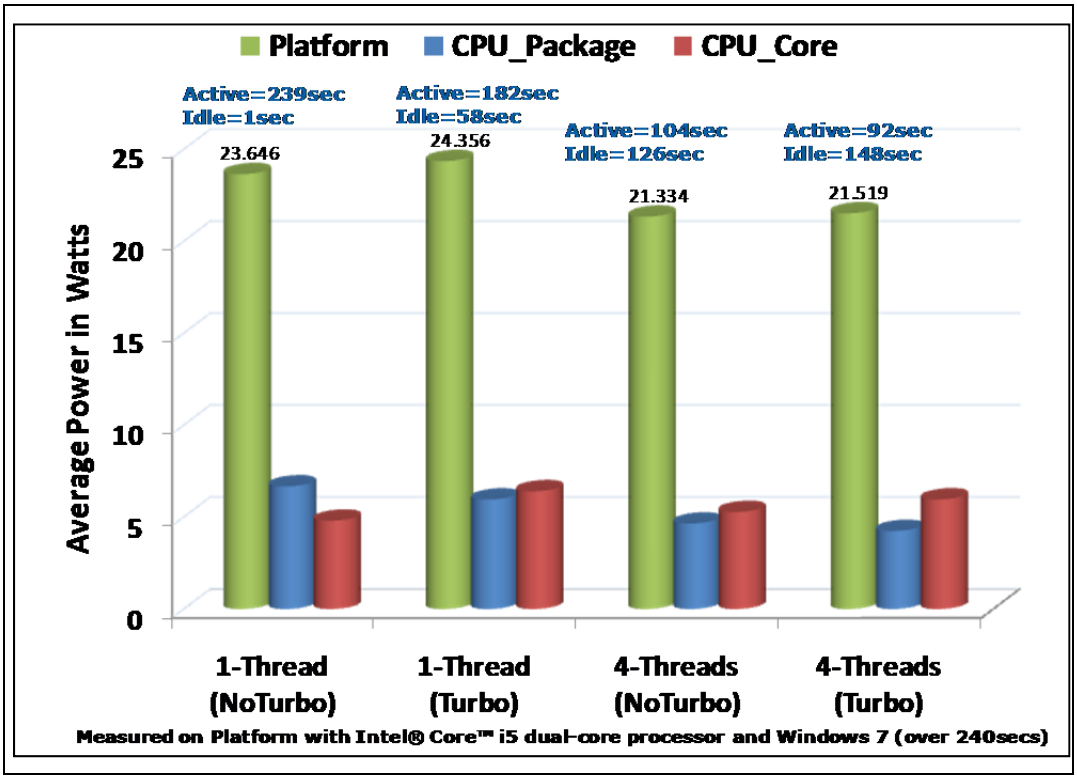
2.2.3 Turbo Impact of Inefficient Multi-core Scheduling: A Case Study

Figure 10 below shows the power impact of running a CPU intensive application on a single thread and on a hyper thread enabled dual core processor with and without turbo boost. The average power is measured over a window of 240 secs.

For the single thread case, in this specific example of a virus scan application, the average power is higher when the turbo boost kicks in, but there is also increased performance as the task gets executed faster and finishes in 182 secs instead of 239 secs. Since the task finishes faster, the CPU package power consumption is lower. There can be other usage models where the power consumption with turbo is lower for single threaded applications, if the performance gain and faster execution sufficiently reduces the time the processor actively executes allowing the processor package and other platform components to enter low power modes sooner.

As can be seen from the graph below, using multiple cores concurrently will give both a performance advantage and a power advantage. When the task is executed concurrently on all four logical CPUs with turbo enabled, it completes in 92 secs (with average platform power consumption of 21.519 W) compared to 182 secs when it is executed on a single thread with turbo enabled (with average platform power consumption of 24.356 W). This is a win-win situation for performance and power saving.

Figure 10: Power Impact of Turbo Mode for Single-core and Multi-core



2.2.4 Software Recommendations

- Maximize concurrent execution between the CPU cores so that the power consumption of shared package level and platform level resources can be minimized
- Maximize concurrent execution between CPU cores and Graphics processing cores when relevant
- Maximizing concurrent execution on multiple CPU cores will not only increase performance by faster completion of tasks, it will also provide a power advantage

2.3 CPU Utilization

A general recommendation for all software components is to improve computational efficiency (performance) and reduce CPU utilization to enable longer idle periods where the hardware components can be power managed. Platforms that are highly optimized for power will be more impacted by any increase in CPU utilization. Also, package-level CPU utilization can be higher than core-level CPU utilization in multi-core systems.

2.3.1 Frequency and Duration of CPU Utilization

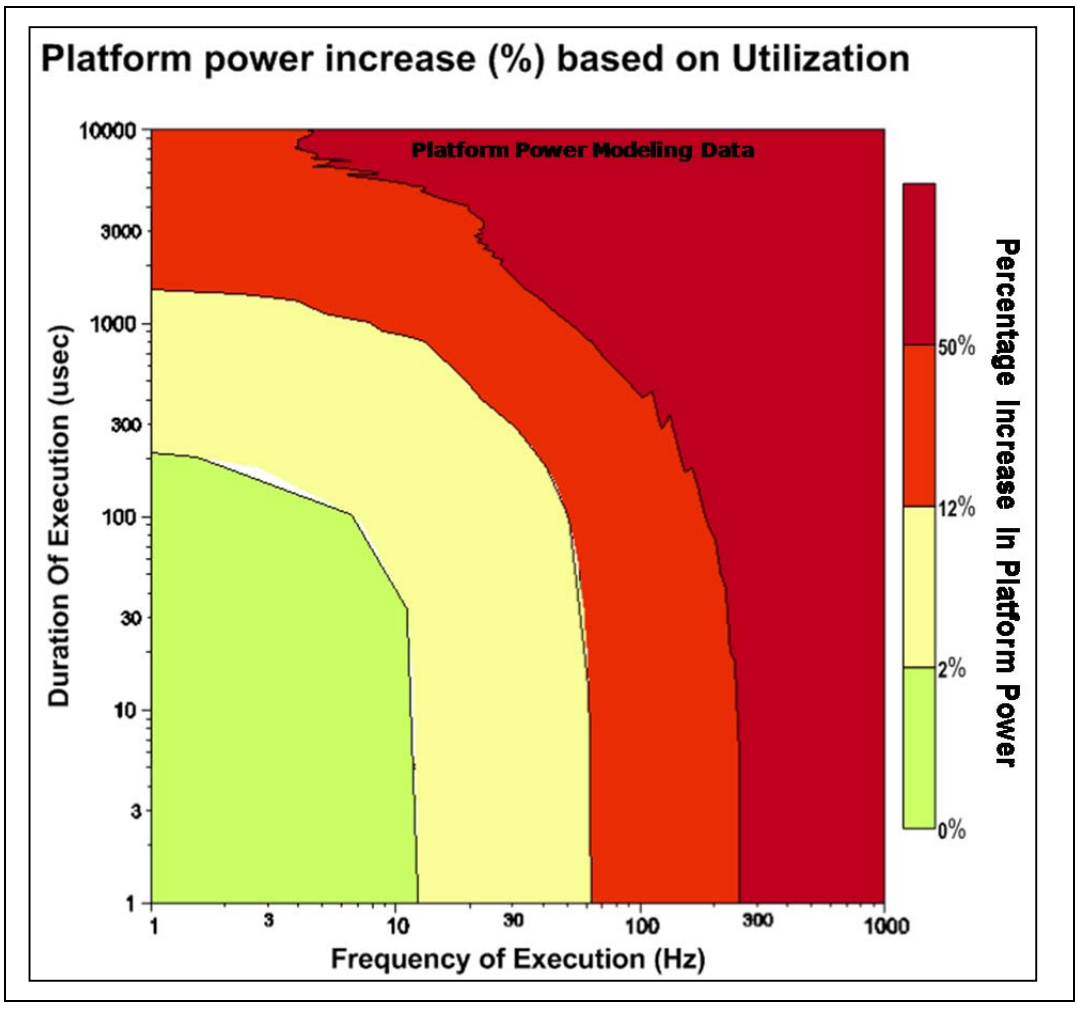
Total CPU utilization alone does not always accurately reflect platform level energy-efficiency. Two metrics that need to be considered are:

- Frequency of code execution
- Duration of code execution

For the same CPU Utilization, a task with higher frequency of execution and lower duration during each execution will burn much more power. This is especially relevant for idle and semi-idle workloads where the platform goes into deep low power states during the idle period.

Figure 11 below shows a graph depicting the platform power increase (%) from baseline idle power (no applications or services running) when varying frequency and duration of execution.

Figure 11: Power Impact of Varying Frequency and Duration of Execution



It can be seen from the graph above that when a task executes every 10 msec (Frequency of execution = 100 Hz) and runs for 10 µsec, there is ~20% increase in platform power compared to platform idle power. When the same task executes every 100 msec (Frequency of execution = 10 Hz) for 100 µsec, there is ~3% increase in platform power.

Energy-efficient software applications or services running on a predominantly idle platform, should keep the increase in platform power consumption compared to platform idle power to ~2% or less (Green zone). Applications and services that increase platform power by more than ~12% when idle are not power friendly (Red zone) and can have a significant impact on battery life.

2.3.2 Software Recommendations

- Focus on CPU Utilization for idle workloads. If background activity is absolutely required, reduce the frequency of execution to avoid frequently bringing the platform out of deep power down states.
- Drive down CPU Utilization for active workloads
- Besides monitoring CPU utilization, also monitor frequency of execution. The CPU utilization may be very low, but if the frequency of execution is high then platform power consumption will be adversely impacted.

2.4 Periodic Timers

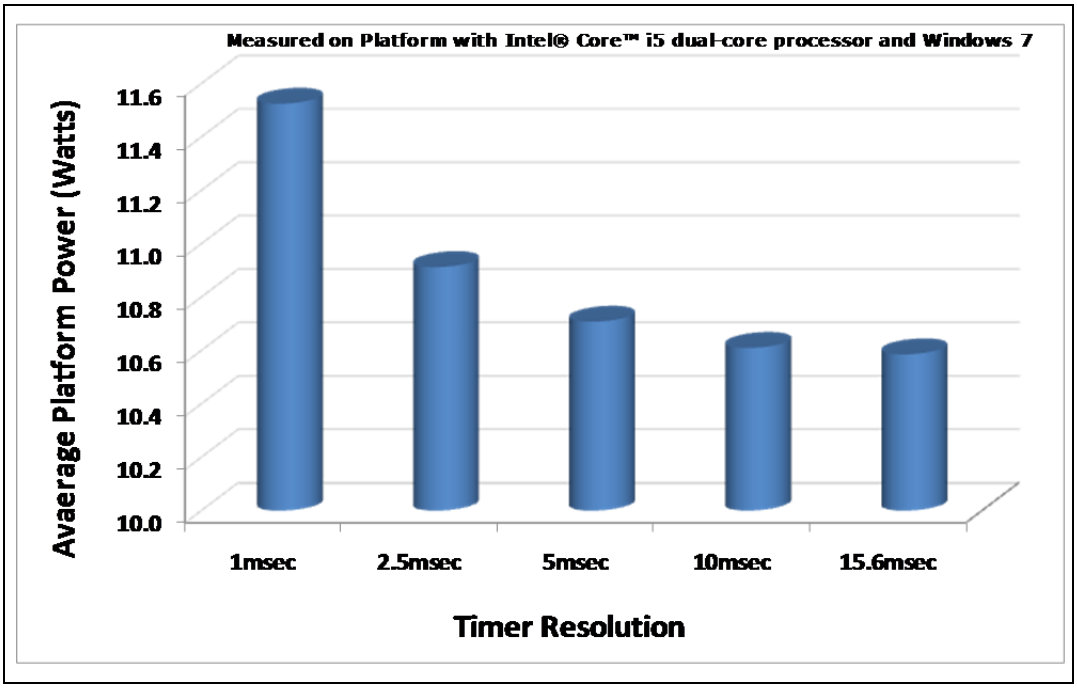
2.4.1 Timer Resolution

The Microsoft Windows Operating System scheduler* is driven by a periodic platform timer with a typical duration of 15.6 msec. Several multimedia applications increase the timer resolution and this has a significant impact on platform power consumption. Frequent timer interrupts results in more C-state transitions burning more power and when the timer resolution is very high (e.g., 1msec), several of the deep power management features in the platform get disabled.

Some non-multimedia applications also increase the timer resolution to get better performance from disabling deeper C-states. This is not good practice as successive generations of processors are optimized for both more power savings and for improved C-state exit latencies. Such applications, often developed and tuned on older platforms may consume more power with no performance benefits on newer platforms.

Figure 12 below shows the impact of increasing periodic timer resolution on platform power

Figure 12: Power Impact of Increasing Periodic Timer Resolution



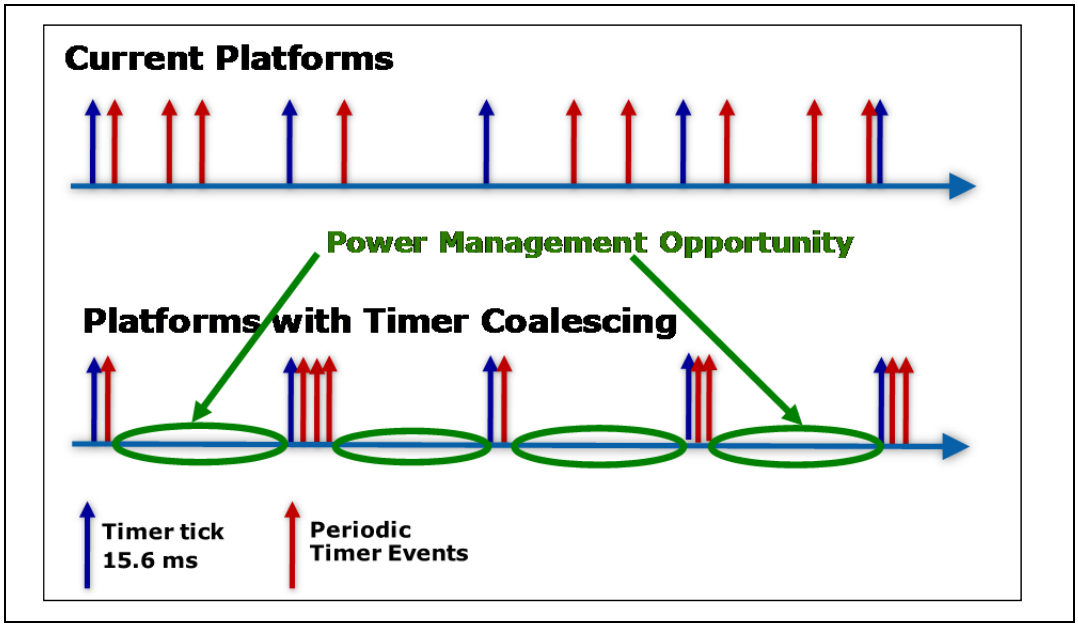
For performance reasons, media and gaming applications use high resolution timers. But it is important that application developers understand the power impact of using these high resolution timers and set it to the lowest resolution that meets the performance requirements of the application for the specific platform.

2.4.2 Timer Coalescing

The new Timer Coalescing API in Microsoft Windows 7* enables callers to specify a tolerance for due time. This enables the OS Kernel to expire multiple timers at the same time thereby reducing the frequency of timer interrupts. Since the application specifies the tolerance, there is no adverse impact to performance.

As seen in Figure 13 below, coalescing timer expiry increases the idle period duration providing opportunities for more efficient power management. As future platforms introduce more enhanced power management features, the power benefits realized by coalescing timer expiry will also increase.

Figure 13: Power Management Opportunities by Coalescing Timers



2.4.3 Software Recommendations

- Minimize the use of APIs that shorten timer period.
- If the application needs to change timer period, use the lowest timer resolution possible that meets the performance requirements of the application
- If the application requires a high-resolution periodic timer, increase the timer resolution only when the application is active
- Software applications should use the new timer coalescing API supported by the OS to align its activities to other application programs' activities.

2.5 Disk and Registry Activity

Software components can increase the power consumption of disk devices by frequently writing or reading file or registry data. Frequent intermittent accesses not only preclude the drive from going to standby (spin-down), it also precludes it from going to low power idle state shown in the table below.

Table 1: HDD Power Saving Modes

State	Power	Entry	Note
Standby	~0.1 W	OS Control	Spin down
Low Power Idle	~0.6 W	Automatic after ~10sec	Head unloaded
Active Idle	~0.9 W	Automatic after <1sec	Servo, R/W circuit off
Performance Idle	~1.8 W	After command completion	No power saving

2.5.1 Software Recommendations

- Applications and services must avoid generating frequent intermittent disk and registry activity especially when the system is predominantly idle
- Reduce file and registry accesses, which bypass file system cache. Frequent calls to file open/close functions for write, file flush operations, etc. should be avoided.

2.6 General Guidelines for Applications

This section provides a brief list of some important power friendly recommendations for software applications and services that have not been covered in this document but are required for platform energy-efficiency. Some of the references in Chapter 4 cover these items more in detail.

- Avoid polling and spinning in tight loops as this precludes the processor from effectively using C-states and P-states and all other platform resources are kept in the highest powered state
- Improve computational efficiency. The improvement in performance will increase idle periods when the platform resources can be power managed
- Adjust performance according to the Operating System supported power policy set by the user
- Reduce platform resource usage and background activity especially when on battery power
- Applications should register for and respond to common power management events from the Operating System
- If the system display is off for power savings, application must not perform unnecessary graphics rendering

- Applications must avoid doing periodic disk and registry accesses during idle workloads. Disks must be allowed to spin down to save power

2.7 Device Specific Applications or Services

Many drivers come with user-mode services or applications. These device drivers can significantly impact platform power consumption if they are not energy-efficient in their behavior. Some general guidelines for device drivers:

- Avoid polling and spinning in tight loops
- Reduce frequency of communication with device. This will generate frequent interrupts. Coalesce activity as much as possible to reduce frequency of interrupts
- Improve computational efficiency
- Reduce background activity. Especially if the device is in low power/suspend state.
- Adjust to user power policy by reducing resource usage or increasing performance accordingly
- If timers are required, use the Timer Coalescing API

3 Debugging Power Issues

When investigating and debugging power issues on Intel® Architecture mobile platforms it is recommended to use the following tools:

- Intel Battery Life Analyzer – A tool developed by Intel specifically to investigate potential power inefficiencies on the platform.
- PowerCfg.exe – A built-in Microsoft Windows 7* command used to control power settings and configure power settings.
- Microsoft Windows Performance Toolkit* - A collection of tools by Microsoft for capturing and viewing detailed system and application behaviors.
- Windows Sysinternals Process Monitor – An advanced monitoring tool for Microsoft Windows showing real-time data on file systems, registry activity, and thread/process details.

All examples shown in this chapter are captured on Intel® Core™ i3/5/7 mobile processor based platforms with the Microsoft Windows 7 operating system unless otherwise noted. Results may vary depending upon environmental changes.

Please see the References section for the description and availability of these tools.

3.1 Debug Strategy

When debugging multiple issues at once it becomes difficult to separate the impact of one change from the others. When debugging particular software power issue, it is highly recommended to create a baseline install before beginning; a vanilla OS install, minimal drivers, and without value-add software for the add-on devices. It is important to compare the system behavior before and after installing the software under investigation.

3.2 CPU / Chipset Power Issues

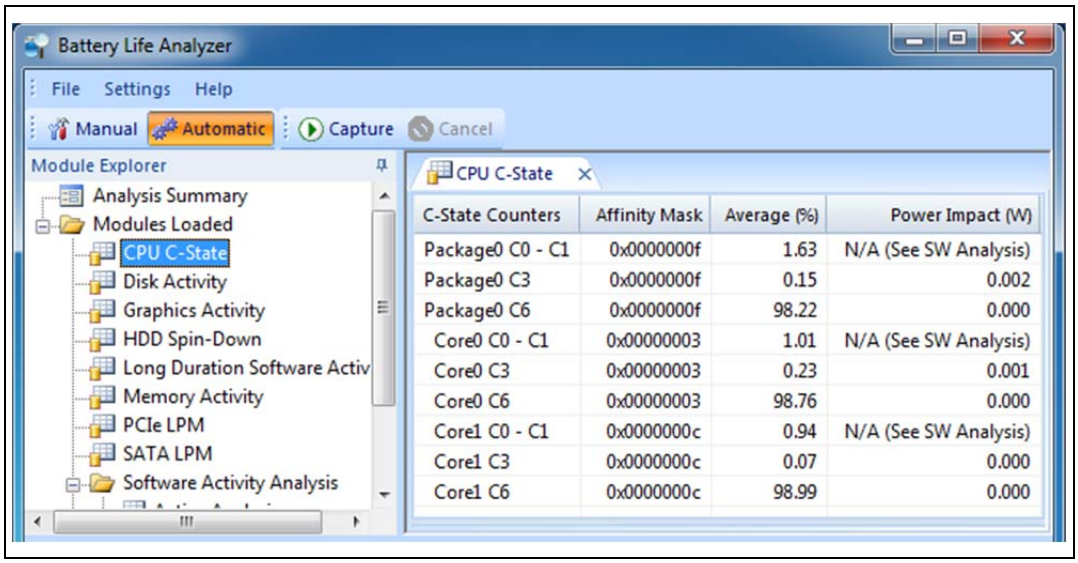
3.2.1 CPU C-State Residency

Measuring CPU C-state (package and core) residency is the recommended method to start debugging software power consumption. From CPU C-sate residency numbers, the following information can be determined:

- Are the CPU and chipset being optimally power managed?
- What is increasing platform power consumption?

Using the CPU C-State Analysis feature of the Battery Life Analyzer tool it is possible to measure CPU C-state residency on the platform. An optimally power managed system typically shows a high (>98% for core C-state, >95% for package C-state) residency in the deepest C-state supported by the platform. Figure 14 below provides an example.

Figure 14: C-State Residency Numbers from Clean Idle System



If high residency values are not seen in the deepest C-states, there are several common causes that can be investigated.

1. If Core C0 state residency is high (>1.5%)
    - A process or driver's activity is consuming CPU cycles - Proceed to Section 3.2.2: "CPU Utilization Issues" to determine which software component is consuming CPU cycles.
    - The OS timer resolution has been increased by an application - Proceed to Section 3.2.3 "Timer Resolution Issue" to determine which software component changed timer resolution.
    - Excessive driver activity - Proceed to Section 3.3 "I/O Issues" to determine which driver is consuming CPU cycles
2. If Core C3 state residency is high (>1.5%)
    - Frequent C-state transitions caused by device interrupts - Proceed to Section 3.3 "I/O Issues" to determine which device driver is causing frequent activities

- Frequent C-state transitions caused by software activity - Proceed to Section 3.2.4 "CPU Activity Frequency Issues" to determine which software component is causing frequent activities.

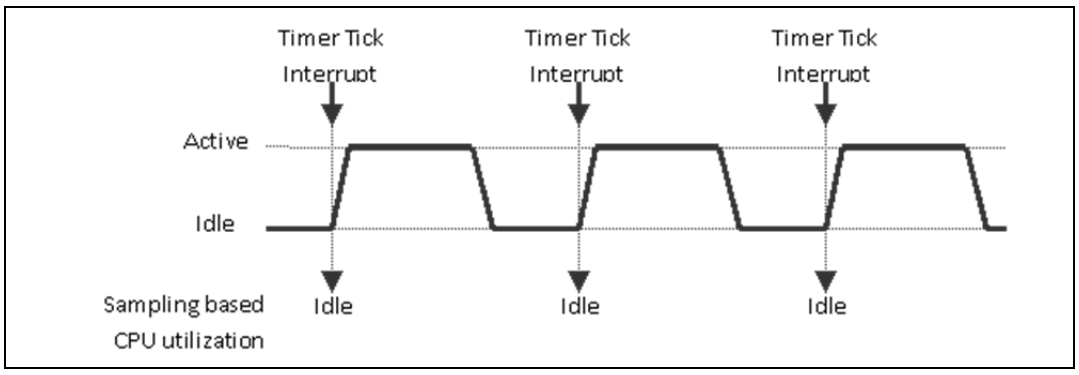
3. If Package C0~1 state residency (for Huron River platform, package C2 state residency) is significantly higher than all core's C0 residency
   - CPU package C state pops up to C1 (or C2, in case of Huron River platform) state to respond to snoop cycles generated by bus master devices
   - Proceed to Section 3.3.5 "Frequent Snoop Cycle Issue" to determine which device is causing the issue.

3.2.2 CPU Utilization Issues

There are many tools for CPU utilization measurements, but most of them don't reflect the real impact of software activity. This can be because of any of the following reasons:

- CPU utilization measurements with a "sampling based" method
  Most tools are built on OS accounting APIs, which use sampling based information. Here, "sampling based" means CPU utilization is observed at every timer tick interrupt, which usually happens every 15.6 msec. As illustrated below, software activities starting at a timer tick interrupt and ending before the next timer tick are not observed by these tools at all. This type of software behavior is common among many background applications and media (isochronous) applications.

Figure 15: Typical CPU Activity Pattern of Background Process and Sampling based CPU Utilization

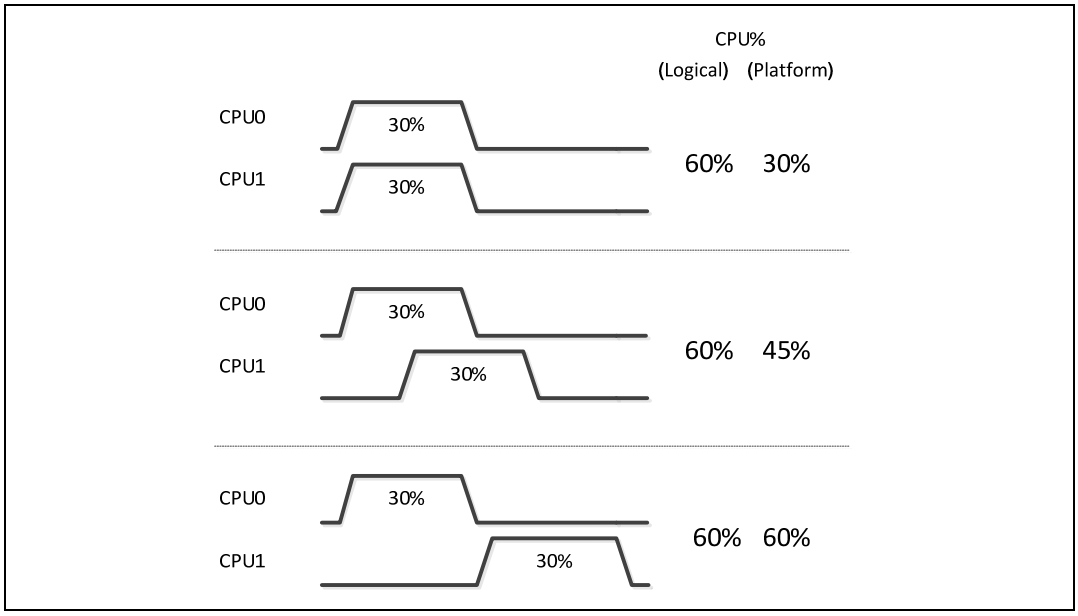


- As discussed in Section 2.2 "Multi-Core Scheduling" , multi-core CPU's power consumption is determined by the total of 1) the duration each Core is active, and 2) the duration the Package is active (defined as the duration at least one Core in the Package is active). Arithmetic average of all logical CPU utilization, as measured by most tools, only reflects the first part of the power.

To address these issues, Battery Life Analyzer uses fine grain process information (µ sec resolution time stamp for activity start and stop) and shows the total active duration of both package and logical CPU. It is highly recommended to use Intel's Battery Life Analyzer instead of traditional CPU utilization tools to get the best picture of software's impact to the platform power.

In addition, Battery Life Analyzer shows CPU utilization with two numbers. One (Logical) is the total of each logical CPU's utilization. This is a good representation of the power consumed by a single CPU core. The other one (Platform) is the duration at least one logical CPU is active. This is a good representation of the power consumed by the package (except for cores). As illustrated below, Platform CPU utilization can vary for the same Logical CPU utilization (for the same amount of instruction execution) depending on how much overlap is happening between logical CPUs. It is, especially for workloads with higher CPU utilization, recommended to lower the Platform CPU utilization for the same Logical CPU utilization.

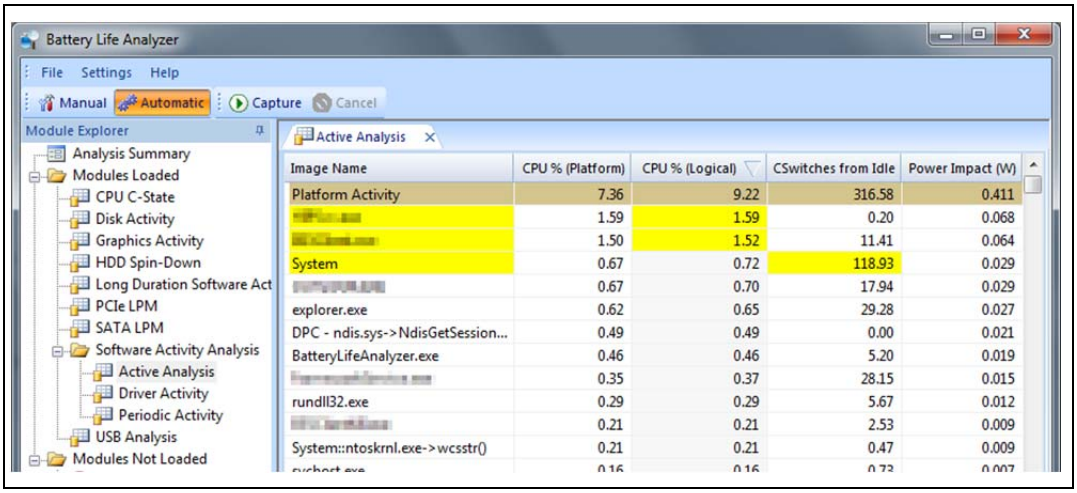
Figure 16: Battery Life Analyzer CPU Utilization Metrics - Logical & Platform



3.2.2.1 Identify Software Components with High CPU Utilization

In Figure 17 an example screen shot from Battery Life Analyzer highlights how two background tasks can create high CPU utilization on an otherwise idle platform.

Figure 17: Battery Life Analyzer - Active Analysis



To accurately measure the impact of an application, just reading the component numbers alone is not enough. One component can directly affect the behavior of other components negatively. For example, if the process of interest is communicating with a server process through the COM interface, the activity increase is also seen in the server process. The best practice of measuring the impact of particular software is, as described in Section 3.1 "Debug Strategy", to compare a clean setup with and without the particular software and compare the difference in the "Platform Activity" row.

3.2.2.2 Understand Why CPU Utilization is High

Once the high CPU utilization component is identified, the next step is to understand why CPU cycles are consumed by the component. For the developer of the software, it may not be difficult to determine the problem(s). For those without this direct knowledge of the software, the following tools will help explain the behavior of the processes:

- Hotspot analysis (with Intel® VTune™ Amplifier XE, Microsoft Windows Performance Analyzer*) will show where the most CPU cycles are spent.
- File, registry, and network I/O analysis (with Sysinternals ProcMon tool) will give some understanding about what kind of activity is happening. See Section 3.3 "I/O Issues" for more details.

Based on the nature of the activity, it can be determined if those activities are really necessary or not.

3.2.3 Timer Resolution Issue

Intel's Battery Life Analyzer is designed to capture timer resolution change requests from application programs. For example, Figure 18 shows svchost.exe (PID 888) and BadProgram.exe (PID 164) requesting a timer tick period of 10 msec and 1msec

respectively. As a result of these requests, the entire platform's timer tick period is set to the shorter of the two requests; 1 msec.

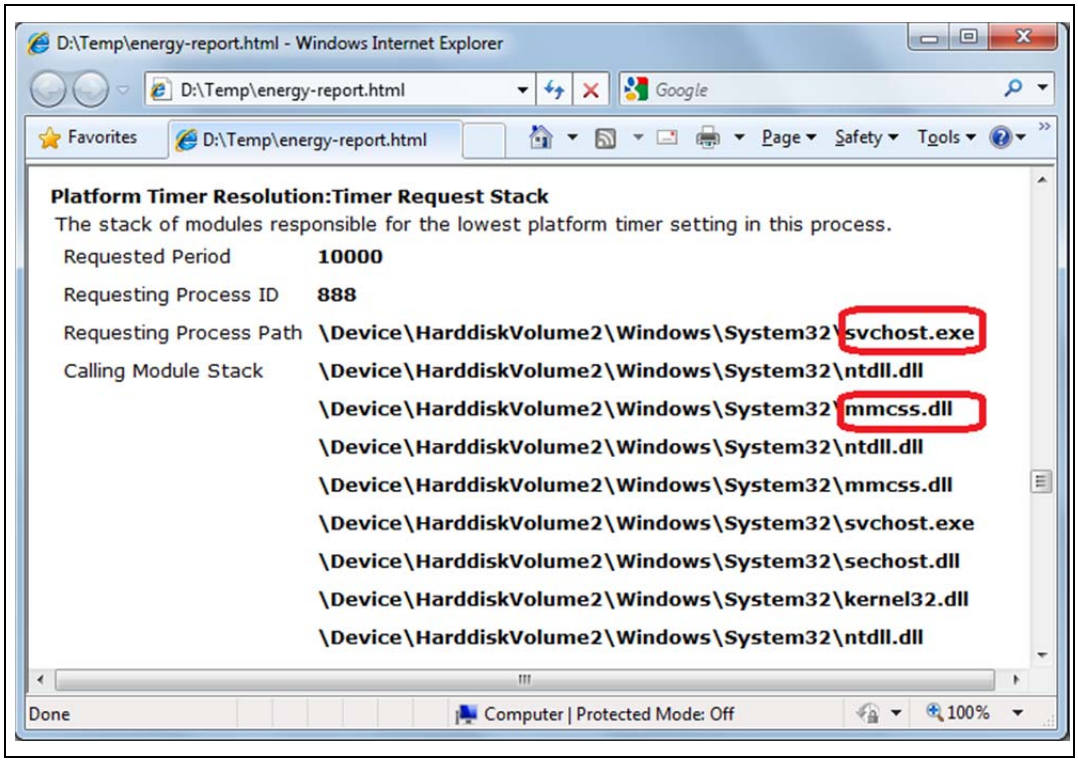
Figure 18: Battery Life Analyzer – Timer Resolution

Active Analysis | Driver Activity | Periodic Activity | P-State Analysis

Image Name	PID	CPU % (Platform)	CPU % (Logical)	CSwitches from Idle	Timer Tick Period (ms)
Platform Activity		9.49	10.56	751.60	1.00
svchost.exe	888	0.28	0.29	102.27	10.00
BadProgram.exe	164	0.00	0.00	0.00	1.00
System	4	0.30	0.33	60.74	
smss.exe	304	0.00	0.00	0.00	
svchost.exe	352	0.01	0.01	2.09	
csrss.exe	432	0.01	0.01	15.45	
wininit.exe	484	0.00	0.00	0.00	

To determine which part of each process is requesting the timer resolution change, run the "PowerCfg" command with the "/energy" option. ("PowerCfg" is a Microsoft Windows 7 built-in command, and has to be executed with Administrator privilege.) This program observes software activity for about a minute, saving the results in the "energy-report.html" file. The process requesting the timer tick period change can be found in the Information section near the end. Figure 19 below shows the example with the svchost.exe process (PID: 888) in Figure 18.

Figure 19: PowerCfg - Timer Resolution Change Request



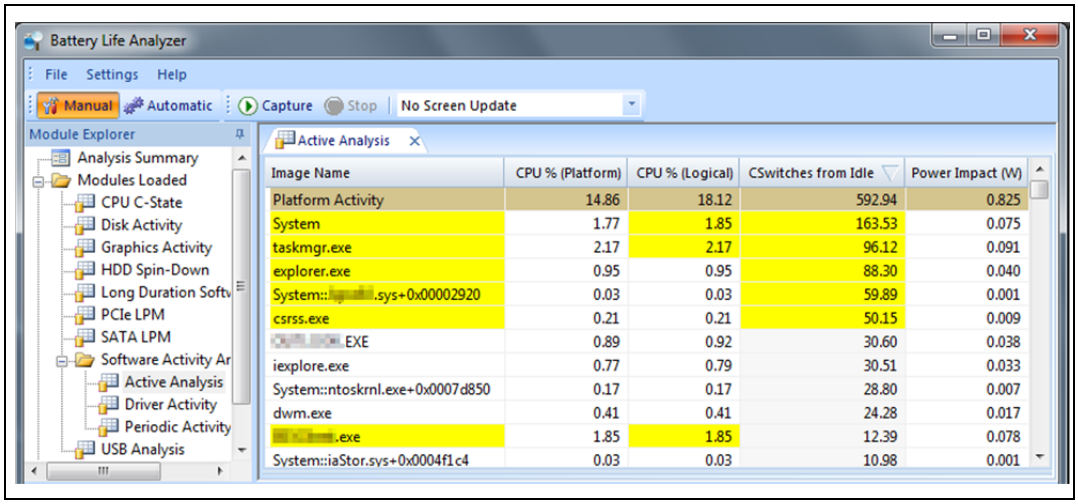
This example shows svchost.exe is requesting timer resolution change through the code in mmcss.dll (DLL for one of the system provided service – MMCSS: Multimedia Class Scheduler Service). This is a part of the OS audio sub-system behavior when multimedia applications are running and can be safely ignored.

3.2.4 CPU Activity Frequency Issues

3.2.4.1 Identify Software Components Causing Frequent Transition

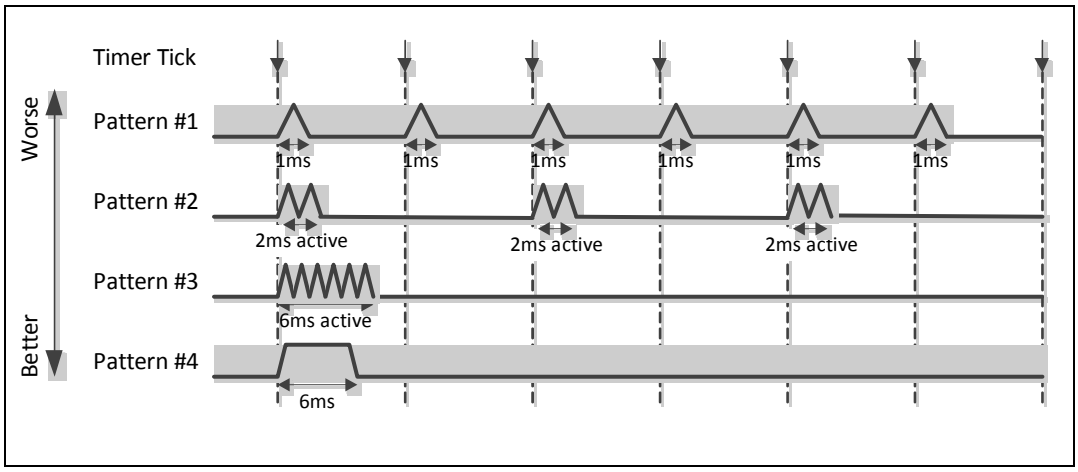
A CPU frequently transitioning between active and idle states consumes more energy, as described in Section 2.1, than a CPU kept in an idle state. Battery Life Analyzer provides two methods to identify the process causing frequent transitions. The first method is to count the context switches from the idle thread to the active thread. The "CSwitches from Idle" column in the Software Activity Analysis module's Active Analysis tab shows how many C-state transitions were caused by each component, including processes and drivers. After running the Software Activity Analysis module, the data is sorted by CPU utilization by default. Click the column header for "CSwitches from Idle" to find out which component caused most C state transitions.

Figure 20: Battery Life Analyzer - Number of C-state Transition Caused by Each Component



In the example shown above, System, csrss (Win32 subsystem), and dwm (Desktop Window Manager) are OS components. High CSwitch activities made by these components are usually caused other processes' activity.

Figure 21: Different CPU Activity Patterns with Same CPU Utilization



While "CSwitches from Idle" numbers can tell the difference between Pattern 4 and Pattern 1~3, it cannot tell the difference between Pattern 1~3. To distinguish the difference between these patterns, use the Periodic Activity tab from the Battery Life Analyzer's Software Activity Analysis module.

Periodic Activity tab summarizes a software component's activity by timer tick period and displays the histogram by the amount of activity that happened during each period. For the activity patterns shown above, the following histogram values will be shown as the result of analysis.

Figure 22: Battery Life Analyzer – Periodic Activity Analysis with Various CPU Activity Patterns

	0µs	…	≤1ms	≤2ms	≤5ms	>5ms
Pattern #1	**0% (=0/6)**		**100% (=6/6)**			
Pattern #2	**50% (=3/6)**			**50% (=3/6)**		
Pattern #3	**83.3% (=5/6)**					**16.7% (=1/6)**
Pattern #4	**83.3% (=5/6)**					**16.7% (=1/6)**

Figure 23 below shows screen shot example from an idle system with many background processes running small activities.

With the Periodic Activity feature, the following can be determined:

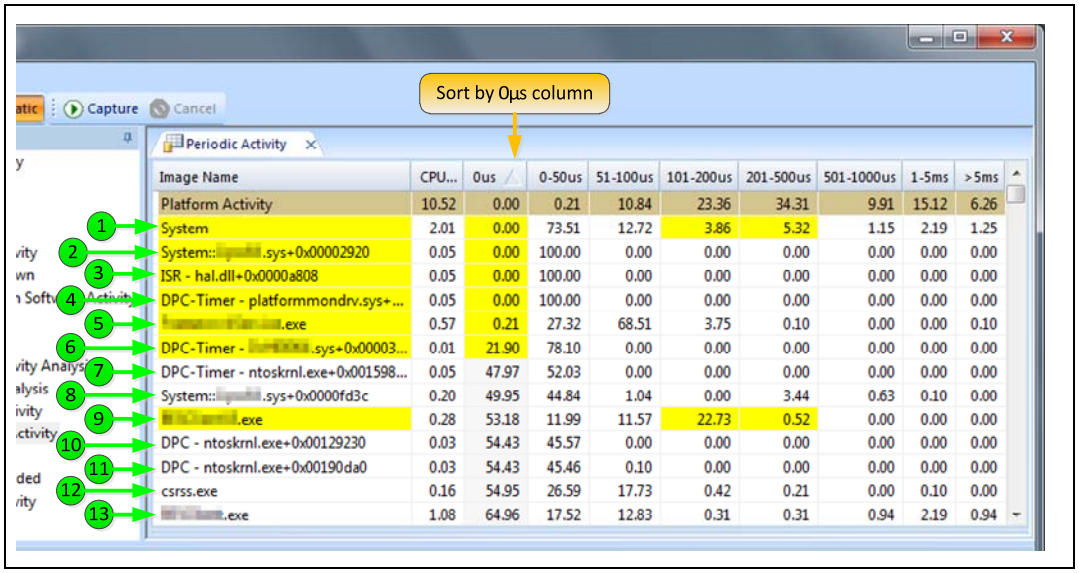
1. How frequently each component is active

   The 0µsec column shows the percentage of the timer tick period without any activity by each component. For example, a value of 100% in this column means the software component was not active during the data collection. This is the desired behavior for an idle platform scenario. Alternatively, a value of 0% means the software component had some activity at every timer tick period. Such components tend to have more impact to the platform power consumption for the same total CPU utilization.

2. How many activities are happening at each tick

   If there's more than a few 100s of µsec (the exact threshold is CPU architecture dependent) activities per timer tick period, the CPU will start to demote to a shallower C-state. By looking at the top row, "Platform Activity" (total activity of all components), it is possible to determine how often long (> 200 µsec) activity is happening and which component is contributing towards the activity.

Figure 23: Battery Life Analyzer - Periodic Activity Analysis



In Figure 23shown above, notice the following:

- #3 (ISR – hal.dll) is an OS timer tick itself, and it's normal for this component to be active every timer tick. This can safely be ignored.
- #4 (DPC – Timer – platformmondrv.sys) is the activity of Battery Life Analyzer. This can safely be ignored, too.
- #2 (System::****.sys) is the activity of a 3rd party driver (kernel mode thread). Although the duration of activity is always very short (less than 50 µsec), it is generating activity at every timer tick (0µsec column is 0) and is not a desired behavior.
- #5, #6, #8, #9 are all 3rd party components generating frequent activity. Especially, #9 is generating relatively long activity frequently (22.73% in 100~200 µsec range) and potentially causing C-state demotion.

3.2.4.2 Understand Why Frequent C-State Transition is Happening

Once the software causing frequent C-state transitions is identified, the next step is to determine why and which part of the code is causing the issue. The exact procedure varies from case to case. This section explains two of the most common cases. The first case is where there is excessive use of a short duration timer, while the second case focuses on multiple threads becoming alternatively active (IPI Storm Issue).

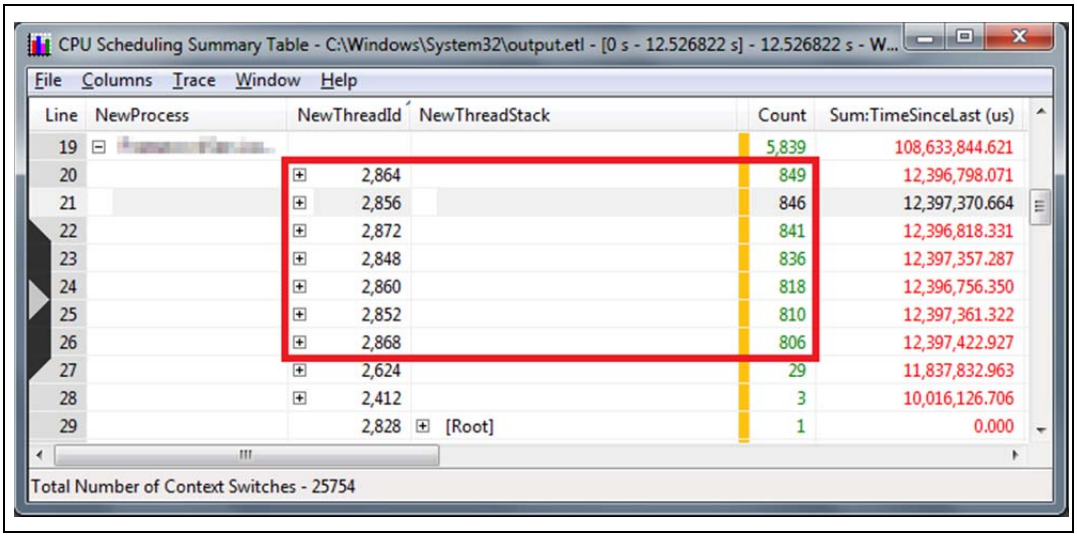
The main focus of Battery Life Analyzer is to identify the bad behaving component(s) in the platform and not meant to do the deep analysis. The Microsoft Windows Performance Analyzer (formerly xperf) can be used as a primary tool for detailed analysis. Please see the Chapter 6 "References" for the availability of this tool.

The following sections demonstrate how to identify and debug these two common causes of frequent C-state transitions.

3.2.4.2.1 Debugging Excessive Use of Short Duration Timer Issue

1. Install Microsoft's Windows Performance Analyzer to the target system.
2. (Optional) If the target system is a 64-bit platform, it is recommended to create the registry key DWORD DisablePagingExecutive with a value of 1 under HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management, and reboot the system. This registry setting will ensure a complete kernel stack dump.
3. Run cmd.exe as the Administrator
4. Start the data collection: *xperf –on DiagEasy –stackwalk CSwitch*
5. Leave the system idle for few minutes
6. Stop data collection: *xperf –d output.etl*
   This will create the log file (output.etl) in the current directory.
7. Open the log file with viewer: *xperfview output.etl*
8. Enable symbol decoding through the menu "Trace" > "Load Symbols"
   If necessary, set the appropriate symbol path at menu "Trace" > "Configure Symbol Paths"
9. Scroll down to the "CPU Scheduling" graph.
10. Right click on the graph and select "Summary Table" in the popup menu.
11. Arrange the columns in the following order (left to right) by dragging column headers,
    - NewProcess
    - NewThreadId
    - NewThreadStack
    - Separation (orange) bar
    - Count
    - Sum:TimeSinceLast (µs)
12. Look for the process that was identified as a cause of the frequent C-state transitions by Battery Life Analyzer. In this example, notice the process #5 in Figure 23.

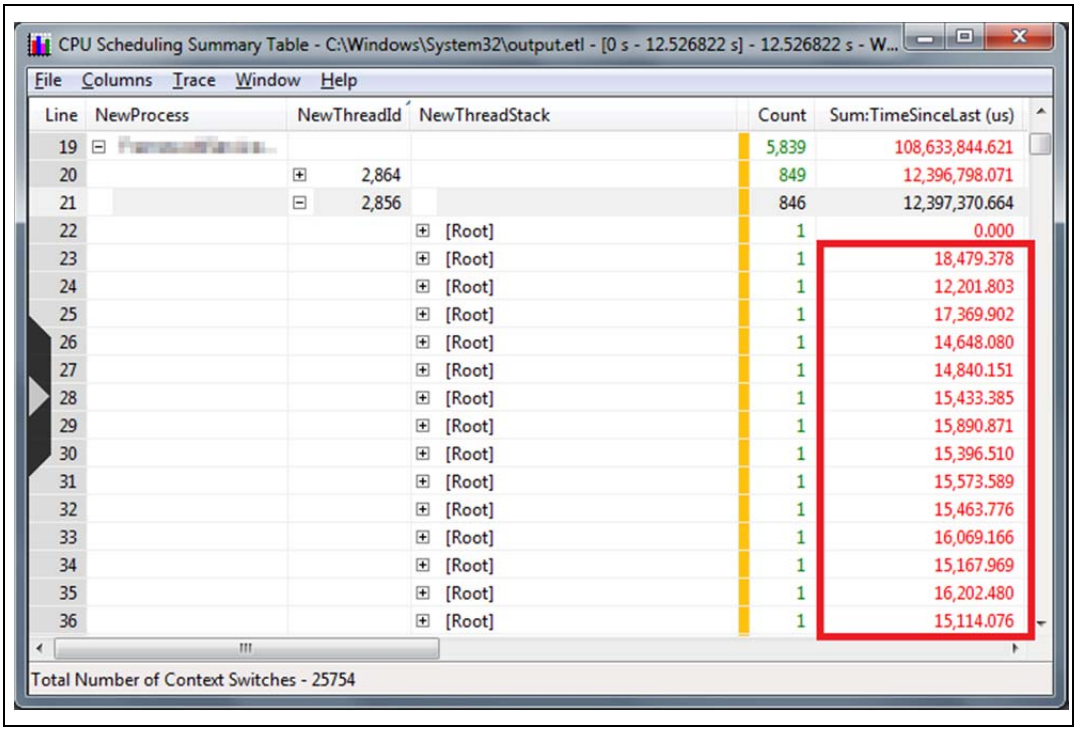
Figure 24: Windows Performance Analyzer – Thread List



Notice the list of the threads in Figure 24. In this example, seven threads (thread id:2864, 2856, 2872, 2848, 2860, 2852, and 2868) are showing high context switch numbers in the "Count" column.

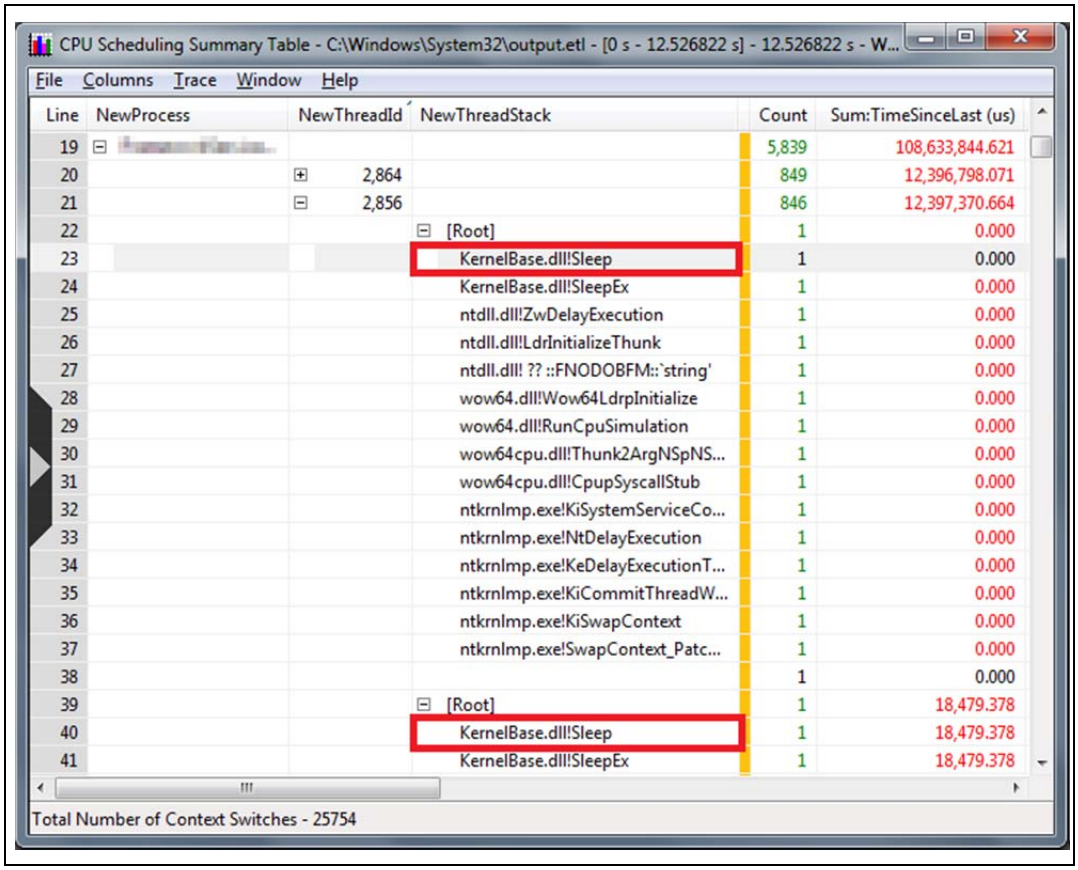
By expanding one of the threads, note each context switch per line as shown in Figure 25. "Sum: TimeSinceLast (µs)" column, shows these context switches are happening approximately 15~16 msec apart. This means each thread is generating activity every timer tick period (15.6 msec).

Figure 25: Windows Performance Analyzer – Thread Activity



Furthermore, by examining the stack dump at each context switch, it can be seen that this is caused by the repeated call of the Sleep() API.

Figure 26: Windows Performance Analyzer – Stack Dump



From this information shown in Figure 26, it is evident that this process creates multiple threads and each thread is looping with Sleep() API.

3.2.4.2.2 Debugging Occasional High C-State Transition Issue

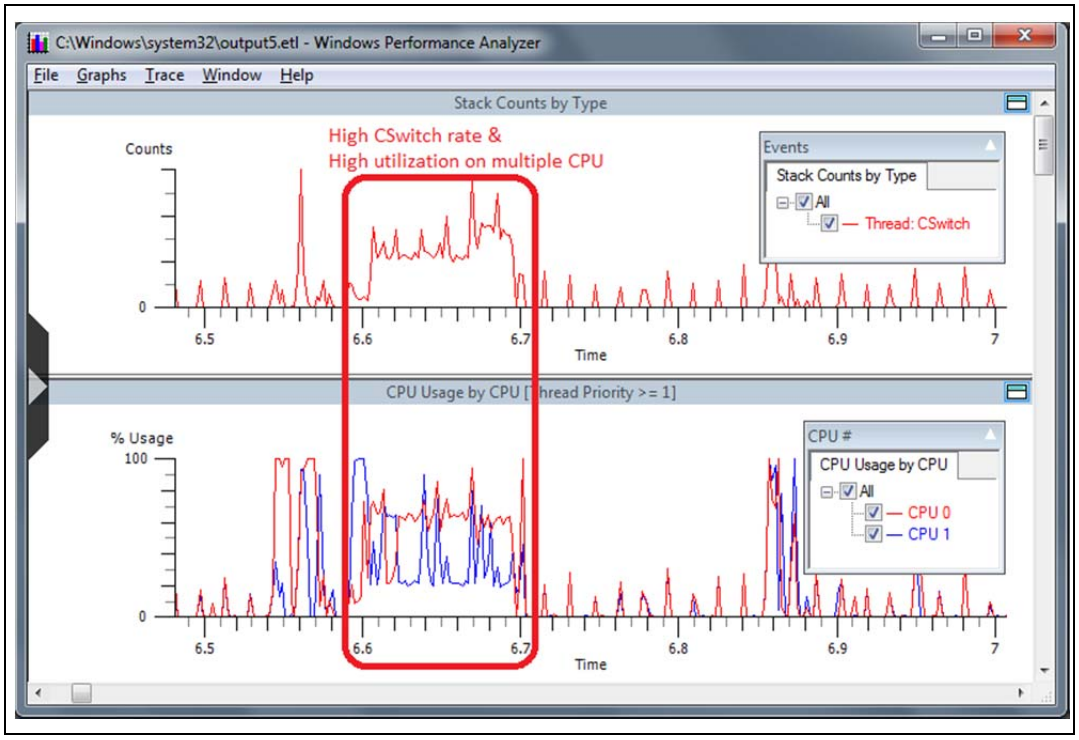
In the previous example, the problem is happening all the time making the investigation easy. Some types of programs that cause frequent C-state transitions for a limited time (but, repeats it). For such programs, the period where high C-state transitions are happening needs to be identified first. This section uses process #13 in Figure 23 as an example.

Follow steps 1 through 8 of the previous section.

9. Display "Stack Counts by Type" and "CPU Usage by CPU" graphs (the first two graphs)
10. Select 0.5 to 1.0 second range in one of these graphs, right click on that graph, and select "Zoom to Selection"
11. Scroll horizontally through the entire data and look for a section with a high number in the "Stack Counts by Type" graph (means high context switch), and

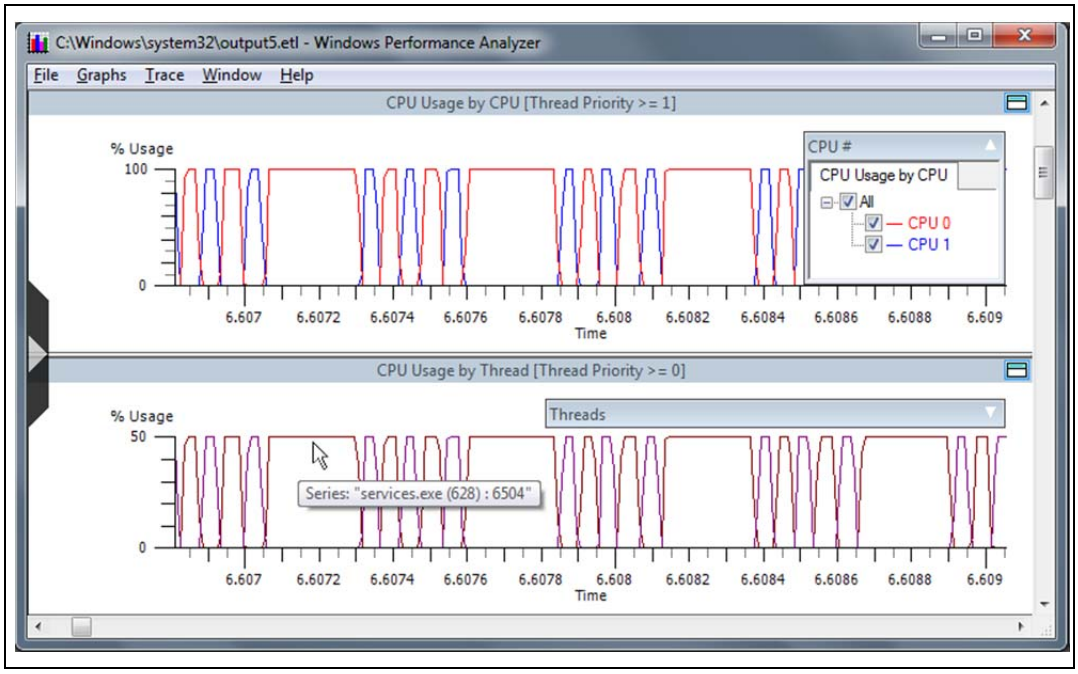
high (20~80%, but not 100%) CPU utilization on multiple CPUs like the example shown below.

Figure 27: Xperf – Occasional High CPU C-state Transition



12. Zoom in "CPU Usage by CPU" graph until each C-state transition can be seen separately (in other words, until CPU utilization of each CPU toggles between 0% and 100%) like shown in the example below.

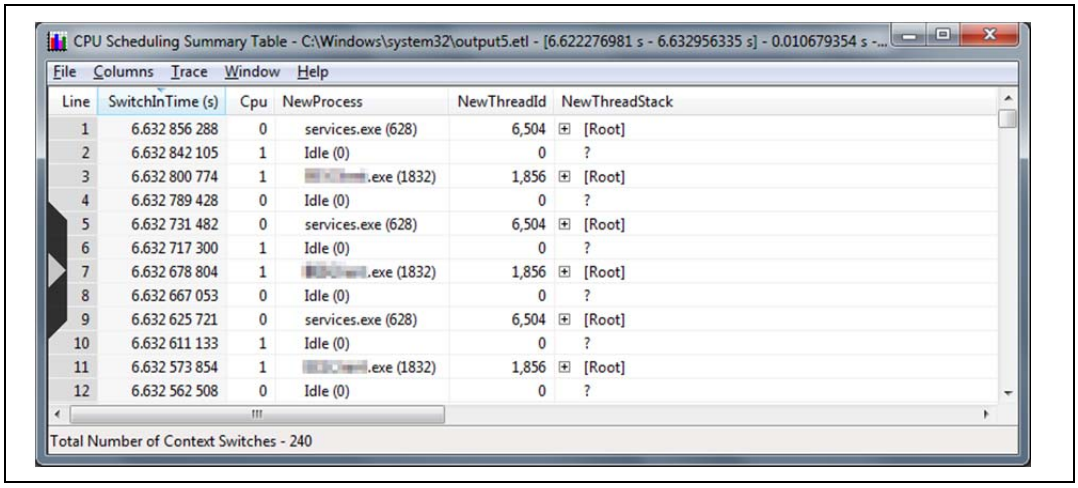
Figure 28: Xperf – IPI Storm Example



By comparing the "CPU Usage by CPU" and the "CPU Usage by Thread" graphs, each CPU's activity will be completely synchronized with the thread's activity. Which process is causing the problem can easily be determined by hovering the mouse cursor on each thread's line. In this example, one of the threads belongs to services.exe (shown by the brown line), and the other thread belongs to a 3rd party application which was shown as process #13 in Figure 23 (shown by purple line).

13. Next step is to determine what kind of code sequence is causing the C-state transition. To do so, scroll down to the "CPU Scheduling" graph, and select several cycles of the transitions, right click on the graph, and select "Summary table".
14. Arrange the columns in the following order (left to right)
    a. SwitchInTime (s)
    b. CPU
    c. NewProcess
    d. NewThreadId
    e. NewThreadStack
15. Click on the "SwitchInTimes (s)" column and sort the entries in ascending order.

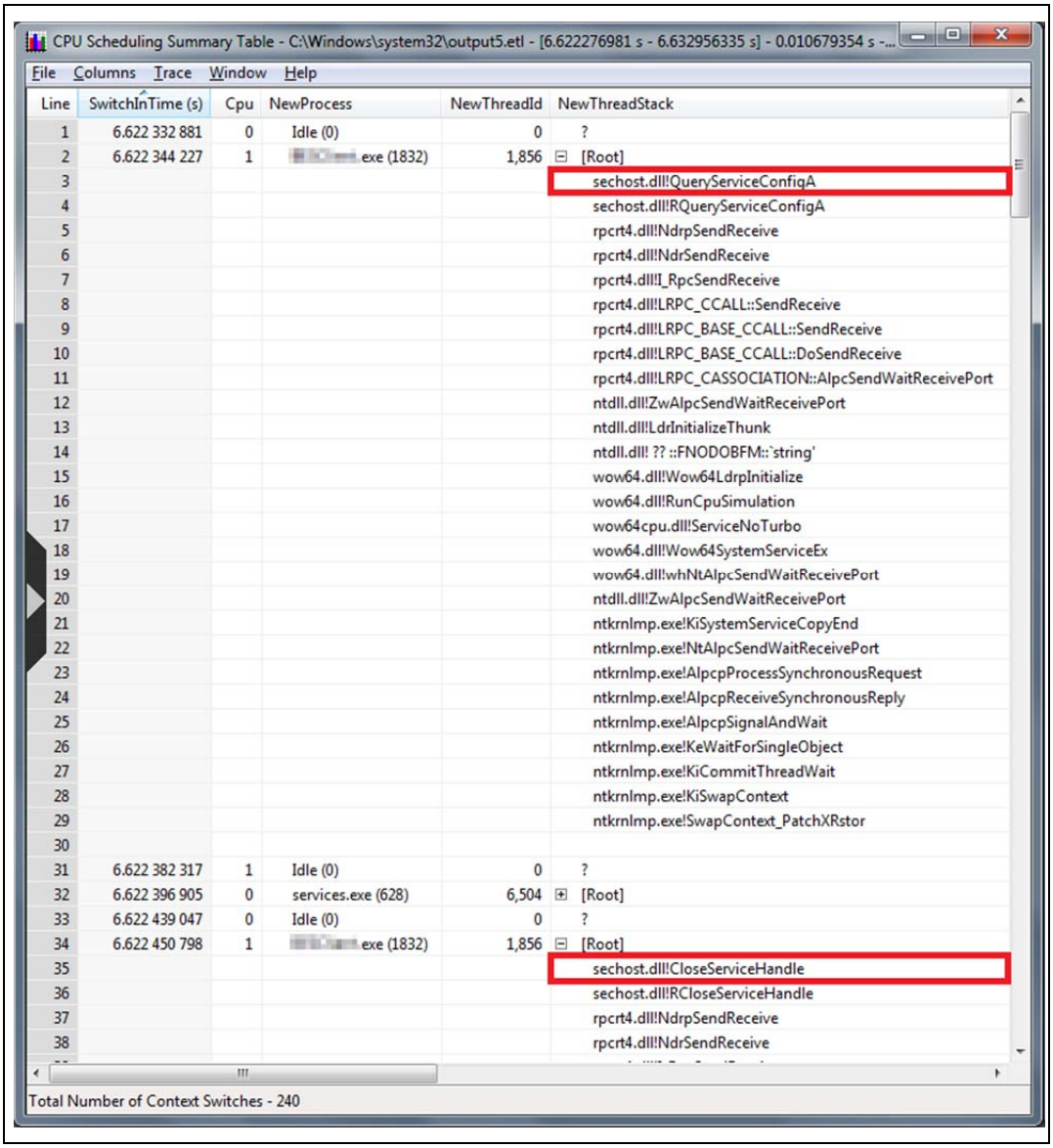
Figure 29: Xperf – IPI Storm Detail



In this example, services.exe (Process ID: 628, Thread ID: 6504) and the other thread (Process ID: 1832, Thread ID: 1856) are running alternately running on CPU0 and CPU1 respectively.

16. Expand "NewThreadStack" at each context switch to the active thread and find out which API call caused the thread to enter wait state.

Figure 30: Xperf – IPI Storm Stack Dump



In the example shown above in Figure 30, Thread ID 1856 was blocked when the application called QueryServiceConfig() and CloseServiceHandle(). When these Win32 APIs are called from the application, they internally communicate with services.exe through RPC (Remote Procedure Call), and while it's waiting for the response from services.exe, CPU1 enters a lower C state. When services.exe is ready with the result, it sends an IPI (Inter Processor Interrupt) back to CPU1 and brings the thread back to active state. In return, services.exe finished with the RPC call and the CPU enters a C state until next RPC call comes in.

Apparently, this application program enumerates all services in the system and repeats OpenServiceHandle() (not shown in the screen shot), QueryServiceConfig(),

and CloseServiceHandle() API calls for each service. This causes thousands of C state transitions on CPU0 and CPU1.

3.3 I/O Issues

Even if an application consumes a very small amount of CPU cycles and a few C-state transitions, the software can still have a large impact to the platform battery life by increasing the I/O subsystem power consumption. This section discusses how to identify I/O activities for each device class.

3.3.1 Disk I/O Issues

The tool most useful for disk I/O analysis is Process Monitor (aka. ProcMon) from Sysinternals. Please see Chapter 6 "References" for the availability of this tool.

When analyzing file/registry access, the following aspects of the file system activity and registry activity need to be checked:

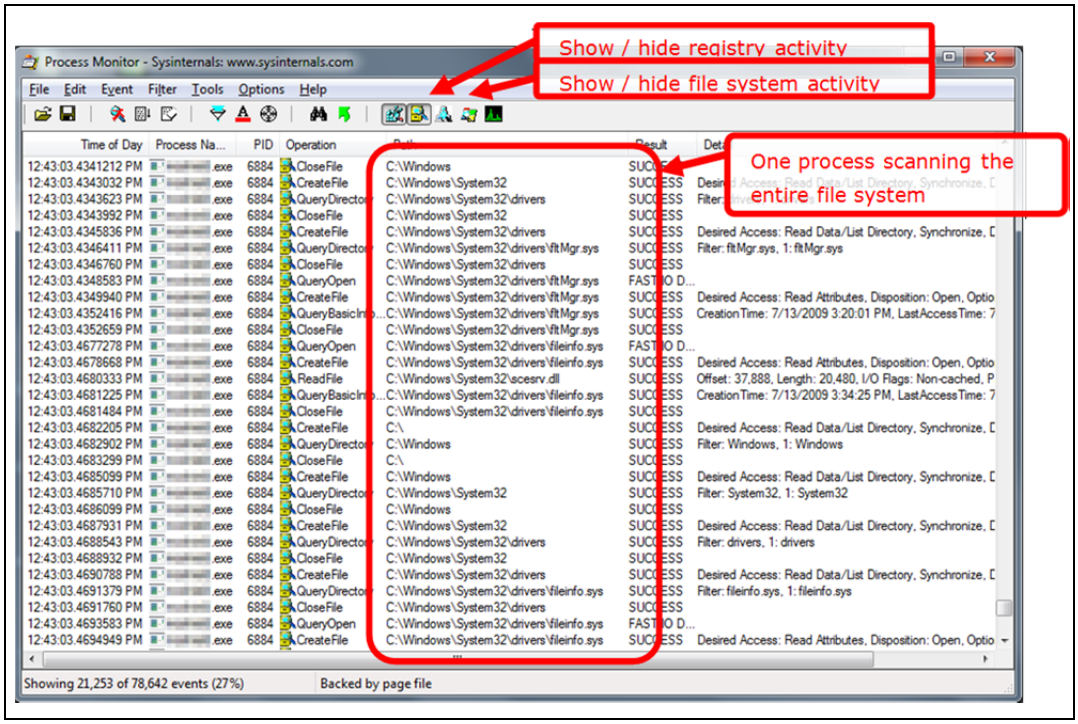
1. Activity type

   Most disk I/O activities are caused by an application program's file access or registry access. Generally speaking, repeated read access to the same file or registry entry doesn't cause a significant impact because the data is cached. If different files or registry entries are accessed (for example scanning all files in the file system) or write access happens, this will increase the activity of the disk drive and the link to the disk drive.

2. Activity timing

   As shown in Table 1, a HDD has multiple power saving states, but they take certain duration of inactivity before those states are enabled. So, the same rule illustrated in Figure 21 applies to disk access. Time stamps of file/registry accesses that potentially causes disk accesses (as described above) have to be checked. For example, if a read request to different files is happening every 1 sec, it will most likely keep the drive in the active idle or performance idle state and loose the opportunity for the HDD to enter the Low Power Idle or Standby (spin-down) state.

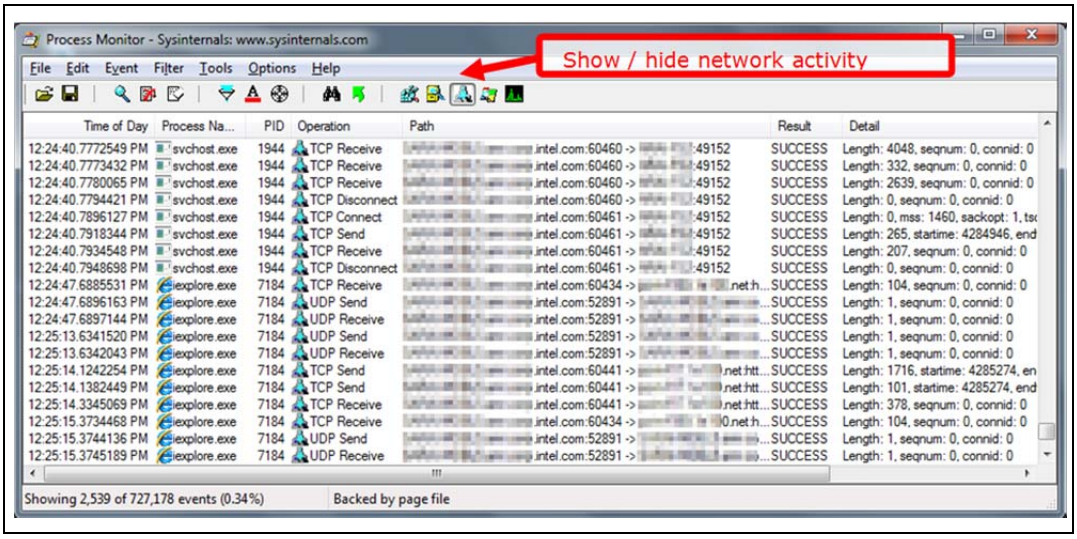
Figure 31: ProcMon -- File System and Registry Activity



3.3.2 Network I/O Issues

Network I/O activities can also be monitored with Process Monitor. Which process is sending/receiving packets and the destination/source of those packets can be determined from the output of the program. Using this information can help to determine whether these activities are necessary or not.

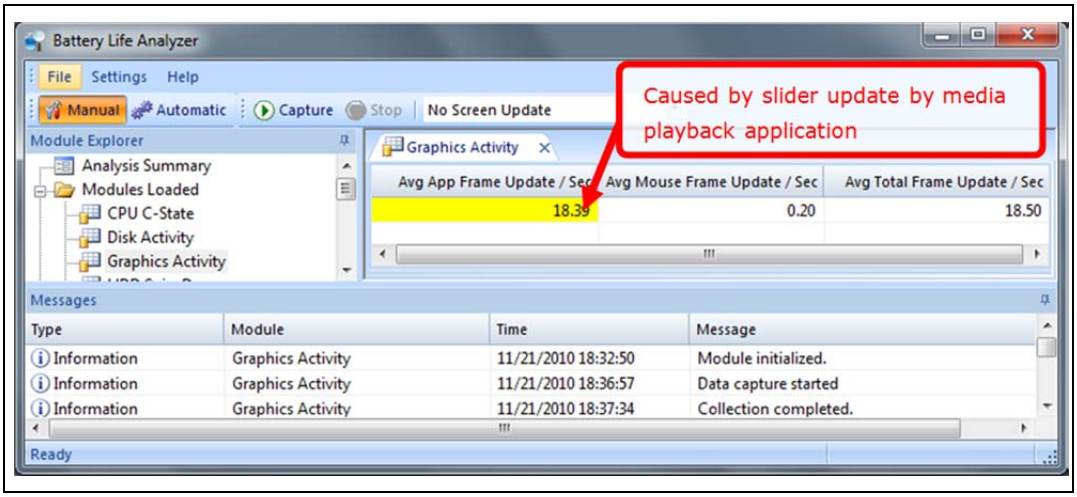
Figure 32: ProcMon -- Network Activity



3.3.3 Graphics I/O Issues

By default, Windows (after Vista SP1) suppresses VBI (vertical blank interrupt) generation of the graphics adapter after 10 VBI cycles with no graphics activity. To take advantage of this power saving feature, the number of frames updated – even the number of pixels updated in each frame is very small – should be minimized. Battery Life Analyzer shows the number of frames updated as illustrated in Figure 33 below.

Figure 33: Battery Life Analyzer - Graphics Activity Analysis



Please note that graphics activity happens every time the application program tries to overwrite the screen with same image. In the screen shot above, the media playback program is calling an API to update the slider (which shows the current playback

position) very frequently. But, only a few of the API calls make the image change. Other calls only overwrite the same slider image.

3.3.4 USB I/O Issues

USB (up to USB 2.0) is a polling based interface and even when there is no meaningful data transfer to/from a device, the host controller has to constantly poll the devices. This activity keeps a large portion of the platform in an active state. To minimize the power impact of the USB subsystem, good behaving devices implement the Selective Suspend state. A device enters the selective suspend state when it is not used or it is in a long idle state. Once all downstream devices are in the selective suspend state, the USB hub can enter selective suspend. Similarly, the host controller can enter selective suspend after all downstream devices enter selective suspend and the USB subsystem become inactive. For more details about the Selective Suspend state implementation, please refer to the Microsoft whitepaper on the topic. Link to the white paper can be found in Chapter 6 "References".

If software accesses USB devices frequently (once in a minute or more), it will significantly reduce the opportunity for the USB host controller to enter the selective suspend state.

Battery Life Analyzer's USB analysis feature measures the duration of selective suspend for each host controller, hub, and attached devices. To verify USB devices are in the selective suspend state, please start looking at host controller. If the host controller's selective suspend duration is high enough (>90%), there is no problem with the devices under that host controller. Otherwise, find the device showing low selective suspend duration under that host controller.

Once the device that is causing the problem is identified, check the software associated with that device.

3.3.5 Frequent Snoop Cycle Issue

Modern devices, including all devices discussed above, transfer data to memory using bus master transfer. This can happen at any time if the CPU is in a sleep state. If the cache memory is still active (i.e., CPU package C-state is in C3 or C6), it will cause the CPU package to wake up (to CPU package C1 or C2 state), snooping the cache memory, and can cause significant platform power increase if this situation happens frequently. As discussed in Section 3.2.1 "CPU C-State Residency", this can be observed as high CPU package C1 or C2 state residency.

If high CPU package C1 or C2 state residency is seen, the next step is to determine which device is causing the problem. The easiest way is to remove potential devices one by one and see if it affects the package C state residency. If it is not possible to physically remove a device, the next best thing is to disable it in the Microsoft Windows Device Manager.

4 Fixing Idle Power Issues

4.1 Minimize Unnecessary Activities

Once the behavior of each activity is understood, determining which activities are really essential can begin. For example, if a media playback application is scanning the entire file system repeatedly, it's probably looking for new media files. This is not a desired activity while the system is operating on battery power. The discussion about what kind of activities are appropriate (or not appropriate) while a system is on battery power is beyond the scope of this document, but this section will introduce Microsoft Windows features to help minimize unnecessary activities.

4.1.1 Task Scheduler

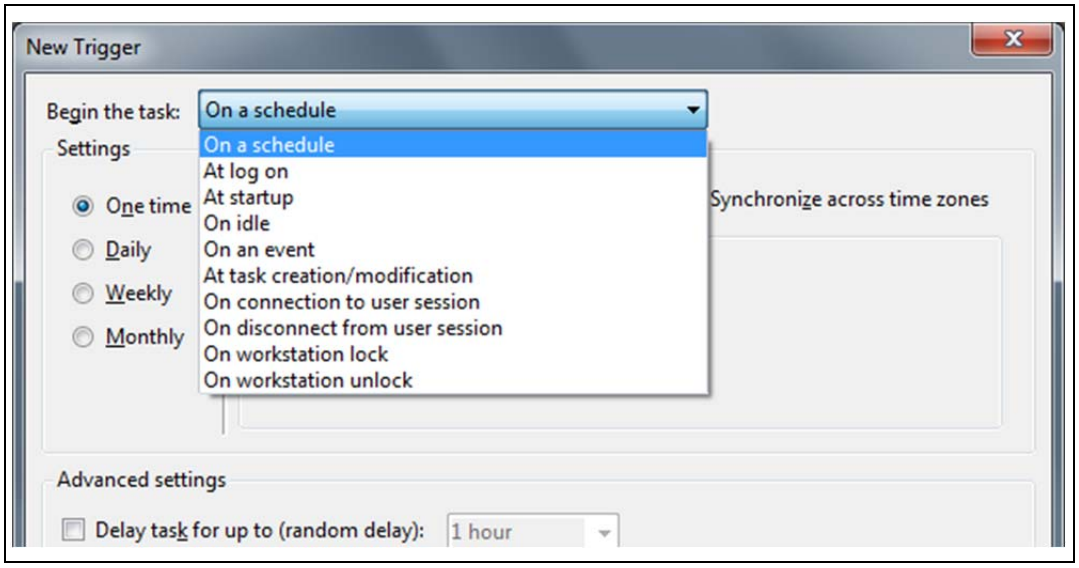
When in the process of minimizing an application's need to poll, the Windows Task Scheduler is a good first step. The Task Scheduler enables a user to automatically perform routine tasks with various triggers and conditions.

A new task can be created through the GUI, command line , or API. Some of the useful triggers and conditions that can be used to minimize unnecessary activities during battery option are:

[Triggers]

- On a schedule – Start a task at a specific time.
- On idle – Start a task when the system is idle.
- On an event – Start a task when an Windows Event Log action occurs.

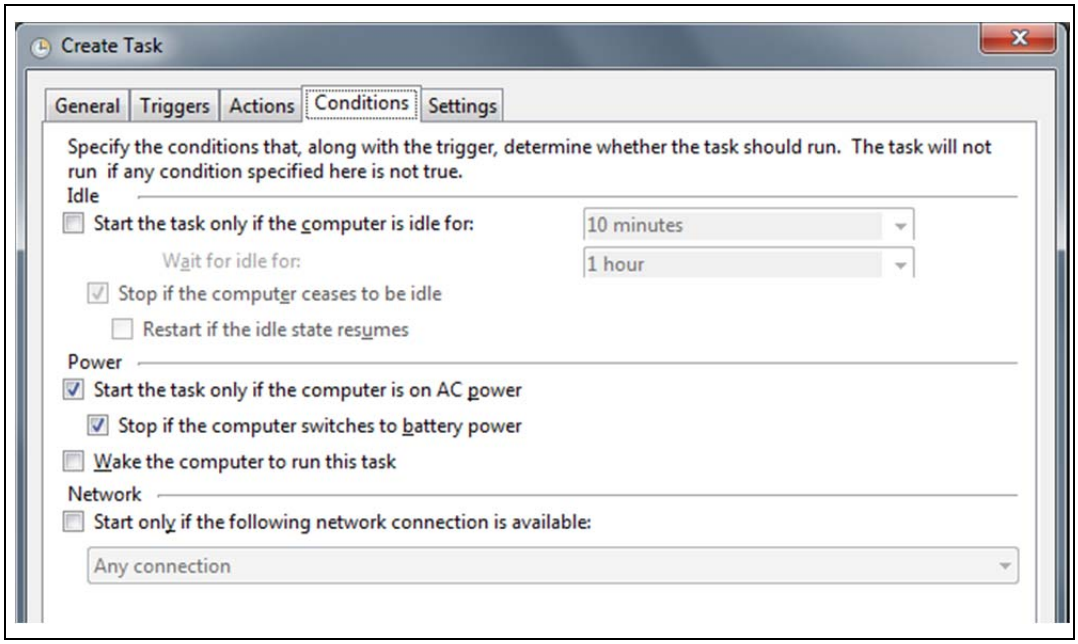
Figure 34: Task Scheduler GUI – Trigger



[Conditions]

- Start the task only if the computer is idle for – Select a certain idle duration before starting the task, and stop the task when the computer ceases to be idle.
- Start the task only if the computer is on AC power - This limits the task from starting only while the computer is on AC power. It is also possible to stop the task when the system is switched to battery power.
- Start the task only if the following network connection is available – Limit the task to be run only when on specific network connections.

Figure 35: Task Scheduler GUI – Conditions



The Task Scheduler is the feature to use when a specific task needs to be started at an event. Please see Chapter 6 "References" for the links to the detailed information.

4.1.2 Event Callback APIs

By making a long running program (such as a GUI program, or service) event driven, the process of regular polling can avoided. This will cut down on unnecessary CPU utilization and CPU C state transitions. The following table provides a list of APIs to replace commonly required polling points by applications.

Table 2: Useful Event Callback APIs

Event	API to setup notification	Event notification	OS Support
File Change	FindFirstChangeNotification()	Change notification handle	Microsoft Windows 2000*
Registry Change	RegNotifyChangeKeyValue()	Blocks until the change is detected Change notification handle	Microsoft Windows 2000
Power Status Change	RegisterPowerSettingNotification() Note: Some messages to GUI application are enabled by default	[GUI Application] Window message (WM_POWERBROADCAST, PBT_POWERSTATUSCHANGE) [Service] Callback	Microsoft Windows 2000

Event	API to setup notification	Event notification	OS Support
Power Profile Change	RegisterPowerSettingNotification()	[GUI Application] Window message (WM_POWERBROADCAST PBT_POWERSETTINGCHANGE) [Service] Callback	Microsoft Vista*
Windows Event Log	EvtSubscribe()	Callback	Microsoft Vista
Network	NotifyIpInterfaceChange()	Callback	Microsoft Vista
PnP Events	RegisterDeviceNotification() Note: Some events to GUI application are enabled by default	[GUI Application] Window message (WM_DEVICECHANGE DBT_DEVICExxx) [Service] Callback	Microsoft Windows 2000

4.2 Optimize Timer Resolution

4.2.1 Timer Resolution for Media Application

Media application programs often change the timer resolution to avoid glitches during media playback. While necessary in earlier versions of Microsoft Windows, in Windows 7 the Audio Engine (OS built in component) adjusts the timer resolution based on the capability of the endpoint device and other factors. Unless the application has a specific timer resolution requirement, media playback programs have no need to change the OS timer resolution.

Figure 36 Code Sample for Timer Resolution Change

```
DWORD dwVersion = GetVersion();
/* Check OS version.
 *   Windows 7 returns major version = 6, minor version = 1
 */
if ((dwVersion & 0xF) << 4 | (dwVersion >> 4) & 0xF < 0x61) {
       timeBeginPeriod(1);
}
```

For older versions of Microsoft Windows, applications may need to change the timer resolution, but should do so only when it is necessary. For example, media playback applications should change the timer resolution when media playback starts not at application startup. Timer resolution should be restored when playback stops or pauses rather than when the application terminates.

The Windows 7 Audio Engine changes the timer resolution automatically when the audio session is established, keeping that timer resolution until the session is closed. To minimize the duration of timer resolution changes, an application program needs to establish an audio session only when audio devices are being used. An application should close the session explicitly when no audio activity is happening. Table 3 shows the APIs that cause timer resolution changes and restores for major audio API sets.

Table 3 APIs for Audio Session Control

API Set	API changes timer resolution	API restores timer resolution
Core Audio	IAudioClient::Initialize()	IAudioClient::Release()
Windows Multimedia	waveOutOpen()	waveOutClose()
Direct Sound	[multiple APIs triggers change]	IDirectSound::Release() IDirectSound8::Release()

4.2.2 Timer Resolution for Other Application

Other than media applications, there are very few reasons for applications to change the timer resolution. Many application programs often change the timer resolution to get a fine grain time stamping with timer tick based APIs including GetSystemTime(), GetTickCount(), GetSystemTimeAsFileTime(), timeGetTime(), and timeGetSystemTime(). This method of time stamping causes too large an impact to the platform (as shown in Figure 12), while only providing up to a 1 msec resolution. It is highly recommended to use a high resolution timer API such as QueryPerformanceCounter() instead, which provides a much better resolution without any power impact.

Some other application programs change the timer resolution hoping that will increase their performance. This method might have benefited older platforms, but it doesn't have any positive impact to modern platforms. Instead, excessively frequent timer tick interrupts might cause a negative impact to performance.

4.3 Reducing Periodic Activities

As explained in Section 2.3.1 "Frequency and Duration of CPU Utilization", frequent application program activity causes a significant increase in the platform power consumption. In many cases, frequent activities are caused by the excessive use of the periodic timer[2] by applications. To reduce periodic timer activity, please consider:

- If an activity is always necessary

[2] Please note that periodic timer event is different from timer tick, which was discussed in the previous section.

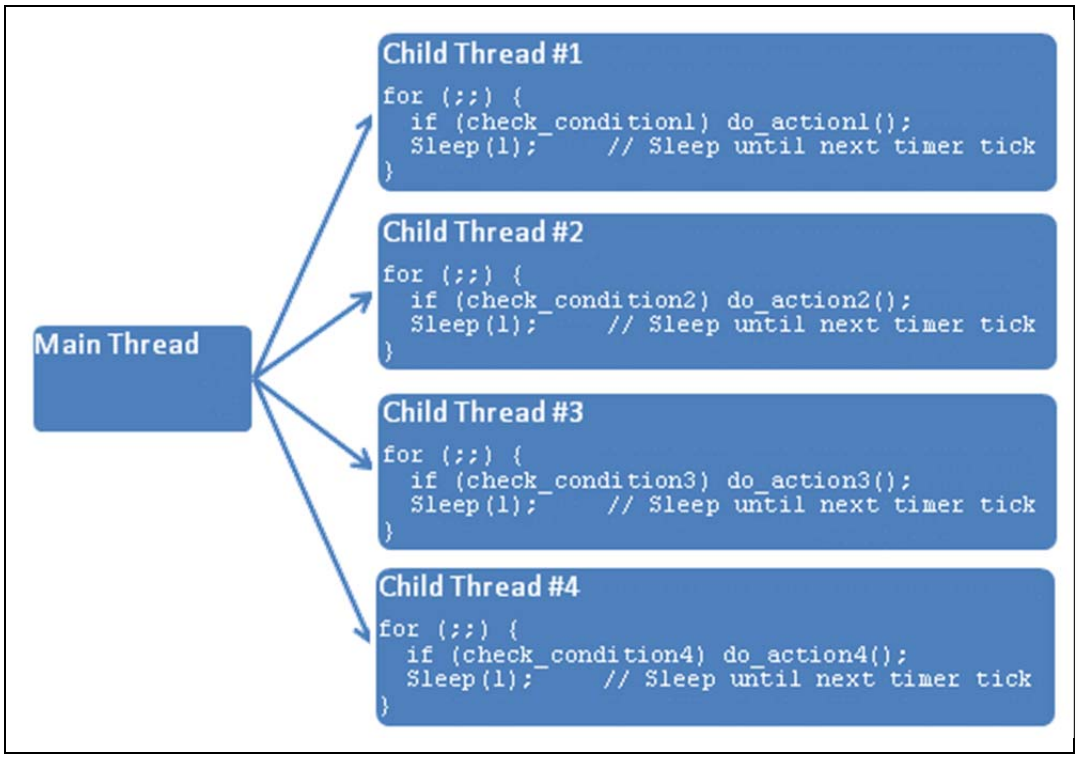
- If an activity is only required at certain times, for example when network connection is available, avoid causing the activity all other times.

- If polling-based code can be replaced by event-driven code
  - Please consider using the event callback APIs explained in Section 4.1.2 "Event Callback APIs".

- Minimizing the frequency of periodic activity
  - Avoid periodic activities with 100msec or shorter periods unless they are absolutely necessary.

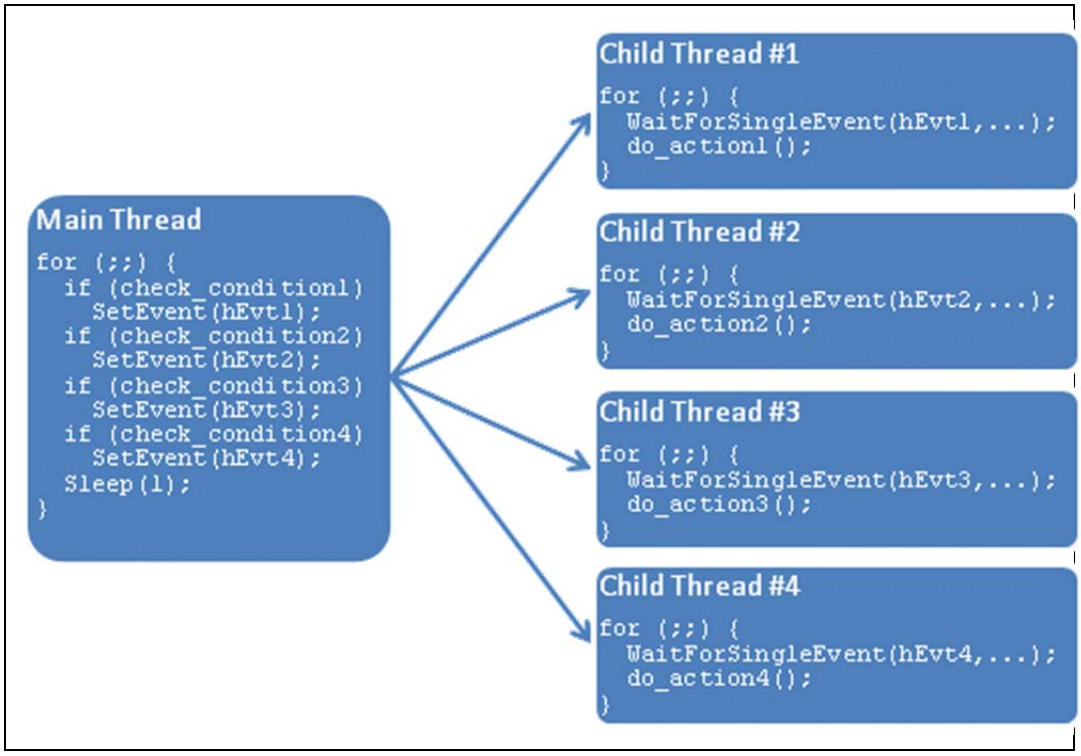
- Coalescing activity within the process
  - Figure 38 illustrates a bad example of a multi-threaded application. In this example, each thread executes an activity every timer tick. These threads are most likely assigned to different CPU and cause multiple CPUs to wake up per timer tick.

Figure 37: Bad Example - Periodic Activity in Multi-threaded Application



The impact of this program can be reduced with a small change illustrated in Figure 39. In this code, until a specific condition happens, the Main Thread is the only activity this program generates, waking only one CPU at every timer event.

Figure 38: Better Example - Periodic Activity in Single Thread



- Coalescing activity between processes

  Total system C-state transitions can be reduced by coalescing the application's activity with other software activities using the Coalescing timer API – SetWaitableTimerEx() as shown in Figure 40. The TolerableDelay parameter instructs the operating system how much delay the application can tolerate. The operating system coalesces multiple timer events within the tolerable delay. Values of 100 msec or greater are recommended to achieve the best possible power saving.

Figure 39: Timer Coalescing API Definition

```
BOOL SetWaitableTimerEx(
    __in     HANDLE hTimer,
    __in     const LARGE_INTEGER *lpDueTime,
    __in     LONG lPeriod,
    __in_opt PTIMERAPCROUTINE pfnCompletionRoutine,
    __in_opt LPVOID lpArgToCompletionRoutine,
    __in_opt PREASON_CONTEXT WakeContext,
    __in     ULONG TolerableDelay          // in msec
    );
```

Figure 41 code sample illustrates how to use this timer coalescing API to replace a simple Sleep() API.

Note: Error checking is omitted to simplify the code

Figure 40: Timer Coalescing API Usage Example

```
typedef BOOL (*PFNSETWAITABLETIMEREX)(
    __in     HANDLE hTimer,
    __in     const LARGE_INTEGER *lpDueTime,
    __in     LONG lPeriod,
    __in_opt PTIMERAPCROUTINE pfnCompletionRoutine,
    __in_opt LPVOID lpArgToCompletionRoutine,
    __in_opt PREASON_CONTEXT WakeContext,
    __in     ULONG TolerableDelay
);

VOID SleepWithDelayTolerance(
   __in DWORD dwMilliseconds,
   __in ULONG TolerableDelay)
{
   HMODULE hMod;
   static PFNSETWAITABLETIMEREX pfnSWTEX;
   HANDLE hTimer;
   LARGE_INTEGER liDueTime;

   hMod = LoadLibrary("kernel32.dll");
   pfnSWTEX = GetProcAddress(hMod, "SetWaitableTimerEx");

   if (pfnSWTEX == NULL) {    // If the OS doesn't support new API
          Sleep(dwMilliseconds);
          return;
   }
   liDueTime.QuadPart = dwMilliseconds * -10000;
   hTimer = CrateWaitableTimerEx(NULL, NULL, 0, TIMER_MODIFY_STATE);
   (*pfnSWTEX)(hTimer,
               &liDueTime,
               0,      // LONG lPeriod
               NULL,   // PTIMERAPCROUTINE pfnCompletionRoutine
               NULL,   // LPVOID lpArgToCompletionRoutine
               NULL,   // PREASON_CONTEXT  WakeContext
               TolerableDelay);
   WaitForSingleObject(hTimer, INFINITE);
   CloseHandle(hTimer);
}
```

For more details about the timer coalescing API, including the DDI for kernel mode code, please refer to "Windows Timer Coalescing" white papers listed in Chapter 6 "References".

4.4 Reducing IPI

An IPI (inter-processor interrupt) is used to implement inter-process communication. As explained in Section 3.2.4.2.2 "Debugging Occasional High C-State Transition Issue", some Win32 APIs internally use RPCs (remote procedure calls) and the requests are processed by the server process and cause extra C-state transitions. The following is a list of some of the Win32 APIs that cause IPIs:

- Service Control Manager (SCM) APIs
  - OpenSCManager()
  - OpenService()
  - QueryService*XXX*()
  - CloseService()
  - … etc.
- Windows Events, Windows Event Log APIs
  - EvtXXX()
  - OpenEventLog()
  - ReadEventLog()
  - CloseEventLog()
  - … etc.
- COM based APIs, when the server is implemented as an EXE
  - WMI
  - … etc.

Alternative APIs can be used instead of the above APIs to reduce CPU utilization and IPI generation. For example, many WMI queries can be replaced with simpler Win32 APIs, and the Windows Event Log APIs in polling-based code can replaced with callbacks implemented using the EvtSubscribe() API.

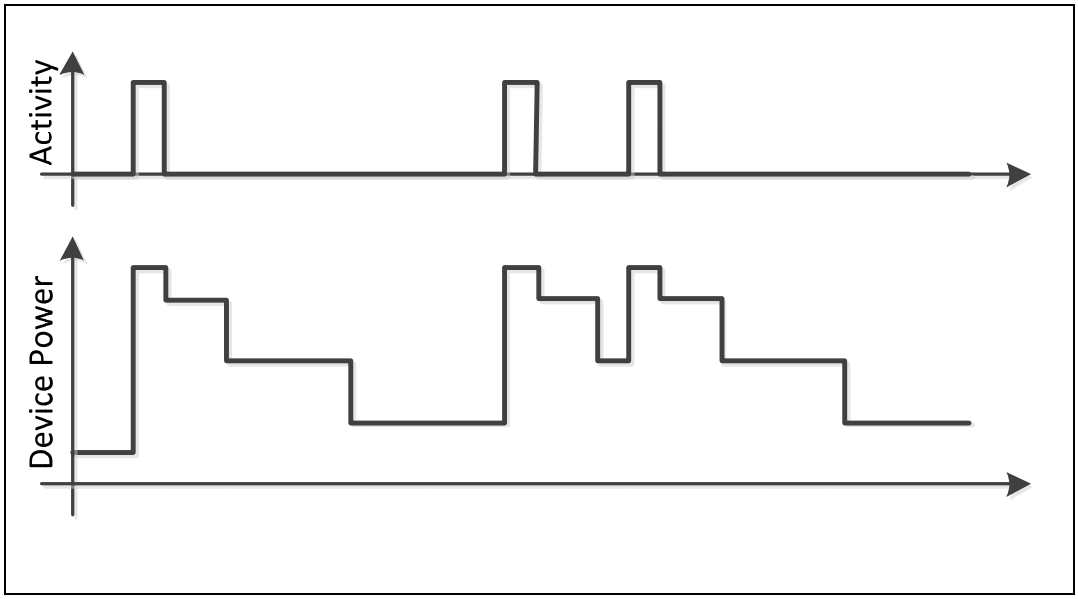
If the above APIs are used, it is recommended to make those calls in short durations rather than spreading them over time. By doing so, excessive C-state transitions occur in a short time period followed by a relatively long idle period. This will allow the platform to do a better job of power management.

4.5 Reducing I/O Activity Impact

Many I/O devices and the interface links to the devices implement idle timer based power saving mechanisms and power saving states that are entered only after certain durations of idleness.

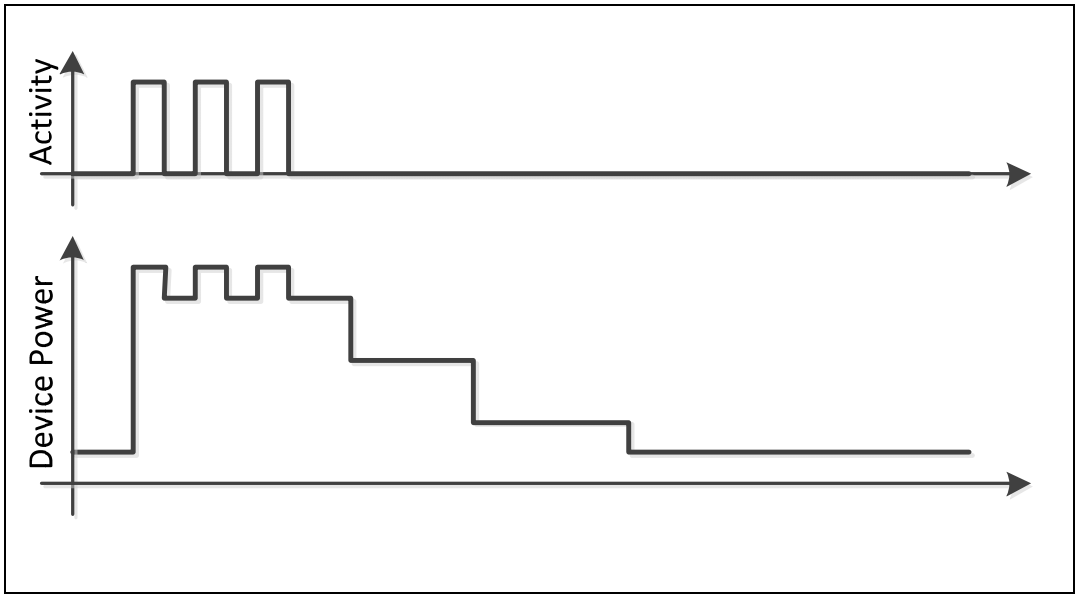
Figure 42 illustrates how the power states change over time with scattered I/O activities.

Figure 41: Device Power Consumption with Scattered Activity



When the activity happens at once and it is followed by an uninterrupted long idle duration, the device can stay in a lower power state for longer duration as shown below.

Figure 42: Device Power Consumption with Coalesced Activity



It is often seen that background processes intentionally throttle their performance (and spread their activity over a long duration) hoping that will minimize the performance impact to other applications. Such behavior has a negative impact to the system's battery life. The recommended solution is to use Task Scheduler and use system idleness as a trigger or condition.

This recommendation applies to most device classes, including the CPU. The following sections discuss device class specific recommendations.

4.5.1 Reducing Disk I/O Activity

Windows will flush the write data to disks soon after an application program calls the Win32 API for write access, such as WriteFile(). It holds dirty data in memory for few seconds to consolidate with other write operations, but doesn't keep it in memory for more than 10 seconds. If the application program doesn't have a strict requirement about when disk writes must happen, please try to buffer write data in the program. For example, if the C library standard I/O functions such as fwrite(), fputs(), etc. are used, write data is automatically buffered until the fflush() function is called explicitly or the I/O stream is closed by the fclose() function.

4.5.2 Reducing Graphics I/O Activity

As explained in Section 3.3.3 "Graphics I/O Issues", frequent graphics activity causes a platform level power increase even if there is no visible graphics change. It is recommended to:

- Minimize the use of animation
  - If animation is necessary to catch a user's attention, animating for a limited duration may be enough.
- Minimize the frequency of graphics updates
  - For example, if a media playback application adjusts its slider position to match the current playback position, only update the slider position a few times per second.
- Consolidate graphics updates within program
  - If the program needs to make multiple periodic graphics updates, they should be aligned with each other to reduce the number of frames updated.

5 Conclusion

The world is moving toward 'Green technologies' and consumer demand for 'Extended Battery Life' is always increasing. The need for higher performance and new usage models will also keep increasing as they have done in the past couple of decades. Energy-Efficiency will be crucial for the computing industry in the future both to increase battery life for mobile platforms and to reduce energy expenses for desktop and server platforms. Software behavior can have a significant effect on platform power consumption and battery life.

In typical usage models, the mobile platform is idle (as measured by CPU C0 residency) for about 90-95% of the time. It is important to reduce power consumption for idle and semi-idle workloads. Energy-efficient applications when idle should have a minimal impact on platform power consumption. Frequent background activity should be avoided.

During active workload execution, applications and services should improve computation efficiency, maximize multi-threaded execution and coalesce activity to increase idle residency. This will allow the platform to go into deeper low power states, reduce C-state transitions, and thereby amortizing the power cost of transitioning the platform into and out of low power states.

6 References

6.1 Tools

- Intel Battery Life Analyzer requests, questions and feedbacks
  mailto:BatteryLifeAnalyzer@intel.com
- Intel® VTune™ Amplifier XE
  http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/
- Microsoft Windows Performance Toolkit (included in Windows SDK)
  http://msdn.microsoft.com/en-us/windows/bb980924
- Sysinternals Process Monitor
  http://technet.microsoft.com/en-us/sysinternals

6.2 Documents

- "Energy-Efficient Platforms: Designing Devices Using the New Power Management Extensions for Interconnects"
  http://www.intel.com/technology/mobility/notebooks.htm
- "Creating Energy-Efficient Software"
  http://software.intel.com/en-us/articles/creating-energy-efficient-software-part-1/
- "Maximizing Power Savings on Mobile Platforms"
  http://software.intel.com/en-us/articles/maximizing-power-savings-on-mobile-platforms/
- "Mobile Battery Life Solutions for Windows 7"
  http://download.microsoft.com/download/7/E/7/7E7662CF-CBEA-470B-A97E-CE7CE0D98DC2/mobile_bat_Win7.docx
- Task Scheduler
  http://msdn.microsoft.com/en-us/library/aa383614(VS.85).aspx
- Windows Timer Coalescing
  http://www.microsoft.com/whdc/system/pnppwr/powermgmt/TimerCoal.mspx
- Selective Suspend in USB Drivers
  http://www.microsoft.com/whdc/driver/wdf/USB_select-susp.mspx
- Microsoft Windows Performance Toolkit
  http://msdn.microsoft.com/en-us/performance/cc825801.aspx