x86混合体系结构的软件优化

白皮书

2021年10月

修订**1.0**

文件编号:348851-001US

注意：本文档包含产品在开发设计阶段的信息。此处信息如有更改，恕不另行通知。不要使用此信息完成设计。

性能因使用、配置和其他因素而异。在www.intel.com/performanceindex了解更多信息。

性能结果基于截至配置中显示的日期的测试，可能不会反映所有公开可用的更新。有关配置详细信息，请参阅备份。任何产品或组件都不可能是绝对安全的。

所有产品计划和路线图如有更改，恕不另行通知。

Intel使用代号来标识正在开发且不可公开使用的产品、技术或服务。这些不是"商业"名称，也不打算作为商标使用。

您的成本和结果可能会有所不同。

Intel技术可能需要启用硬件、软件或服务激活。

您不得在涉及本文所述英特尔产品的任何侵权行为或其他法律分析时使用或便利使用本文档。您同意授予英特尔一个非排他性的，免版税的许可，任何专利权利要求，包括在此披露的主题。

本文件不授予任何知识产权许可（明示或暗示、禁止反悔或其他方式）。所描述的产品可能包含设计缺陷或错误，称为勘误表，这可能导致产品偏离公布的规范。可根据要求提供当前特征勘误表。

所描述的产品可能包含设计缺陷或错误，称为勘误表，这可能导致产品偏离公布的规范。可根据要求提供当前特征勘误表。

Intel否认所有明示和默示保证，包括但不限于适销性、适合特定目的和不侵权的默示保证，以及任何因履行过程、交易过程或贸易中的使用而产生的保证。

Intel does not control or audit third-party benchmark data or the web sites referenced in this document. You should visit the referenced web site and confirm whether referenced data are accurate.

Contents

1 Overview of x86 Hybrid Architecture 8

1.1 12th Generation Intel® Core™ Processors Supporting Performance Hybrid Architecture 8

1.2 11th Generation Intel® Core™ Processors Supporting Hybrid Architecture8

2 Hybrid Scheduling 10

2.1 Hardware Guided Scheduling....10

2.2 Intel® Thread Director....10

2.3 Scheduling with Intel® Hyper-Threading Technology Enabled on Processors Supporting x86 Hybrid Architecture 11

2.4 Scheduling with a Multi-E-Core Module 11

2.5 Scheduling Background Threads on x86 Hybrid Architecture 11

3 Key Windows Power and Performance Features 12

3.1 Thread Scheduling Overview 12

3.2 Windows Core Parking Engine Overview....12

3.3 Performance State Control Engine....13

4 12th Generation Intel® Core™ Processor Windows Scheduling/Parking Examples 14

4.1 Single Thread Scenario....14

4.2 Limited Thread Scenario Example 1....14

4.3 Limited Thread Scenario Example 2....15

4.4 Multithread Scenario 15

4.5 Background Threads 16

4.6 Multimedia Threads 16

4.7 Eco QoS Threads....17

4.8 Low Utilization Threads 17

4.9 Events tracing for windows 18

4.9.1 Is Intel® Thread Director enabled on the system?....18

4.9.2 How can I check the QoS of the process/thread? 18

5 11th Generation Intel® Core™ Processor Windows Scheduling/Parking Examples 19

6 Software Application: Common Questions and Recommendations for Intel® Core™ Processors Supporting x86 Hybrid Architecture 20

6.1 What are the ISA differences visible to software? 20
6.2 Can affinity be used on processors supporting x86 hybrid architecture? 20
6.3 What are Windows power throttling APIs? How can application developers influence hybrid scheduling? 20
6.4 Number of Threads 22
6.5 Active Spins 22
6.6 Driver developers can contribute towards optimal interrupt steering? 22
6.6.1 Do Not Configure Interrupt Policy 22
6.6.2 Architectural Relationship 22
7 Hybrid Key PPM Settings and Setting Values 23
7.1 Windows Power Management Settings 23
7.2 Core Parking Engine Settings 23
7.2.1 HeteroPolicy 23
7.2.2 Checking Parking State of a Processor 25
7.3 Performance State Settings 26
7.3.1 PerfEnergyPreference 26
7.3.2 PerfEnergyPreference1 26
7.3.3 MinimumPerformance 26
7.3.4 MinimumPerformance1 26
7.3.5 MaximumPerformance 26
7.3.6 MaximumPerformance1 27
7.4 Windows Performance Power Slider 27
7.5 Key Power Profiles 28
7.5.1 Default Profile 28
7.5.2 Low Latency 28
7.5.3 Low Power 28
7.5.4 Screen Off 28
7.6 QoS, HWP, and Hybrid Scheduling 28
7.6.1 Default 28
7.6.2 Occluded Window 28
7.6.3 Background Threads 28
7.6.4 Multimedia Threads 29
7.6.5 Eco Threads 29

Revision History

Revision Number	Description	Date
1.0	• Initial release of the document.	October 2021

INTRODUCTION

This technical document provides information on optimizing software for Intel® Core™ processors that support hybrid architecture.

The document provides an overview of x86 hybrid architecture, hybrid core usage with Windows, and provides details on how software applications and drivers can ensure optimal core usage.

Key Windows Processor Power Management Settings (PPM Settings) that can be used on Intel Core processors that support x86 hybrid architecture to meet system performance vs. power goals are also described.

1 Overview of x86 Hybrid Architecture

1.1 12th Generation Intel® Core™ Processors Supporting Performance Hybrid Architecture

12th Generation Intel Core processors supporting performance hybrid architecture consist of up to eight Performance cores (P-cores) and eight Efficient cores (E-cores). These processors also include a 3MB Last Level Cache (LLC) per IDI module, where a module is one P-core or four E-cores. It has symmetrical ISA and comes in variety of configurations.

P-cores provide single or limited thread performance, while E-cores help provide improved scaling and multithreaded efficiency. P-cores on these processors can also have Intel® Hyper-Threading Technology enabled. All cores can be active simultaneously when the OS decides to schedule on all processors.

A key OSV requirement for enabling hybrid is symmetric ISA across different core types in a performance hybrid architecture. In 12th Generation Intel Core processors supporting performance hybrid architecture, ISA is converged to a common baseline between the P-Cores and E-Cores.

1.2 11th Generation Intel® Core™ Processors Supporting Hybrid Architecture

11th Generation Intel Core processors supporting hybrid architecture consist of one P-core and a four E-core cluster connected to CCF with a shared 4MB LLC. The E-core module and P-core are connected over two IDI channels to CCF and shared 4MB LLC. The LLC is inclusive, and all cores are fully coherent.

The P-core is a single physical core with Intel Hyper-Threading Technology disabled. The OS enumerates and sees five physical cores. All cores can be active simultaneously when the OS decides to schedule on all five cores.

A key OSV requirement for enabling hybrid is symmetric ISA across different core types in a hybrid architecture.

The following table provides additional details on the 11th Generation Intel Core processors supporting hybrid architecture.

	Details
Platform Goals	• New dual screen device category and experiences. • Mobile form factor and battery life. • Workloads: light productivity, browsing, media consumption, connectivity, dual screen.
Hybrid Configuration & ISA	• Four E-cores + one P-core, 4M LLC, SMT disabled on P-core. • Converged to unified/common ISA. • Expose Hybrid ISA (Perfmon, MCA) to debug/performance tuning tools.
Hybrid Scheduling	• Expose all cores (Performance and Efficient) to the OS. • Concurrent P-core and E-core execution and scheduling. • Use new Windows Hybrid scheduling to accommodate IA Hybrid (i.e., HW-guided scheduling).

2 Hybrid Scheduling

2.1 Hardware Guided Scheduling

With hardware guided scheduling, hardware provides dynamic feedback to the OS (a.k.a. hardware feedback interface) in the form of:

- Dynamic performance and energy efficiency capabilities of P-cores and E-cores based on power/thermal limits.
- Idling hints when power and thermal are constrained.

In processors that support hybrid architecture, all cores are exposed to the OS. The OS scheduler is responsible for determining which software threads should be scheduled on which core type.

Hardware support is enumerated via the CPUID instruction and enabled by the OS via writing to a configuration MSR.

Related CPUID:

- CPUID[6].EAX[19] – Indicates support for hardware feedback.
- CPUID[6].EDX[7:0] – Bitmap of supported capabilities.
- CPUID[6].EDX[11:8] – Size of HGS table.
- CPUID[6].EDX[31:16] – Index of logical processor's row in the HGS table.

Configuration MSR:

- IA32_HW_FEEDBACK_CONFIG MSR (17D1H)
    - Bit 0 – Enable HGS.

For more detailed information on this technology, refer to the Intel® 64 and IA-32 Architectures Software Developer Manuals located here: www.intel.com/sdm.

2.2 Intel® Thread Director

With Intel Thread Director, hardware provides runtime feedback to the OS per thread (i.e., enhanced hardware feedback) based on various IPC performance characteristics, in the form of:

- Dynamic performance and energy efficiency capabilities of P-cores and E-cores based on power/thermal limits.
- Idling hints when power and thermal are constrained.

Intel Thread Director is available on Intel Core processors that support the performance hybrid architecture beginning with the 12th Generation Intel Core processor family, to help the OS choose the right core for the right thread.

Thread specific hardware support is enumerated via the CPUID instruction and enabled by the OS via writing to configuration MSRs.

Related CPUID:

- CPUID[6].EAX[23] – Indicates support for Intel Enhanced Hardware Feedback.
- CPUID[6].ECX[11:8] – Number of Intel Thread Director classes.
- CPUID[0x20, ECX=0][EBX[0]] – HRESET instruction support.

Configuration MSRs:

- IA32_HW_FEEDBACK_CONFIG MSR (17D1H)
  - Bit 1 – Enable Intel Thread Director multiclass support.
- IA32_HW_FEEDBACK_THREAD_CONFIG MSR (17D4H)
  - Bit 1 – Enable Intel Thread Director multiclass support.

For more detailed information on this technology, refer to the Intel® 64 and IA-32 Architectures Software Developer Manuals located here: www.intel.com/sdm.

2.3 Scheduling with Intel® Hyper-Threading Technology Enabled on Processors Supporting x86 Hybrid Architecture

E-cores are designed to provide better performance than a logical P-core with both hardware sibling hyper-thread busy.

2.4 Scheduling with a Multi-E-Core Module

E-cores within an idle module help provide better performance than E-cores in a busy module.

2.5 Scheduling Background Threads on x86 Hybrid Architecture

In most scenarios, background threads can leverage scalability and multithread efficiency of E-cores.

3 Key Windows Power and Performance Features

3.1 Thread Scheduling Overview

Processors that support x86 hybrid architecture are categorized based on their performance and efficiency capabilities enumerated in the hardware-OS shared memory area. This memory area is set up and enumerated to the hardware by the OS and is referred to as the hardware feedback memory table in this document. The memory area is then populated by the hardware and notified to the OS upon update.

Scheduling on processors that support x86 hybrid architecture is QoS and then priority-driven and is preemptive in nature. In most scenarios, and within scheduling constraints, the highest QoS, and then the priority thread gets to run on the most performant core.

Developers can also opt in threads to run on Efficient cores at efficient frequencies to optimize power and performance. This action is done by using QoS APIs to define the *PowerThrottling* of the thread. (See API details in Section 6.4.)

Hard processor affinities can disrupt OS decisions. While the affinity might be used to guide threads towards performance or efficiency, given performance/efficiency are dynamic capabilities and not core type based. Therefore hard affinities may not yield desired results. Leveraging PowerThrottling can help guide work towards the right core for performance/efficiency. If affinity is absolutely needed, the CPUSets API can be leveraged. (See details in Section 6.3.)

In 12th Generation Intel Core processors supporting performance hybrid architecture, hardware also provides thread class information to Windows, to help place higher performing threads on a processor. This hint is used within the same or lower QoS/Priority threads.

See specific scheduling examples in Section 5.

3.2 Windows Core Parking Engine Overview

In general, the Windows Core Parking Engine makes global scalability decisions about the workload and determines the optimum set of compute cores for execution. In processors that support x86 hybrid architecture, it additionally helps by determining the optimum set of P-cores and E-cores.

Performance and efficiency are derived from a hardware feedback memory table. In performant cores, the most performant cores are unparked first. In efficient cores, the most efficient cores are unparked first.

Two high level parking configurations exist that can be used on processors supporting x86 hybrid architecture:

- One, where the parking decisions are based off performance capability alone, i.e., standard parking or favored core parking configuration.

- Second, where the parking decisions are based off efficiency and performance capabilities both; this is known as hetero parking configuration.

These configurations can be controlled via the *HeteroPolicy* PPM registry and related parking PPM threshold tunings.

See specific examples and configurations in Section 5. Additional PPM details are found in Section 7.2.

3.3 Performance State Control Engine

Similar to homogeneous Intel systems, Intel® Speed Shift Technology (i.e., HWP) is leveraged by Windows to indicate performance constraints on threads that need to run at efficient performance levels, or tune energy performance preference on different slider positions to meet system wide performance vs. battery life goals.

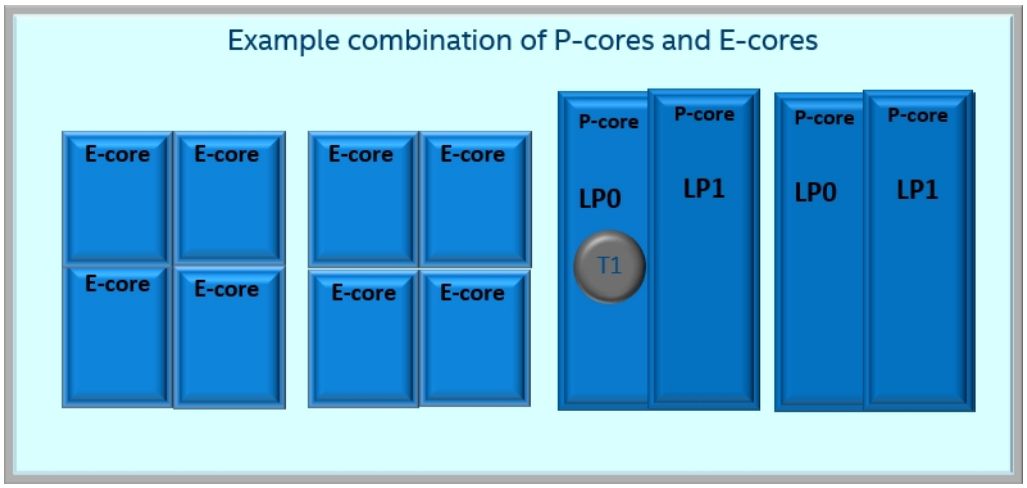
Various PPM settings – *EnergyPerfPreference, MinimumPerformance, FrequencyCap, MaxPerformance* – are dynamically changed by Windows in different Windows Slider Positions, Profile and to meet QoS needs of different threads.

See related PPM setting details in Section 7.3.

4 12th Generation Intel® Core™ Processor Windows Scheduling/Parking Examples

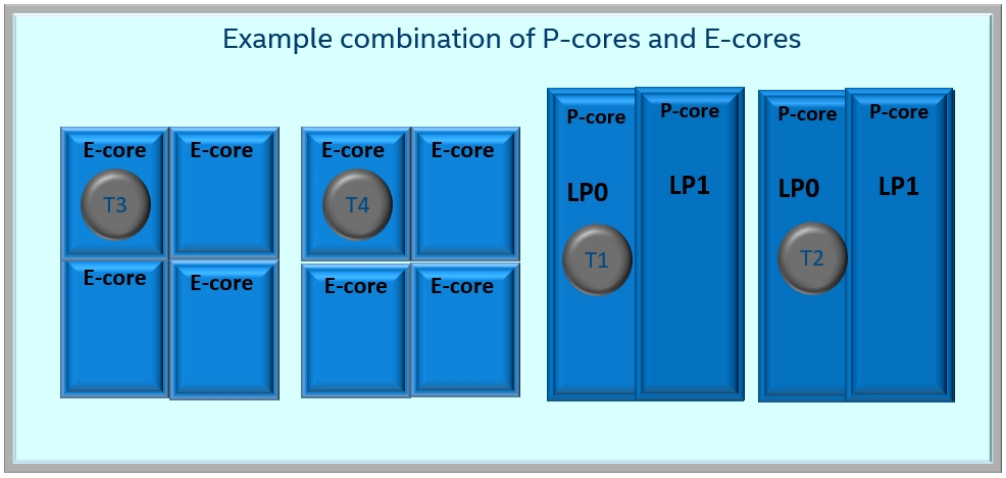
4.1 Single Thread Scenario

The following example shows Windows leveraging an Intel core for single thread performance. This behavior is dynamically achieved when Logical Processor (LP) 0 has the highest performance capability.



4.2 Limited Thread Scenario Example 1

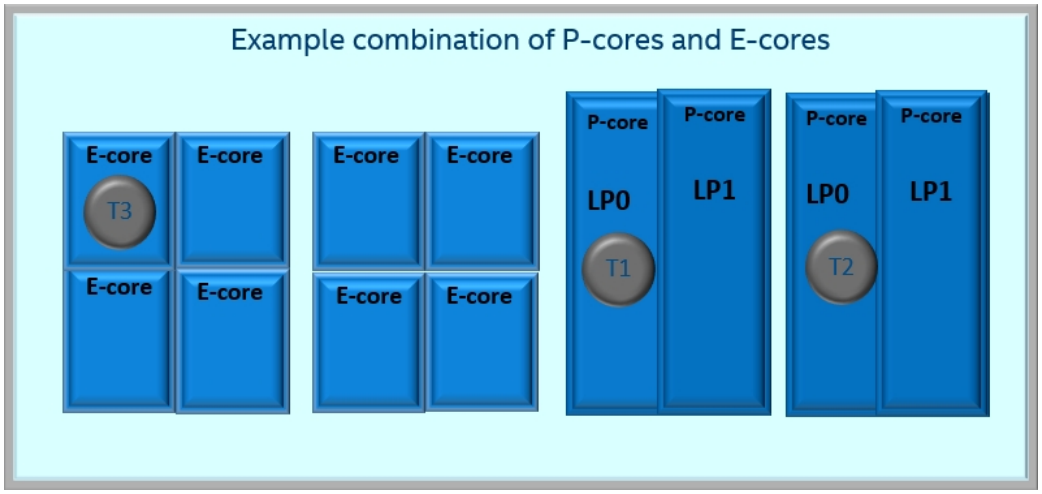
The following example shows an example scheduling behavior in a limited software thread scenario. This behavior is dynamically achieved by the Windows scheduler/parking engine when P-cores are more performant than the E-cores. E-cores are more performant than the SMT sibling of a busy core. When the capabilities dynamically change, Windows automatically accounts for this for optimal scheduling.



4.3 Limited Thread Scenario Example 2

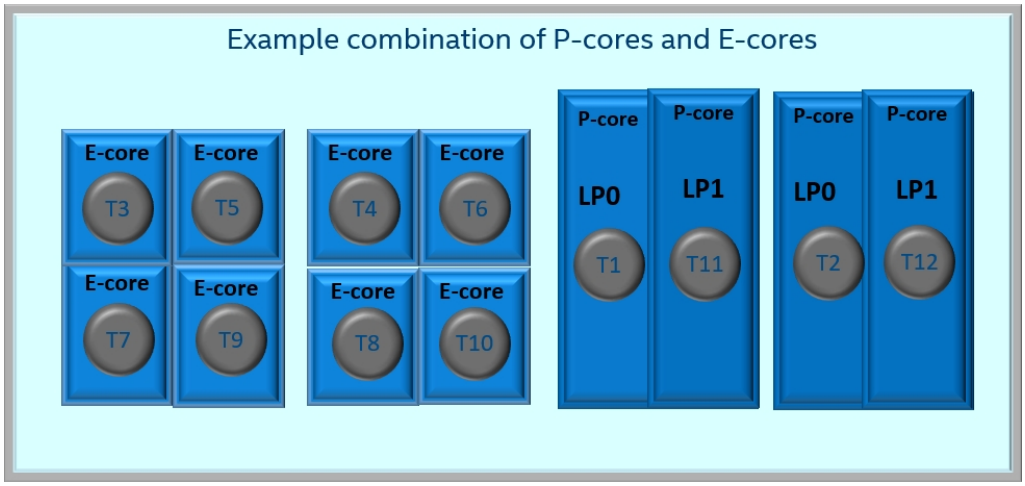
The following example shows T1 and T2 being placed on P-cores, and T3 on an E-core. In this example, T3 has performance capability on core less than T1 and T2 respectively.

When the capabilities dynamically change, Windows automatically considers this for optimal scheduling.



4.4 Multithread Scenario

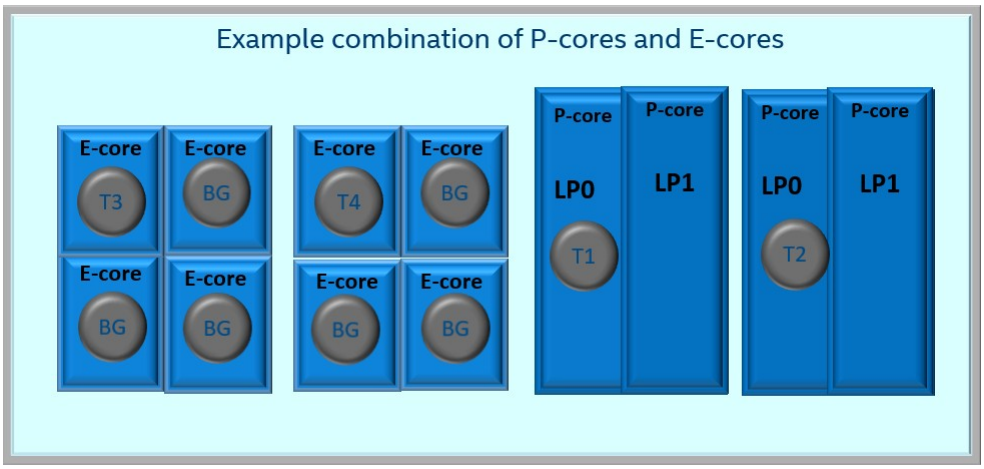
All cores are used by Windows in multithread scenarios.



In power/thermal constraint scenarios, there may be times when all cores aren't used for optimal system performance/efficiency. The behavior is dynamically achieved by hardware providing feedback to Windows, and Windows automatically acting on that feedback.

4.5 Background Threads

The following example shows Windows dynamically placing background threads on Efficient cores. In this example, an E-core has the highest multithreaded efficiency capability.



This option may not be enabled on some performance-oriented configurations and can be enabled by the PPM package for OEM specific tuning.

The behavior is enabled by default if the PPM setting is seen on the *CurrentScheme:*

a. *Background->SchedulingPolicy* is set to 3 or 4.
b. If *Background->SchedulingPolicy* is not present, then check whether *default-> SchedulingPolicy* is set to 5.

In some scenarios, it may be more optimal to run background activity on P-cores when P-cores are idle. This is achieved with Windows Core Parking Engine making optimal decision on which P-cores and E-cores to make available to the scheduler.

4.6 Multimedia Threads

Similar to background threads, multimedia threads are also dynamically placed on the Efficient cores.

This option may not be enabled on some configurations and can be enabled by the PPM package for OEM specific tuning.

The behavior is enabled by default if the below PPM setting is seen on the *CurrentScheme.*

a. *MultiMediaQoS->SchedulingPolicy* is set to 3 or 4.
b. If *MultiMediaQoS->SchedulingPolicy* is not present, then check whether *default-> SchedulingPolicy* is set to 5.

In some scenarios, it may be more efficient to run background activity on cores when cores aren't needed by foreground or higher QoS threads.

4.7 Eco QoS Threads

Similar to background threads, eco threads are also dynamically placed on the Efficient cores.

This option may not be enabled on some configuration and can be enabled by the PPM package for OEM specific tuning.

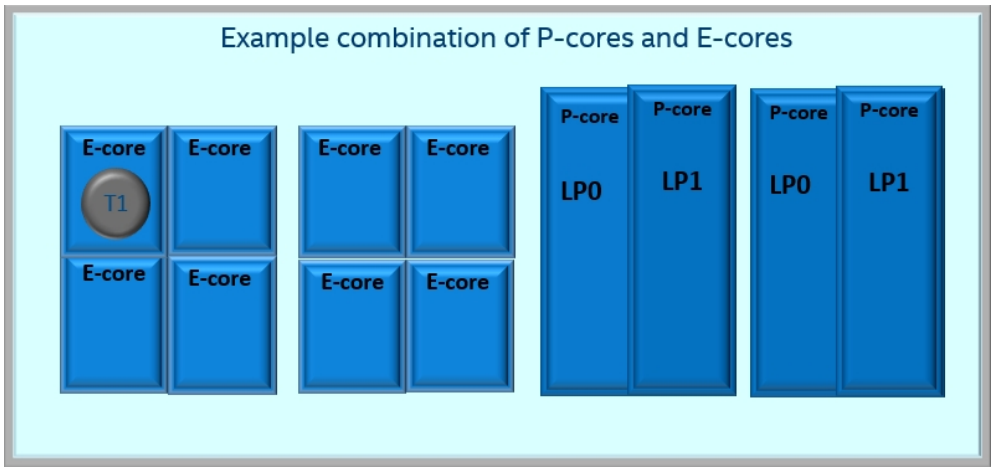
The behavior is enabled by default if the PPM setting is as shown here:

a. *EcoQoS->SchedulingPolicy* is set to 3 or 4.
b. If *EcoQoS->SchedulingPolicy* is not present, then check whether *default-> SchedulingPolicy* is set to 5.

In some scenarios, it may be more efficient to run background activity on cores when the cores aren't needed by foreground or higher QoS threads.

4.8 Low Utilization Threads

In certain low power envelope configurations, it can be more desirable to run low utilization threads on E-cores.



The battery slider settings on some 12th Generation Intel Core processors may enable this behavior by default on Windows. Alternatively, this behavior can be enabled by leveraging the PPM setting – *HeteroParking* policy by the OEMs.

Note: It can come with a performance cost to low utilization important/critical threads; consider this when enabling this Windows Core Parking feature.

4.9 EVENTS TRACING FOR WINDOWS

4.9.1 Is Intel® Thread Director enabled on the system?

From event tracing for Windows, one can successfully confirm whether Windows successfully enabled and is leveraging Intel Thread Director. This is done by checking the "AdvanceHgsEnabled" field under "*microsoft-windows-kernel-processor-power"* event in the tracing log.

4.9.2 How can I check the QoS of the process/thread?

From Event tracing for Windows, under *CPU Usage -> New Thread BAMQoSLevel*, one can identify the QoS levels of the various process or threads. For more details on Windows QoS, refer to section 7.6.

5 11TH GENERATION INTEL® CORE™ PROCESSOR WINDOWS SCHEDULING/PARKING EXAMPLES

Hardware guided scheduling is used by Windows to achieve various goals:

1. The main goal is higher burst performance for ST-dominant workloads. The OS scheduler identifies performance demanding threads and schedules them on the P-core when the P-core has higher performance compared to the E-cores (P-core is not power/thermal constrained).
2. The second goal is to use E-cores for all scenario workloads and background activity where IA performance doesn't provide any additional value and energy efficiency/battery life is the main objective. The expectation is that these workloads execute at an energy efficient frequency where E-cores are more efficient than the P-core for most workloads.
3. The third goal is to enable energy efficient performance for GFX applications. In these applications, it's important to maximize the GFX power budget while providing enough IA performance to feed the GFX device. In most GFX applications, E-cores provide sufficient IA performance with enhanced energy efficiency, maximizing the power budget for GFX. These applications need to be scheduled on E-cores. However, for some GFX workloads it's better to execute on the P-core, either because these workloads need higher IA performance, or the E-core needs to execute at frequencies where P-core is more efficient. Scheduling on a P-core in these cases meets the application performance demand, while also increasing the power budget for GFX.

The following table provides a snapshot of various scenarios with scheduling expectations.

Scenario	Scheduler Behavior
Video Playback App Launch (profile support)	P-core unparked and always immediately available for important threads.
Video Playback (minus App launch) (semiactive scenario)	P-core always parked. Work scheduled on E-core as utilization <60% (unparking threshold*). Unimportant threads do not use P-core.
ADK App launch (profile support)	P-core unparked and always immediately available for important threads.
CINEBENCH minimized	When minimized, CINEBENCH threads run only on an E-core module.
wPrime 1T (Foreground/HighQoS)	wPrime single thread runs on P-core.
Geekbench MT (Foreground/HighQoS MT)	MT performance higher than ST due to five core (one P-core + four E-cores) vs. four core (four E-cores) comparison.
Dual Screen	Opportunity to schedule MediumQoS* different from HighQoS*.
Games/Graphics	Based on power/thermal, P-core parked based on hardware feedback.
Screen Off	P-core always parked. Only E-core available for scheduling.

6 Software Application: Common Questions and Recommendations for Intel® Core™ Processors Supporting x86 Hybrid Architecture

6.1 What are the ISA differences visible to software?

Some ISA differences are visible to software. Examples include:

1. Structural differences (TLB, cache, topology) between P-cores and E-cores.
2. Some aspects of Performance Monitoring, Intel Processor Trace, Last Branch Records, and MCA ISA.
3. Some model specific MSRs.

The expectation is that these differences are not going to impact software applications.

6.2 Can affinity be used on processors supporting x86 hybrid architecture?

In general, software should avoid setting affinity. Setting processor affinity may result in suboptimal performance or efficiency on processors that support x86 hybrid architecture. The performance and efficiency decisions are dynamically taken based on current performance and efficiency capabilities enumerated by the hardware to Windows, and other factors like QoS, priority, etc.

Affinity may be used in certain scenarios. For example, when reading counters on a specific processor.

Alternately, if affinity must be used, then leverage C*PUSets* API – *SetProcessorDefaultCpuSets*, *SetThreadSelectedCpuSets* to declare application affinity in a "soft" manner that is compatible with OS power management.

The same is applicable to interrupts. Drivers should avoid affinitizing interrupts on a specific processor for optimal steering of interrupts to Performance or Efficient cores by Windows on Intel Core processors supporting x86 hybrid architecture.

6.3 What are Windows power throttling APIs? How can application developers influence hybrid scheduling?

Developers can optimally guide work towards Performance or Efficient cores by using the Windows API *SetProcessInformation, SetThreadInformation.*

If these APIs aren't used with the *ProcessPowerThrottling* parameter, then Windows' automatic mechanism is triggered by default. This may lead to Windows misidentifying threads that can leverage Efficient cores (e.g., EcoQoS), and threads that can leverage Performance cores (e.g., HighQoS). Developers can help improve Windows classification by using these APIs, which may help lead to better efficiency/battery life, reduced fan noise, and better performance of the system.

SetProcessInformation function is defined as:

```
BOOL SetProcessInformation(
  HANDLE                    hProcess,
  PROCESS_INFORMATION_CLASS ProcessInformationClass,
  LPVOID                    ProcessInformation,
  DWORD                     ProcessInformationSize
);
```

The following example shows how to call SetProcessInformation with ProcessPowerThrottling to enable throttling policies on the current process.

```
PROCESS_POWER_THROTTLING_STATE PowerThrottling;
RtlZeroMemory(&PowerThrottling, sizeof(PowerThrottling));
PowerThrottling.Version = PROCESS_POWER_THROTTLING_CURRENT_VERSION;

//
// Turn ExecutionSpeed throttling on. ControlMask selects the mechanism and
// StateMask declares which mechanism should be on or off.
//

PowerThrottling.ControlMask = PROCESS_POWER_THROTTLING_EXECUTION_SPEED;
PowerThrottling.StateMask = PROCESS_POWER_THROTTLING_EXECUTION_SPEED;

SetProcessInformation(GetCurrentProcess(),
                      ProcessPowerThrottling,
                      &PowerThrottling,
                      sizeof(PowerThrottling));

//
// Turn ExecutionSpeed throttling off. ControlMask selects the mechanism and
// StateMask is set to zero as mechanisms should be turned off.
//

PowerThrottling.ControlMask = PROCESS_POWER_THROTTLING_EXECUTION_SPEED;
PowerThrottling.StateMask = 0;

SetProcessInformation(GetCurrentProcess(),
                      ProcessPowerThrottling,
                      &PowerThrottling,
                      sizeof(PowerThrottling));

//
// Let system manage all power throttling. ControlMask is set to 0 as we don’t want
// to control any mechanisms.
//

PowerThrottling.ControlMask = 0;
PowerThrottling.StateMask = 0;

SetProcessInformation(GetCurrentProcess(),
                      ProcessPowerThrottling,
                      &PowerThrottling,
                      sizeof(PowerThrottling));
```

6.4 Number of Threads

Thread creation based on processor count may not always be optimal. Some cores may be less performant than others, or even idled/parked. Multithread scaling in certain scenarios is a potential issue and is not just hybrid related.

Understanding the trade-off in using different core count may be important to identify ideal thread count for an application.

6.5 Active Spins

Threads that are doing active spin-waits may impact the performance of important or critical threads. Instead, software can choose to replace spin-waits with lighter weight spins containing UMWAIT, TPAUSE, or PAUSE.

6.6 Driver developers can contribute towards optimal interrupt steering?

6.6.1 Do Not Configure Interrupt Policy

Drivers should opt into Windows interrupt steering by using IrqPolicyMachineDefault as the interrupt policy. This will, by default, enable Windows to steer the interrupts to the core generating the interrupt and works with the core parking engine. By addressing the interrupt on the same core, you avoid latency to wake up an idle core or latency to move metadata to another core for addressing the interrupt, essentially helping provide latency/performance benefit. In addition, avoiding interrupts to the idle core can also have positive battery life impact.

6.6.2 Architectural Relationship

On Windows, software applications can use the user mode API GetLogicalProcessorInformationEx(), or the Kernel mode API KeQueryLogicalProcessorRelationship() to learn about the processor relationship.

These APIs can be used by drivers for optimal queue allocations if per processor queue is being allocated by the driver.

7 Hybrid Key PPM Settings and Setting Values

7.1 Windows Power Management Settings

Starting with Windows 10 October update'18 (RS5), these settings have an additional enhancement to leverage hybrid capabilities:

a. Performance state engine settings.
b. Core parking engine settings.
c. Platform specific controls.

7.2 Core Parking Engine Settings

Core parking engine makes global scalability decisions about the workload and determines the optimum set of compute cores to execute with. In processors supporting x86 hybrid architecture, it additionally determines the optimum set of Performance cores vs. Efficient cores.

7.2.1 HeteroPolicy

`Common\Power\Policy\Settings\Processor\HeteroPolicy`

7.2.1.1 Setting Value: 0 (i.e., Standard Parking or Favored Core Parking)

In this configuration, the optimum set of compute cores are unparked starting with the most performant cores first.

Related PPM Settings for deciding optimum number of cores to unparked:

- *CPMinCores*: Specifies the minimum percentage of logical processors that can be placed in the unparked state at any given time starting with the most performant cores. Logical processor refers to all logical processors that are enabled on the system within each NUMA node.
- *CPMaxCores*: Specifies the maximum percentage of logical processors that can be placed in the unparked state at any given time starting with the most performant cores. Logical processor refers to all logical processors that are enabled on the system within each NUMA node.
- *CPIncreaseTime*: Specifies the minimum amount of time that must elapse before additional parked most performant logical processors can be transitioned from the parked state to the unparked state. The time is specified in units of the number of processor performance time check intervals.
- *CPDecreaseTime*: Specifies the minimum amount of time that must elapse before additional unparked least performant logical processors can be transitioned from the unparked state to the parked state. The time is specified in units of the number of processor performance time check intervals.
- *CPConcurrency*: Specifies the threshold for determining concurrency of the node.
- *CPDistribution*: Specifies the utilization, in percentage, to use in the concurrency distribution to select the number of most performant logical processors to distribute utility to. This

number may be fewer, but never greater, than the number of logical processors that are selected to be unparked.

- *CPHeadroom*: Specifies the value of utilization that would cause the core parking engine to unpark an additional most performant parked logical processor if the least used processor out of the unparked set of processors had more utilization. This setting enables increases in concurrency to be detected.
- *CpLatencyHintUnpark*: Specifies the minimum number of unparked cores (starting with the most performant cores) when a system low latency hint is detected.

7.2.1.2 SettingValue: 4 (i.e., Hetero Parking)

In this configuration, based off utilization, a combination of most performant or most efficient cores are unparked first.

In certain scenarios like low power envelope SKUs or better battery life goals, it can be more efficient to run low utilization work on cores with higher efficiency capability at efficient frequency. This policy is used in these scenarios in combination with optimal performance state engine settings.

Related PPM Settings for deciding optimum number of cores to unparked:

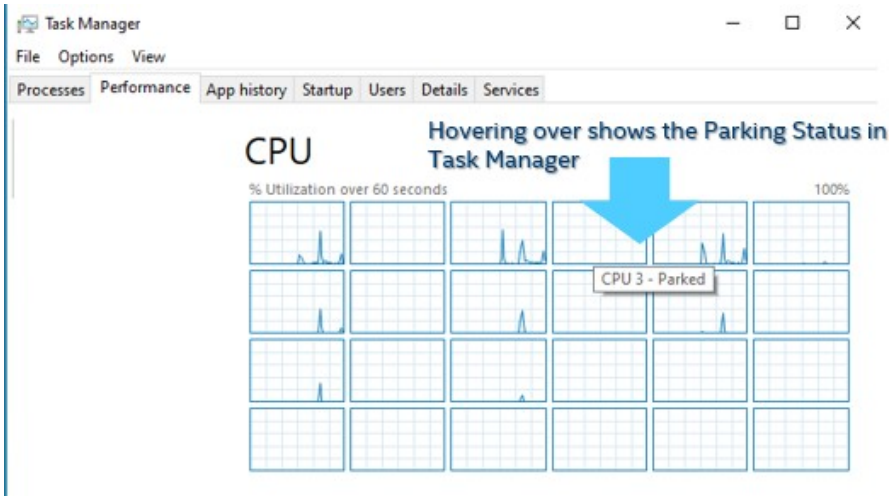
- *CPMinCores*: Specifies the minimum percentage of logical processors that can be placed in the unparked state at any given time. Logical processor refers to all logical processors that are enabled on the system within each NUMA node.
- *CPMaxCores*: Specifies the maximum percentage of logical processors that can be placed in the unparked state at any given time. Logical processor refers to all logical processors that are enabled on the system within each NUMA node.
- *CPIncreaseTime*: Specifies the minimum amount of time that must elapse before additional parked most efficient logical processors in parked state can be transitioned from the parked state to the unparked state. The time is specified in units of the number of processor performance time check intervals.
- *CPDecreaseTime*: Specifies the minimum amount of time that must elapse before additional unparked least efficient logical processors can be transitioned from the unparked state to the parked state. The time is specified in units of the number of processor performance time check intervals.
- *CPConcurrency*: Specifies the threshold for determining concurrency of the node.
- *CPDistribution*: Specifies the utilization, in percentage, to use in the concurrency distribution to select the number of most performant logical processors to distribute utility to. This number may be fewer, but never greater, than the number of logical processors that are selected to be unparked.
- *CPHeadroom*: Specifies the value of utilization that would cause the core parking engine to unpark an additional most performant parked logical processor if the least used processor out of the unparked set of processors had more utilization. This setting enables increases in concurrency to be detected.
- *CpLatencyHintUnpark*: Specifies the minimum number of unparked cores (starting with the most performant cores) when a system low latency hint is detected.

- *HeteroIncreaseThreshold*: Specifies the threshold value to cross above, which is required to unpark the Nth most performant parked core. Separate values exist for each core index.
- *HeteroDecreaseThreshold:* Specifies the threshold value to cross above, which is required to park the Nth least performant unparked core. Separate values exist for each core index.
- *HeteroIncreaseTime:* Specifies minimum amount of time that must elapse before additional parked performant cores (starting from most performant parked processor) are unparked.
- *HeteroDecreaseTime:* Specifies minimum amount of time that must elapse before additional processors (starting from least performant unparked processors) can be transitioned from the unparked state to parked state.
- *HeteroClass1InitialPerf*: Specifies the initial performance percentage of the processors unparked for Performance.
- HeteroClass0FloorPerf: Specifies the floor performance percentage of the efficient processors when at least one processor is unparked for performance.

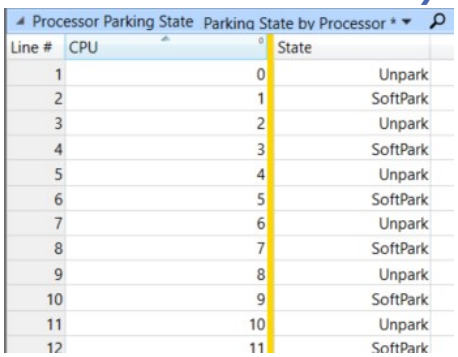
7.2.2 Checking Parking State of a Processor

The parking state of a processor can be viewed through various tools.

7.2.2.1 Example Tool #1: Task Manager



7.2.2.2 Example Tool #2: Event for Windows tracing through Windows Performance Analyzer



Processor Parking State Parking State by Processor *

Line #	CPU	State
1	0	Unpark
2	1	SoftPark
3	2	Unpark
4	3	SoftPark
5	4	Unpark
6	5	SoftPark
7	6	Unpark
8	7	SoftPark
9	8	Unpark
10	9	SoftPark
11	10	Unpark
12	11	SoftPark

7.3 Performance State Settings

On HWP enabled systems, performance state is coordinated via hardware-controlled performance states (i.e., HWP). Various settings exist that can assist HWP hardware engine to make optimal performance state decisions.

7.3.1 PerfEnergyPreference

Specifies the value to program in the energy performance preference (EPP) register for E-cores. The value is specified in terms of percentage and populated by Windows in register on a scale of 0–255.

On HWP Systems, the EPP register is 774H [bits 31:24] for a single logical processor.

7.3.2 PerfEnergyPreference1

Specifies the value to program in the energy performance preference (EPP) register for P-cores. The value is specified in terms of percentage and populated by Windows in register on a scale of 0–255.

On HWP Systems, the EPP register is 774H [bits 31:24] for a single logical processor.

7.3.3 MinimumPerformance

Specifies the value to program in the minimum performance register for E-cores. The value is specified in terms of percentage and populated by Windows in register on a scale of 0–255.

On HWP Systems, the EPP register is 774H [bits 7:0] for a single logical processor.

7.3.4 MinimumPerformance1

Specifies the value to program in the minimum performance register for P-cores. The value is specified in terms of percentage and populated by Windows in register on a scale of 0–255.

On HWP Systems, the EPP register is 774H [bits 7:0] for a single logical processor.

7.3.5 MaximumPerformance

Specifies the value to program in the maximum performance register for E-cores. The value is specified in terms of percentage and populated by Windows in register on a scale of 0–255.

On HWP Systems, the EPP register is 774H [bits 15:8] for a single logical processor.

7.3.6 MaximumPerformance1

Specifies the value to program in the maximum performance register for P-cores. The value is specified in terms of percentage and populated by Windows in register on a scale of 0–255.

On HWP Systems, the EPP register is 774H [bits 15:8] for a single logical processor.

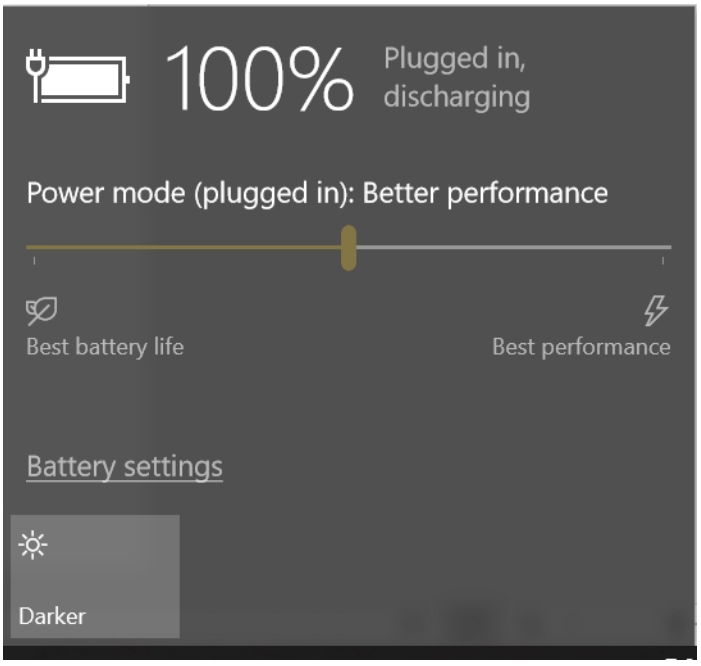
7.4 Windows Performance Power Slider

The Windows performance power slider enables end customers to trade performance of their system quickly and intelligently for longer battery life. As a customer switches between the four slider modes to trade performance for battery life (or vice versa), Windows power settings are engaged behind the scenes.

The different slider positions that are available:

a. Better Battery.
b. Better performance.
c. Best performance.

The following image shows a snapshot of the Windows performance power slider available to the user to switch between the various slider positions.



The PPM settings underneath various slider positions are changing if:

a. PerfEnergyPreference, is different on each slider position, with higher value on more battery-oriented slider position.
b. QoS, profile settings are different, with throttling oriented settings (e.g., higher PerfEnergyPreference) for battery life or power oriented QoS/profiles.

7.5 Key Power Profiles

Like the Windows performance power slider, the Windows power settings can be configured differently for different power profiles. The following sections list the various profiles that Windows supports and recommended configuration.

7.5.1 Default Profile

The default profile is the configuration set that is active most of the time. These settings are identical to the settings for the current power scheme.

7.5.2 Low Latency

Low Latency is the profile that is activated during boot and during app launch time.

7.5.3 Low Power

Low Power is the profile that is activated during the buffering phase of media playback scenarios.

7.5.4 Screen Off

Screen Off is a profile used on modern standby systems. This profile is engaged when the system enters its long-term sleep phase; all system quiescing behavior has completed, no audio is playing, and no mobile hotspot is engaged. This profile is disengaged when the system awakes from sleep.

7.6 QoS, HWP, and Hybrid Scheduling

7.6.1 Default

The behavior is optimized specifically for default and high QoS (or important threads). And, if no other QoS is defined, then also influences other QoS threads.

- Example: Default->EnergyPerfPreference can have different values across different slider positions to help achieve different performance vs. battery life goals.

7.6.2 Occluded Window

The PPM settings allow configuring Occluded Window HWP and scheduling policies to be different from Default/HighQoS threads.

7.6.3 Background Threads

The behavior is optimized specifically for background threads by default with the following PPM Setting:

a. Background->EnergyPerfPreference has a different value compared to Default->EnergyPerPreference.
b. Background->MaximumPerformance has different value compared to Default->MaximumPerformance.
c. Default->FrequencyCap is set to a nonzero value.
d. Background->SchedulingPolicy has different value compared to Default->SchedulingPolicy.
e. Background->SchedulingPolicy is not present, and Default-> SchedulingPolicy is set to 5.

On some configurations (e.g., recommended for best performance slider position), it may be more optimal to have the background settings the same as default settings to avoid throttling of background threads. For this behavior, the settings would be populated as:

a. Background->EnergyPerfPreference = default->EnergyPerPreference.
b. Background->MaximumPerformance = default->MaximumPerformance.
c. Default->FrequencyCap is not set.
d. Background->SchedulingPolicy = default->SchedulingPolicy.
e. Default-> SchedulingPolicy is set to 2.

7.6.4 Multimedia Threads

The behavior is optimized specifically for multimedia threads by default with the following PPM Setting:

a. MMQoS->EnergyPerfPreference has different value compared to default->EnergyPerPreference.
b. MMQoS->MaximumPerformance has different value compared to default->MaximumPerformance.
c. MMQoS->SchedulingPolicy has different value compared to default->SchedulingPolicy.
d. MMQoS->SchedulingPolicy is not present, and Default->SchedulingPolicy is set to 5.

Note: Similar to background threads, on some configurations (for example best performance slider position), it may be more optimal to have multimedia Settings same as default settings to avoid throttling of multimedia threads. For this behavior, the settings would be populated (under example best performance slider position) as:

a. Background->EnergyPerfPreference = default->EnergyPerPreference.
b. Background->MaximumPerformance = default->MaximumPerformance.
c. Background->SchedulingPolicy = Default->SchedulingPolicy.
d. Default-> SchedulingPolicy is set to 2.

7.6.5 Eco Threads

Developers can leverage the Windows power throttling API *SetProcessInformation, SetThreadInformation* to opt-out of Windows dynamic detection of performance or efficiency goals of the developer's application threads or process. The threads marked by developers to run efficiently are called Eco QoS threads.

From the PPM perspective, the behavior is optimized specifically for EcoQoS threads with these settings:

a. EcoQoS->EnergyPerfPreference has higher value compared to default->EnergyPerPreference.
b. EcoQoS->MaximumPerformance has lower value compared to default->MaximumPerformance.
c. EcoQoS->SchedulingPolicy has different value compared to default->SchedulingPolicy.
d. EcoQoS->SchedulingPolicy is not present, and default->SchedulingPolicy is set to 5.