CHAPTER 8

Characterization and Optimization

Servers have a wide variety of different hardware and software configuration options. These include simple options such as enabling and disabling a feature. It also includes more advanced options that allow operators to control usage conditions, functionality, or other feature behavior. The individual features discussed in this book are primarily for power management—for example, P-states, C-states, link, and device states. Servers also have a large number of configurable features that are designed for performance including Turbo, memory prefetchers, or memory controller page policy.
In addition, servers have a large number of configurable features that are designed to add functionality, such as virtualization; security; or reliability, availability, and serviceability (RAS) features.

Servers have an equally large number of software options including different operating systems, applications, and management software options. For any given workload, a server's default hardware and software settings are designed to provide a good balance between low power and high performance. However, for many workloads, these default settings may not be optimal or may not be in line with an operator's performance, power, or cost goals. Most platforms enable power management features, so they are used aggressively when utilization is low. They are used conservatively as utilization increases, and they are disabled when the server reaches full capacity. At full capacity, power management features are either explicitly or implicitly disabled, so power management has no negative impact to maximum throughput.

All of the different types of features just described have an impact on performance, power, or other factors. The specific impact varies significantly based on the workload being measured and the system components being used. In many cases, the only way to definitively understand how various configuration options will impact power or performance is by experimenting on a specific system configuration. For example, an I/O-intensive workload with simple transactions may realize an undesirable performance impact from PCIe L1. However, the impact will vary significantly depending on the I/O (network or storage), device latency, device bandwidth, and whether the system is running at low or high utilization. There is no one-size-fits-all answer to questions about individual feature performance or power characteristics.

Before determining whether to tune, which features to tune, or how to tune, operators need to determine their requirements, such as functionality, power, and performance. It is also important to identify optimization goals. An operator who is optimizing for maximum performance may make decisions regardless of power consumption and will end up choosing a very different set of features and tunings than an operator who wants to minimize power. Other operators may take a more balanced approach; they may seek to lower power as much as possible, but at the same time, still meet a response time (performance) requirement. Other operators will use power management as aggressively as possible as long as those features do not impact maximum throughput. For a small number of servers, the investment in advanced feature tuning may not be worthwhile. However, the value increases significantly as the number of servers deployed increases.

■ **Note** The process outlined in this chapter frequently refers to tuning to decrease power or increase performance and the tradeoffs between the two. You can use the same process and analysis techniques to assess other tradeoffs, such as cost or functionality. In addition, requirements may extend beyond power and performance, such as temperature or reliability limits.

Feature tuning is very straightforward with the right tools and process. Chapters 2-6 describe various power management features, including how they work, and feature power and performance characteristics. Chapter 7 described how to monitor those features to understand their use. This chapter describes a process for characterizing and optimizing servers for the datacenter.

The following steps provide an overview of optimizing a server. Many of these individual steps will be described in greater detail throughout the remainder of the chapter.

1. Set power and performance requirements and optimization goals. For example, minimize energy while meeting a response time requirement or maximize performance below a temperature limit.
2. Collect data in the target environment to understand runtime characteristics. This includes data collected using power, performance, and thermal monitoring capabilities over a range of use conditions.
3. Analyze data to identify gaps relative to requirements and to understand what improvements the operator needs to make to reach optimization goals.
4. Analyze data to uncover new issues and opportunities. For example, an operator may identify during characterization that they only use a fraction of the available memory capacity or that the number of software threads running is frequently less than the number of logical processors.

5. Create a test environment for tuning experimentation. Identify a workload that is representative of the target environment. Reuse industry and open source workloads that model similar applications and services or create new workloads based on the target environment's key characteristics.
6. Identify options to tune including BIOS setup options, OS options, and application options.
7. Measure the target workload and collect data with server default settings. This represents the baseline measurement that all future experiments will be compared against.
8. Measure the target workload with each identified feature and tuning, making only one change at a time.
9. For each change, collect and analyze data to identify the power and performance impact.
10. Identify those changes that aid in meeting requirements and optimization goals and measure the target workload with a combination of these.
11. Deploy beneficial changes to the target environment.
12. Repeat the process whenever there is a significant change to requirements, optimization goals, use conditions, or system components.

Workloads

Workloads are software services, applications, or testing tools that measure the performance of a server. They attempt to model representative usage scenarios based on usage conditions of interest. Workloads provide a repeatable way to measure performance of a system and are particularly useful when you are experimenting with and trying to understand the impact of changes to hardware or software in a datacenter.

Performance can be represented in a variety of different ways depending on the workload of interest. For example, throughput metrics, such as transactions per second or I/O per second, measure the peak performance capabilities of the platform. Latency metrics, such as transaction response times, time to completion for compute jobs, or I/O (drive and network) latency, measure the responsiveness of the platform. Power metrics, such as platform power, memory power, frequencies, and voltages, are used to understand the energy cost per unit of work. Several workloads bring together a combination of the metrics of interest, providing performance per watt or performance per dollar.

Workloads can be used to study a subset of system components, the system as a whole, or a cluster of servers. A profound understanding of system behavior can be gained when representative workloads are coupled with extensive data collection (such as core and uncore performance monitoring units, digital power meters, and OS metrics) and the results are carefully analyzed.

Identifying Suitable Workloads

The first requirement for any workload is that it is measurable. A service, application, or testing tool must have one or more metrics captured during a measurement that can be used to convey various measured throughput, latencies, or a composite metric of the two.

Workloads must be repeatable. If an experiment is conducted twice, three times, or a hundred times without any configuration changes in-between, the measured results must be the same. Workloads with poor repeatability make it difficult or impossible for operators to tell whether a measured change in performance is due to a configuration change or simply due to normal experiment variation. Ideally, workloads used for experimentation have less than 1% variation in measured power and performance; however, 2%-3% is common. If variation exceeds 5%, it presents a problem because the workload can no longer be used to assess the impact of smaller or more subtle changes to system configuration.

Workloads must be reproducible—executing the same transactions or computations, using the same inputs, and following the same order or distribution for every measurement. Reproducible workloads measure performance during a timed interval that is the same for every measurement. They also can be reset to a starting state that is identical across measurements. For example, workloads that utilize a database must be able to restore a backup database before every measurement.

Workloads or systems with poor scalability can affect how repeatable or reproducible a measurement is. For example, a workload that is being used to measure compute performance will not shed light on changes if there is an I/O bottleneck. Similarly, a workload that only utilizes a few threads may not accurately illustrate the performance difference between an eight-core or an eighteen-core processor.

Workloads must be representative. Representative workloads perform similar transactions or computations as the scenario being modeled and also use the same software stack and configuration as the scenario being modeled. For example, a representative Infrastructure as a Service (IaaS) workload would utilize a virtualized environment. It would have virtual machines utilizing heterogeneous applications, it would vary the load on the server over time, and it would vary the number of running instances. Many workloads include random elements to improve their representativeness—for example, workloads that vary the client think time or the interarrival rate of transactions.

Workloads that are not representative typically only model a small portion of the scenario of interest. This makes it more likely that the tuning results from a test environment will not apply to the production environment. For example, a testing tool that measures single-threaded TCP roundtrip latency wouldn't be representative of a web server. An operator could make several changes to improve the performance of the test workload that would have no impact or a negative impact on their production workload.

Another key attribute for characterization and optimization is whether the workload is configurable; for example, does the operator have the ability to change the problem size in a scientific workload or to change the number of connected clients in a transactional workload? Having ample configuration options is key to tailoring the workload setting to best match an environment of interest.

Workload Types

There are a wide variety of different workload types—for example, testing tools, energy efficiency benchmarks, industry benchmarks, and datacenter workloads. Each has different applications, different purposes or goals, and a different level of complexity.

Testing Tools

Testing tools don't necessarily model a representative service or application; instead, they are used to stress a single component or subset of related components in the system. For example, Intel provides a testing tool to characterize cache and memory performance. The Intel Memory Latency Checker tool (Intel MLC) can be used to measure maximum memory bandwidth, idle latency, and loaded latency.[1] Although these tools are great for testing some key system characteristics, it is also relatively easy to misinterpret the results. One common mistake is to measure idle memory latency with clock-enabled (CKE) power savings enabled. These power savings commonly engage during idle latency tests, causing a significant increase in the idle latency. CKE tends to have much smaller impacts on real system performance.

There are also testing tools that focus on I/O. The open source iperf workload is popular for measuring network bandwidth; the NetPIPE workload is popular for network latency. For storage, the open source FIO or IOmeter tools are popular and flexible and allow users to vary different I/O parameters and think times.

Testing tools can give a preliminary indication of how a given feature might impact power or performance. They are also very helpful in identifying the base capabilities of a platform and can be used as guides for detecting bottlenecks. Individual components can be monitored when testing tools are being measured to understand state residencies, bandwidth, or throughput limits. Chapter 7 provides an outline of the different types of metrics to look at when monitoring components.

It can be easy to misinterpret the results of some micro benchmarks. For example, idle latency benchmarks (such as the Intel MLC) will exhibit significant increases in latency as a result of memory CKE power savings features (see Chapter 3). Not only will CKE result in an increase in idle latency due to the CKE wakeup, but the precharge powerdown (PPD) feature will also result in closed memory pages (and further latency increase). It is not uncommon for users to draw the conclusion that CKE causes significant performance loss based on this benchmark. In practice on real workloads, CKE has a much smaller impact on latency.

Energy Efficiency Workloads

Energy efficiency workloads are used to study both the power and performance characteristics of a system. They exercise CPU and memory and provide a great preliminary analysis of system behavior. The most popular energy efficiency workloads are SPECpower_ssj2008 (SPECpower) and the Server Efficiency Rating Tool (SERT) from

[1]See https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

the Standard Performance Evaluation Corporation (SPEC). SPECpower is widely used across the industry and features several years of published results, so there is a great amount of data to use for system comparisons.

SPECpower

SPECpower measures simple transactions running in a Java Virtual Machine (JVM) across a broad range of CPU and memory utilization. It includes idle, maximum throughput, and a range of load points in between those endpoints. It provides a way to visualize power consumption at various ratios of maximum performance. This visualization, with power on one axis and performance on the other axis, is commonly called the *load line*. There are examples of this visualization spread throughout this book; however, many examples use a workload other than SPECpower. The load line methodology introduced by SPEC in SPECpower has seen broad use across the industry because it can be easily applied to different workloads.

Figure 8-1 shows CPU power during a SPECpower measurement. It illustrates several calibration load points used to determine system performance, as well as several load points of varying utilization all the way down to idle.

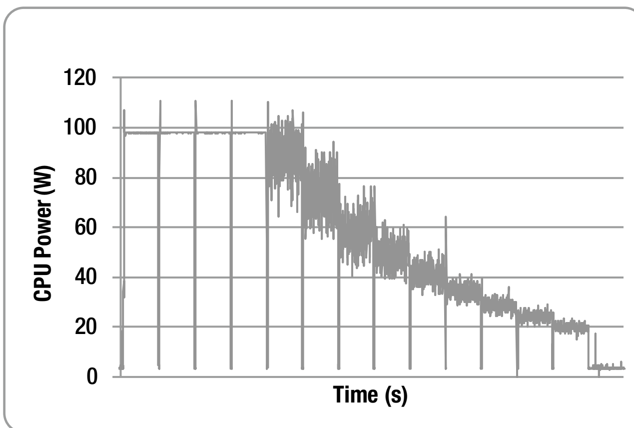

Figure 8-1. *CPU power measured over various load levels of SPECpower*

SPECpower has some representative characteristics, but it lacks more sophisticated transactions that are common in datacenter workloads. It does not exercise a complete software stack and does not have any significant storage or network I/O. As a result, SPECpower includes some characteristics that are not commonly seen in production workloads. SPECpower has an order of magnitude fewer power state transitions compared to typical datacenter workloads. Residency in low-power states is much higher in SPECpower than in other workloads. In SPECpower, transactions are initiated in batches, rather than being individually initiated by network connected clients. This means the workload is both idle and active for longer periods of time, leading to very different P-state behavior than is seen with a typical datacenter workload.

Due to the workload's focus on CPU and memory, certain features or platform-level changes may improve SPECpower scores, but they do not provide a benefit in a typical datacenter workload. As is the case with every workload, there are optimizations that may benefit SPECpower that aren't generally a good idea to apply elsewhere.

■ **Note** A significant challenge in workload selection is balancing ease of use with representativeness. Workloads that model realistic use scenarios tend to include a large number of network-connected clients, sophisticated software stacks, and multiple different systems under test. Workloads that are easy to install, measure, and maintain do not.

The Server Efficiency Rating Tool (SERT)

SERT is a new SPEC tool suite under development that measures power and performance across a broader range of use conditions and applications. Rather than executing a single test, it includes a variety of different worklets that stress the CPU (different types of operations, including mixing integer, floating point, data references, and modification), memory, and drives separately. SPECpower loads are one of the worklets, so SERT extends the type of analysis that can be done above and beyond SPECpower.

SERT has been adopted by the Environmental Protection Agency (EPA) for their Energy Star program for servers. SERT is also being investigated for use in energy efficiency programs by government agencies around the globe including Europe and Asia. It is interesting to note that SERT is being used not only for power and performance assessments, but is also being considered for government-based environmental programs.

Industry Workloads

Industry workloads are typically two- or three-tier workloads with transactions driven over the network based on more realistic transaction interarrival rates. These workloads use representative application and software stacks and often include quality of service or response time requirements. The ability to measure transaction response time is a key capability because it allows operators to characterize the performance impact of power management features. The downside to using industry workloads is that they represent a significant resource investment, both in engineer time and hardware.

Industry workloads are one of the best tools available to characterize and optimize a server. These workloads are maintained and updated by various open source efforts and an industry consortium, and they see extensive use across the industry. The majority of charts in this book were generated using monitoring power and performance with industry workloads.

A number of good industry workloads span various market segments. For example, many HPC and scientific workloads span life sciences, computer-aided engineering, financial modeling, and weather simulation, and many of these are open source. Similarly, there are a number of enterprise workloads modeling database and mail servers, customer relationship management (CRM) systems, and enterprise

resource planning (ERP) systems, and there are also many cloud workloads that model multitenant environments or that model environments with distributed services that utilize the Web, memory cache, and databases.

The Transaction Processing Performance Council (TPC) and SPEC are two industry organizations that develop and maintain workloads. TPC workloads such as TPC-C (order-entry OLTP), TPC-E (brokerage-firm OLTP), and TPC-DS (decision support system) are popular for power and performance characterization. SPEC workloads such as SPECweb (web server), SPECvirt (infrastructure consolidation), and SPECimap (mail server) provide similar capabilities. In addition to the industry organizations, there are also a number of company-sponsored workloads such as SAP Sales and Distribution (SAP SD) workload or VMware VMmark.

In addition to the workloads maintained by industry organizations, a number of good open source workloads span various market segments: Olio (web, social networking), mcblaster (object cache), HammerDB (OLTP), and TPoX (Transaction Processing over XML), for example.

■ **Note** Open source workloads are an excellent starting point for workload development or for customizing a workload to better model a target environment.

Industry workloads are more complex and realistic, and they represent a step above. Unlike SPECpower, most industry workloads do not have integrated power and performance metrics, and they do not automatically generate a load line. However, the load line concept from SPECpower can be easily applied to industry workloads.

Industry workloads allow users to create a variety of different loads, typically specified by a number of virtual clients or delay time between client transactions. For example, Olio requires an operator to specify the number of users that will be used to generate different types of web transactions. Measurement across a variety of different utilization or throughput levels is possible by running with a different number of users. For many virtual workloads, several thousand virtual clients are required to reach maximum throughput, so it is easy to make fine-grained adjustments to load based on varying the number of users.

Idle Workloads

As is the case with industry workloads, the conditions used to test an idle server should also be representative. Many idle power measurements are taken shortly after a server is powered on and initialization is complete. The server may not have any applications or services initialized and ready for use. This state is called *operating system idle*. In operating system idle, it is typical to see long, uninterrupted idle durations and excellent residency in package C-states.

To get a representative measurement, it is best to measure idle on a system that has all applications and services initialized and has recently completed running a representative workload. This state is called *active idle*, and unlike operating system idle, it may include intermittent network activity, periodic management and security

operations, and sporadic application activity. Active idle for virtualized environments typically includes a variety of different virtual machines running different software stacks, adding to its complexity. As the name implies, active idle is significantly more active than operating system idle, leading to higher power. Figure 8-2 illustrates the difference in activity between operating system idle and active idle with numerous applications and services loaded.

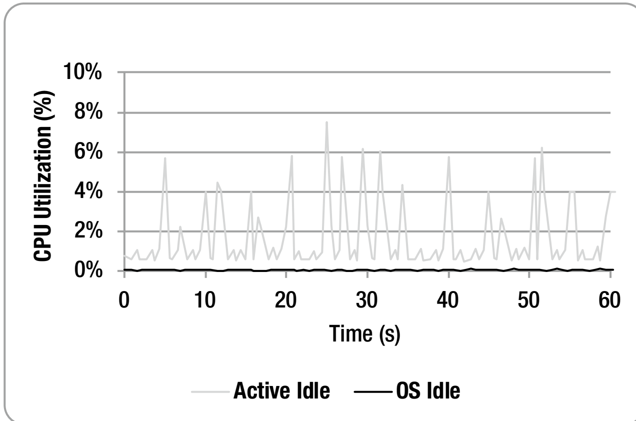

Figure 8-2. *Variation in CPU utilization between active idle and OS idle*

Measuring and comparing the differences in operating system idle and active idle can help to identify software and hardware components that impact idle power. For example, applications that utilize polling rather than events or applications that change the timer frequency of the operating system can both result in higher idle power. Because most servers spend a significant amount of time idle, optimizing for idle can be as beneficial as optimizing for active loads. The tools operators can use to diagnose active idle issues are discussed in Chapter 7.

System Characterization

Hardware and software monitoring tools are key capabilities needed for system characterization. The best techniques for data collection can change based on the workload type or based on the amount or type of data collected.

Steady State vs. Non-Steady State

When monitoring workloads it's important to identify whether the workload of interest is steady state or non-steady state. During a steady state workload, system characteristics don't change significantly over time. Industry benchmarks that model transactional systems are typically steady state workloads; TPC-E or SPECweb are examples of this type of workload. These workloads execute a predetermined mix of transactions, repeatedly, over a very long period of time.

For many workloads, there is a significant amount of ramp-up time before steady state is reached. When a workload first starts, application and system characteristics are changing frequently as clients connect, load is balanced, and frequently-used data is cached.

Non-steady state workloads consist of frequently changing system characteristics. It's common for application behavior to change from one second to the next—for example, a scientific application modeling weather patterns or a data analytics workload with different map and reduce phases. Figure 8-3 shows how the rate of instruction retirement varies for a steady state and non-steady state workload. The multiple different phases of the non-steady state workload can be clearly identified.

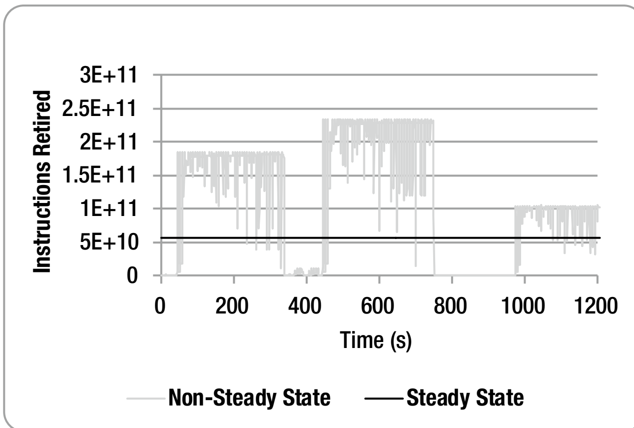

Figure 8-3. *Variation in instructions retired between non-steady state and steady state workloads*

Data Collection

There are a number of important considerations for collecting data during workloads. If data collection tools are executed too frequently or multiple tools are executed in parallel, it can alter the behavior of the workload. If data collection is not started and stopped at a consistent point in time, data cannot be compared between measurements.

Collection Duration

The start time and length of data collection depends on whether the workload is steady state or non-steady state. For steady state workloads, power and performance data is typically not collected during ramp-up since initialization phases are not of interest. Similarly, data collection can be skipped during ramp-down phases or when a workload has finished and is restoring files to a known starting state.

When data is collected during the steady state between ramp-up and ramp-down, it is only necessary to collect data for a small portion of time because system characteristics don't change significantly from one moment to the next. Data collected

during one minute of steady state should have characteristics very similar to data collected during the next minute of data collection. Data collection may need to be started and stopped for workloads that have several significant intervals. For example, SPECpower may require tools to be started and stopped at specific times to allow for load points to be individually analyzed.

For non-steady state workloads, performance and power data is typically collected throughout the entire duration of a measurement. Unlike steady state workloads, it is not reasonable to collect data for a smaller portion of time since the characteristics of data collected during one minute of time can be very different from the data collected during the remainder of the workload.

Collection Frequency

If power and performance events are collected at high frequency, data collection tools will interrupt applications frequently, stealing CPU cycles from the workload of interest. This may alter the natural behavior of the workload, revealing power or performance issues that would not occur when data collection is stopped. If events are collected at low frequency, interesting workload characteristics may be difficult to discern. Operators will be unable to both identify unique phases of the workload and determine how power and performance characteristics change over time.

■ **Note** Unlike hardware monitoring tools, operating system tools can collect hundreds of different events in a single interval. Collecting data at low frequency can still perturb the workload if too many events are collected at the same time.

For steady state workloads, data collected at low frequency will produce a similar result as data collected at high frequency. This simplifies the data collection process for steady state workloads since the workload can be characterized with only a small amount of data. For non-steady state workloads, high frequency collection is necessary. Finding the ideal data collection frequency will require some experimentation as it varies from workload to workload. The goal is to collect data as frequently as possible while having no impact on system power or performance.

Event Ordering and Event Groups

Many times it is necessary to analyze several events collected at the same time to get a complete picture of component or system behavior. For example, to characterize memory references during a particular phase of a workload, an operator may want to measure read transactions, write transactions, page hits, page empty accesses, page misses, and memory latency. As discussed in Chapter 7, this may not be possible since many monitoring units can only collect a small number of events in parallel.

For steady state workloads, this isn't a significant issue. Monitored events measure very similar values from one moment to the next, so it is possible to gain the desired insight into component or system behavior by splitting up events over several different groups measured at different times. For non-steady state workloads, the limitation of monitoring units can be an issue. The best practice is to collect related events as close (in time) to each other as possible and to collect the same event groups repeatedly at high frequency.

Multiple Tools

During workload characterization, operators may be required to measure data using several different tools targeting monitoring capabilities spread across software, firmware, and hardware. When multiple tools are used, there are some cases where it is beneficial to measure multiple tools simultaneously. For example, to provide complete coverage for a non-steady state workloads or when there is interest in comparing software metrics to hardware metrics collected at the same time. There are other cases where it is beneficial to measure multiple tools one at a time. For example, to limit how intrusive data collection is during a steady state workload.

Similar to the process of determining collection frequency, event groups, and event ordering, there are several choices for how to collect data using multiple tools. These require some experimentation as each of those choices has unique strengths and weaknesses.

Methodology

Characterizing and optimizing a server requires not only good workloads and a good data collection strategy, it also requires good methodology. The following recommendations serve as guidelines for basic server characterization.

- Quality test each server before investing time in characterization and analysis. Use testing tools to confirm the configuration is healthy, and measure the target workload several times to ensure workload variability is within acceptable limits.
- Always collect data with monitoring tools. Without proper visibility into power and performance characteristics, the result of changes to software or hardware from one measurement to the next will not be well understood.
- Always use a consistent baseline measurement, making no change to the configuration throughout the duration of an experiment. For example, if a driver or BIOS is updated, drive capacity is added, or a network is reconfigured, measurements on the server can no longer be compared to previous measurements.
- Only change one variable at a time between measurements. For example, coupling several software optimizations together in a measurement may result in no change in power, whereas in reality, some of the optimizations may be helpful while the others are harmful.

- Always automate workloads and data collection. This ensures measurements will collect data with the same start time, end time, collection frequency, and event ordering every time. Without this consistency in timing, comparisons between different measurements may provide misleading results.
- Use a controlled thermal environment. Repeatability of a workload can be impacted if the temperature varies significantly based on weather, time of day, or activity of other systems.

Analysis

With thousands of different events that can be monitored during a measurement, using a top-down analysis is one of the best strategies for analyzing changes to a server. First, it is important to look at the big picture. The most important metrics to analyze are power, performance, and cost since most feature tuning will result in an increase or decrease in one or more of these. Performance will be measured in throughput and latency (response time or time to completion) and will be collected by the target workload. Power will be measured either using a digital power meter or through the power supply or current sensors using the monitoring features described in Chapter 7.

Power Metrics

Configuration and tuning changes can cause very large differences in system power. Figure 8-4 highlights how substantial the change can be. The lowest power tuning in this example is able to minimize the energy cost per transaction without impacting maximum throughput.

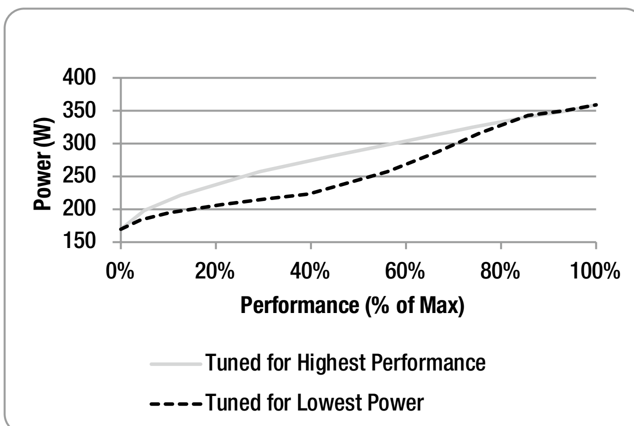

Figure 8-4. *Comparison between power and performance (throughput)—Server OLTP Workload*

To build a deeper understanding of a configuration or tuning change, first identify the component or components that caused the change. Then, identify specifically why those components' power changed. For a given power increase or decrease, there are several events and metrics in the data collected that can be used to identify the cause.

In an ideal scenario, the system is capable of component-level power measurements, and the components contributing to a change in power can be identified simply by comparing the power of individual components (processors, memory, PCH, LAN adapters, drives, and fans).

However, systems with component-level power measurement capability are uncommon today. A more assessable approach to determining a change in power is to identify differences in active and idle states and operating conditions. These are the primary factors that ultimately determine power consumption. For example, a change in CPU power could be identified by analyzing CPU C-state and P-state residency and transitions. A change in fan power could be identified by analyzing each fan's tachometer, and a change in interconnect power could be identified by analyzing interconnect voltage, frequency, and link width.

If power measurement and monitoring capabilities are insufficient to identify what caused a change in power, performance and thermal events can be used to provide alternative insight. For example, if a measurement showed a 10-times increase in IOPS to a SSD, this increased activity may indicate a measurable increase in drive power.

A change in system power is always the result of a single component change because there are many complex dependencies between various components in the system. For example, an operator could decrease memory power by limiting the memory frequency used and capacity installed; however, this change might increase system power as CPUs are stalled waiting for data to be returned from memory and drives. As the workloads used become more sophisticated, the number and complexity of component and system dependencies increase. For example, tuning any given node in a cluster of servers running a distributed application may result in a small local difference but a substantial global difference when the cluster is viewed as a whole.

The following list serves as a prioritized guide for top-down power analysis. Although this list is targeted for power analysis, performance metrics are also valuable, and will be covered momentarily. See Chapter 7 for more details on how to monitor these various metrics.

- CPU Power
    - P-state residency
    - C-state residency and transitions
    - Temperature
- Memory power
    - Frequency and voltage (static at runtime)
    - Self-refresh and CKE residency and transitions

- Thermal management
    - Platform temperatures
    - Fan power
    - Tachometer
- Device power
    - Link width and frequency
    - I/O bandwidth and transactions per second
    - QPI L1 and L0p residency and transitions
- Other
    - Additional system performance metrics

Performance Metrics

Configuration and tuning changes can cause very large differences in system performance. Figure 8-5 highlights how substantial the change can be. This figure underlines the importance of performance requirements. If this server had a performance requirement that 95% of transactions are completed in less than 10 milliseconds, then the lowest power tuning would decrease power and cost while still meeting that performance requirement. However, if the server had a performance requirement that 95% of transactions are completed in less than 2 milliseconds, an operator may both tune for performance and load systems to no more than 80% of maximum capacity.

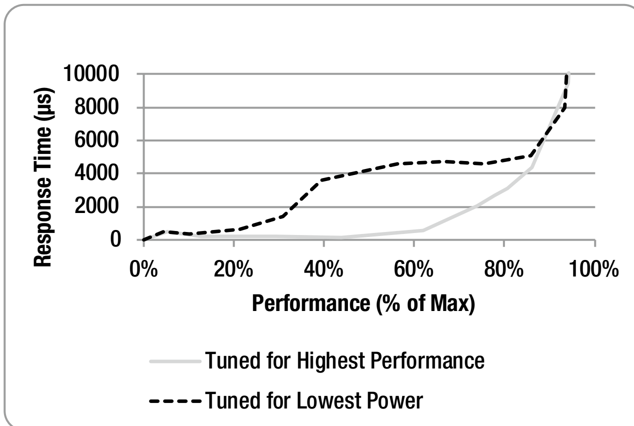

Figure 8-5. *Comparison between response time and performance (throughput)*

To build a deeper understanding of a configuration or tuning change, first identify the component or components that caused the change. Then identify specifically why those components' performance changed. This process of performance analysis is easier if the target workload's sensitivity to CPU frequency and I/O subsystem performance is well understood.

■ **Note** One good method for determining sensitivity to CPU performance is through frequency scaling studies. This involves running a target workload several times across a range of fixed frequencies. The results allow operators to calculate scaling efficiency between two frequencies by comparing the percent increase in performance to the percent increase in frequency.

If a workload exhibits poor frequency scaling, transaction time is likely dominated by waiting for memory or I/O. Alternatively, the poor frequency scaling may be the result of a bottleneck. For example, a datacenter workload that drives line rate network traffic is unlikely to see a significant increase or decrease in performance based on CPU frequency.

Differences in performance can be identified by analyzing the changes in CPI, path length, or power state residencies. For example, a decrease in operating frequency will almost certainly result in a decrease in performance. An increase in CPI could be the result of a degrading cache hit rate or increasing memory latency. The source of an increase in path length can be identified by analyzing execution profiles collected by tools such as Linux perf. The execution profile will allow an operator to drill down to the specific process, modules, or function that is exhibiting a change in behavior.

The following list is a prioritized guide for top-down analysis. Note that system power, covered in the previous section, is very important to consider when understanding changes in performance. For example, monitoring C-state residency and transitions not only tells the operator about time spent in state, it also tells about accumulated exit latency (C0 impact) and effects of flushing caches (CPI impact). See Chapter 7 for more details on how to monitor these various metrics.

- CPU performance
    - Cycles per instruction (CPI)
    - Cache misses and latency
    - Memory latency
    - C-state transitions (latency) and residency (C0)
    - Path length
    - Software execution profiles
    - P-state residency (frequency)
    - Thermal throttling

- Memory performance
  - Memory latency
  - Self-refresh and CKE residency and transitions
  - Memory bandwidth
  - Thermal throttling
- Interconnect power
  - QPI L1 and L0p residency and transitions
- Device performance
  - Link width and frequency
  - I/O bandwidth and transactions per second

Optimization

There are a large number of different optimization opportunities in the system, and it can be challenging to know which are worth the effort. This section explores a variety of different optimization opportunities and investigates the potential tradeoffs involved. In addition to highlighting valuable opportunities, it will cover a selection of items that you may want to avoid. Some optimizations must be performed at boot in the BIOS, whereas others may be possible at runtime. This chapter provides recipes, where possible, for how to make different changes in the system.

Power management algorithms can also provide improved performance in select cases. This section will highlight some of these opportunities for improved performance.

CPU Power Management

Typically, the CPU is one of the first places to start performing power (and performance) optimizations because it frequently contributes significantly to the overall power in the platform. However, there are times when the CPU is not as significant to the overall power consumption of the platform. A good example is a storage system with a large JBOD (just a bunch of disks). In such a system, the CPU power may not be a major contributor to the overall power, and it may not be worth the effort to focus on CPU optimizations. One of the first tasks in performing any optimization work is to determine where the power is going. Total platform power should be characterized and compared against the CPU subcomponent power (see Chapter 7 for details). It is common for the CPU (and memory) to contribute a significant percentage of the overall platform power, but this should be confirmed prior to focusing significant effort on optimizations.

Chapter 2 provides background on many of the features discussed in this section.

P-States and Turbo

Voltage and frequency can play a significant factor in power consumption. Running at a lower frequency when performance is not required can save significant platform power, particularly on high TDP Xeon processors.

Running at lower frequency (P-states) will likely increase the response time of workloads when they are running at low to moderate utilizations. However, as shown in Figure 8-5, many workloads exhibit much higher overall response times when running at peak utilization (with or without power management) than what is observed when power management is enabled at lower utilizations. As a result, the response time impact of these features may not be the dominant component in the worst-case latency situation.

Turbo can be used to increase the achieved frequency beyond the base frequency. On recent processor generations, it has been common for Turbo to provide a 10%–20% (or more) peak performance increase. A common misconception is that Turbo provides "burst" performance for only a short period of time. Although this is frequently true in thermally constrained consumer devices, it is generally not the case in server deployments. Server workloads can frequently sustain some level of Turbo indefinitely.

RAPL (and other frequency management algorithms) can engage while Turbo is running to limit the frequency of the system. Frequency is typically managed by these algorithms on (small) millisecond granularities. Frequency transitions block execution for about 10 to 20 microseconds. The algorithms have been tuned so that these transition periods have minimal impact on the overall throughput (and performance) of the system. However, users who are very sensitive to latency disturbances, such as high-frequency traders, may not want to use Turbo in order to avoid execution being blocked. Many server workloads are able to easily tolerate this latency cost, and the benefits provided by the increased frequency far outweigh the latency cost.

A common misconception is that Turbo frequencies are less power efficient than running with Turbo disabled. Although this is true on some processor SKUs, it is not always the case. Some lower power and lower frequency products achieve optimal platform performance per watt while running in Turbo.

Modern operating systems take tens of milliseconds to detect changes in demand and utilization. As a result, the use of OS-controlled P-states can result in short periods of time where the operating frequency is lower than what would be best for the demand of the system. One of the potential upsides of hardware power management in future products is the ability to improve the response time to changes in demand and utilization.

■ **Note** The use of P-states and Turbo does not have to be a yes-or-no question. Modern operating systems can be constrained to request frequencies within a range. This can result in significant power efficiency savings with contained impacts to response time.

P-states and Turbo need not be an all-or-nothing decision. For example, a user has a system that typically runs at ~60% utilization, but periodically it has an increase in demand. That user's system has a SKU that can run at an "all-core Turbo" (P0*n*) frequency of 3 GHz and a base frequency (P1) of 2.6 GHz. When the customer runs the system with Turbo requested all the time, there is a notable increase in peak performance

(which is useful for those periods of high demand), but it comes at a notable power increase during the typical levels of demand. Running with all of Turbo and P-states enabled results in good power efficiency, but at times, it has undesirable response time characteristics. In such a case, the user could instruct the OS to always request at least 2.4 GHz. Under such a configuration, the user is able to avoid the power cost of running at 3 GHz at typical utilizations, but then transition up to 3 GHz when demand increases.

■ **Note** It is recommended that both Turbo and P-states be controlled through the operating system and not through the BIOS. This provides significant flexibility for a longer term to changes in decisions without requiring system reboots (which can be very undesirable in large-scale deployments).

The BIOS does have the ability to disable Turbo, and many BIOS designers expose this option to end users. However, there is no way to disable frequency transitions in the system. Some BIOSes include an option to disable EIST (Enhanced Intel Speedstep, but this will only change how P-states are enumerated to the OS in ACPI.

■ **Note** Some OEMs have proprietary mechanisms for managing frequency that exist between the OS and the CPU by leveraging capabilities made possible by ACPI. Special care may be necessary if these capabilities are enabled in the OEM's BIOS.

Frequency Control in Windows

In Windows, frequency can be managed through the Power Options control panel as well as by using the `powercfg` command line utility. The High Performance tuning option (available both in the control panel and through `powercfg`) will instruct the OS to request the maximum frequency at all times. By default, the Balanced mode will request frequencies across the spectrum, including Turbo if it was enabled by the BIOS.[2] Under the Advanced Settings options, you can find additional tuning options to control the range of frequencies that are selected (see the Minimum Processor State and Maximum Processor State configuration options). Note that these options do not control actions in the Turbo range.

The `powercfg` utility can also be used to manage both frequency selection and Turbo.[3] Turbo can be disabled with the following command line (or enabled by changing the 0 to a 1):

```
powercfg -setacvalueindex scheme_current sub_processor PERFBOOSTMODE 0
powercfg -setactive scheme_current
```

---

[2]Note that Intel processors prior to the Westmere generation had Turbo disabled by default in the Windows Balanced configuration mode.

[3]See the *Performance Tuning Guidelines* for more details about `powercfg`. https://msdn.microsoft.com/en-us/library/windows/hardware/dn529134.

Frequency selection outside the Turbo range can also be controlled with powercfg:

```
powercfg -setacvalueindex scheme_current sub_processor PROCTHROTTLEMIN 60
powercfg -setacvalueindex scheme_current sub_processor PROCTHROTTLEMAX 100
powercfg -setactive scheme_current
```

The current system configuration can be dumped from powercfg with this command:

```
powercfg -Q,
```

Frequency Control in Linux

The Intel P-state Linux driver provides a mechanism for constraining the requested frequencies in sysfs.[4] These are located today in

```
/sys/devices/system/cpu/intel_pstate/
```

The max_perf_pct, min_perf_pct, and no_turbo files can be used to configure the desired P-state behavior on a per-logical-processor granularity. The percentage values that are configured here include control in the Turbo range, making it possible to limit the maximum frequency used even in the Turbo range. Although it is not possible to determine which part of the range is for Turbo today using the sysfs interface, you can disable Turbo at runtime by executing the following (as root):

```
# echo 1 >> /sys/devices/system/cpu/intel_pstate/no_turbo
```

The algorithms that select frequency can be further tuned using debugfs. However, most users have generally reported minimal benefit by moving away from the default configurations.

Prior to the introduction of the intel_pstate driver, the acpi_freq driver was used by default to manage frequency in Linux. Little effort has been spent in recent years to optimize this driver to work well in a server environment, and it is not recommended. One drawback of the intel_pstate driver is that support is required in the actual kernel for it. It is not possible to load the driver as a module on an older kernel. Note that it is possible to fall back to the older acpi_freq driver on kernels that include support for intel_pstates using the boot-time kernel parameter intel_pstate=disable. ,

The Linux kernel also supports the concept of P-state "governors," which control the aggressiveness of the frequency selection algorithm. With acpi_idle, the ondemand and performance governors were common in server deployments. The performance governor simply requested the max performance at all times, whereas ondemand attempted to change the frequency based on system utilization. Many users experienced performance issues with the ondemand governor, pushing them to use the performance governor (which effectively disabled P-states outside of Turbo). With intel_pstates,

[4]See www.kernel.org/doc/Documentation/cpu-freq/intel-pstate.txt for details.

ondemand is no longer an option. Instead, intel_pstates provides the performance and powersave options. The performance governor operates like it used to—the system will request the max frequency at all times. The powersave governor replaces ondemand and provides a new algorithm (compared to acpi_freq), which is intended to provide power savings without some of the performance drawbacks that were observed with acpi_freq ondemand.

Turbo Ratio Limit

As described in Chapter 2, processors generally have varied maximum Turbo frequencies based on the number of active cores. These limits are fused into each unit and are fixed across a given processor SKU. Users can, at runtime, reduce the maximum allowed Turbo frequencies based on the number of active cores using MSRs 0x1AD, 0x1AE, and 0x1AF. The exact semantics of thee MSRs is processor specific (and summarized in Figure 8-6).

Cores	Sandy Bridge		Ivy Bridge		Haswell	
Active	MSR	Bits	MSR	Bits	MSR	Bits
1	1AD	[7:0]	1AD	[7:0]	1AD	[7:0]
2	1AD	[15:8]	1AD	[15:8]	1AD	[15:8]
3	1AD	[23:16]	1AD	[23:16]	1AD	[23:16]
4	1AD	[31:24]	1AD	[31:24]	1AD	[31:24]
5	1AD	[39:32]	1AD	[39:32]	1AD	[39:32]
6	1AD	[47:40]	1AD	[47:40]	1AD	[47:40]
7	1AD	[55:48]	1AD	[55:48]	1AD	[55:48]
8	1AD	[63:56]	1AD	[63:56]	1AD	[63:56]
9	-	-	1AE	[7:0]	1AE	[7:0]
10	-	-	1AE	[15:8]	1AE	[15:8]
11	-	-	1AE	[23:16]	1AE	[23:16]
12	-	-	1AE	[31:24]	1AE	[31:24]
13	-	-	1AE	[39:32]	1AE	[39:32]
14	-	-	1AE	[47:40]	1AE	[47:40]
15	-	-	1AE	[55:48]	1AE	[55:48]
16	-	-	-	-	1AE	[63:56]
17	-	-	-	-	1AF	[7:0]
18	-	-	-	-	1AF	[15:8]
Semaphore	-	-	1AE	[63]	1AF	[63]

Figure 8-6. *Turbo ratio limit configuration*

Starting with the Sandy Bridge generation, specific Turbo frequencies could be requested directly by the operating system. As a result, controlling Turbo frequencies with these MSRs may not be necessary in all conditions. As an example, if you are simply looking to set a maximum requested frequency with intel_pstates, this can be done with sysfs (as described previously). However, in other OSs where ACPI is used, there may not be a user interface to the OS for this level of control. In such situations, these MSRs can be used.

These MSRs can also be used for fine tuning the Turbo levels based on the number of active cores. However, in practice, such configuration is generally not needed. On Sandy Bridge servers, MSR 0x1AD was sufficient to control the limits. Each byte in the register specified the ratio for a given core count. There was a maximum of 8 cores, so the 64-bit MSR was sufficient.

Ivy Bridge servers support up to 15 cores. MSR 0x1AE was added in order to provide 1 Byte per core. The user was required to first configure MSR 0x1AD and then 0x1AE. When writing to MSR 0x1AE, bit [63] needed to be set to 1 to instruct the hardware to take the configuration.

Haswell servers supported up to 18 cores. MSR 0x1AF was added, and the semaphore bit was moved out to MSR 0x1AF [63].

Note that these MSRs are package-scoped, meaning that one copy is shared across all cores on a given socket. When writing these registers, you should generally perform the writes from one logical processor on each socket in the system.

Turbo Ratio Limit provides a mechanism for users to find a "compromise" between enabling and disabling Turbo. For example, a user may find that using all of Turbo results in undesirable frequency transitions, but disabling Turbo significantly reduces peak performance and throughput. By reducing all maximum Turbo ratios to some level in between P1 and P0*n*, the user may be able to find a sustainable Turbo frequency (on a given workload) that does not generate any frequency transitions. This can be a useful compromise between completely disabling Turbo and using the full Turbo capabilities.

On Haswell processors, the use of AVX instructions could reduce the maximum Turbo frequencies. The user could use Turbo Ratio Limits to set the max Turbo frequencies to match the levels achieved with AVX, providing more consistent frequency as workloads transitioned between AVX sections.

Note that due to scalability concerns, this approach may be discontinued on future processor generations and replaced with an adapted interface.

Uncore Frequency Scaling

Uncore Frequency Scaling (UFS) was introduced on Haswell E5/E7. This feature autonomously controls the frequency of the uncore based on a variety of metrics inside the CPU. UFS not only saves power, but also works to share power with the cores in order to provide higher frequency and improved performance. UFS transitions block all system traffic for about 20 μs today. As a result, these transitions have been tuned to be rare (milliseconds between transitions).

The UFS frequency algorithm can be controlled using the non-architectural MSR 0x620. [7:0] controls the minimum ratio; [15:8] controls the max. With these two fields, the algorithm can be bounded. The internals of the algorithm cannot be configured and may change from generation to generation.

The base (untuned) UFS algorithm has been tuned for performance on most enterprise workloads. A common (and simple) alternative configuration is to set this MSR to 0x3F3F, which will attempt to lock the uncore at the highest frequency allowed by the processor SKU. This configuration is useful for latency-sensitive usage models. Many networking users have also found this configuration to be desirable for their usage models. Note that the TDP frequency assumptions assume that UFS is allowed to be dynamically managed, and locking the frequency at the maximum may result in frequency reduction below P1 on very high-power workloads. In practice, this is not observed on real workloads.

Core C-States

In addition to P-states and frequency controls, Core C-states provide some of the biggest impact CPU power across a range of system utilizations. As described in Chapter 2, it is important to remember that Core C6 can provide a significant performance boost when paired with Turbo by allowing periods of time where only a few threads are active to operate at higher frequencies. This can be valuable in certain parallel workloads where Amdahl's Law[5] is at work.

Unlike P-states, which slow down the rate that instructions are executed, C-states provide power savings at the cost of a "wake-up" when execution is resumed. These wake latencies are typically in the tens of microseconds for C6 and about a microsecond for C1. Just like P-states, this will ultimately manifest itself as an increase in response time.

The core will enter the C1 state when the `HLT` (halt) instruction is executed. This is part of the instruction set and C1 cannot be disabled in hardware. Core C1e can similarly not be directly disabled by hardware. However, there is a configuration bit that the BIOS can set that causes C1 requests to be promoted into C1e requests. This is frequently documented as "C1e Enable," but that definition is not strictly true. C1e on Windows is generally enabled and disabled using this bit as a result of the way that ACPI enumerates C-states to the operating system. This is also the case with older versions of Linux using the ACPI idle driver. The Linux `intel_idle` driver will automatically disabled the promotion of C1 to C1e independent of BIOS configuration so that it can autonomously select between those two states based on the system behavior (C1 is used when a short idle period is predicted, whereas C1e will be used for slightly longer idle periods).

Core C6 is used when `MWAIT(C6)` is executed by the operating system. There is no hardware disable for Core C6 on current processors. The BIOS has the option of selecting which C-states are enumerated to the operating system using ACPI. Different BIOS designers expose this to the customer in different manners. Disabling C6 through the BIOS will effectively disable it on both Windows and older versions of Linux using the `acpi_idle` driver.

Linux with the `intel_idle` driver today will not look at the BIOS configuration when deciding what C-states to use. Because `intel_idle` does not look at ACPI, it is not possible for the BIOS to communicate this information to the driver. Instead, it uses the `intel_idle.max_cstates` kernel parameter to control the level of C-states used. This definition is product- (and even kernel-) specific, so some experimentation may be required. Note that setting this to a value of 0x0 will disable the driver rather than disable C-states, so values greater than 0x0 are recommended

There is also a demotion algorithm that can autonomously decide to use a shallower C-state than what the OS has selected. In general, it is not recommended that users manipulate this configuration.

■ **Note** Today the CPU will not autonomously grant a deeper C-state than the one that the operating system requested.

---

[5]The speed-up of parallel computing is limited by the percentage of a workload that is run in a serial manner.

Runtime Core Disable

C-states can provide a performance boost by providing access to higher Turbo frequencies and also by saving power in a power-constrained environment. Although C-states will autonomously be requested by the operating system if no work is pending for that core, it is also possible to provide a hint to the OS that a given CPU should not be used. This will prevent the OS from scheduling tasks to that core so that it can stay in a deep C-state. Note that this behavior is not 100% robust, because cores can still be woken up in certain cases (e.g., a broadcast thermal interrupt). However, in practice, they tend to be very effective.

On Linux, this is called core offline and can be done on an individual core basis using `sysfs`.

```
#echo 0 > /sys/devices/system/cpu/cpuX/online
```

On Linux, the logical processors to physical core mapping (including sockets and hyperthreads) is technically controlled by the BIOS. The topology can be discovered using `sysfs`:

```
/sys/devices/system/cpu/cpu*/topology/*
```

Although it is recommended that the topology be discovered and decoded, it is useful to understand what to expect in general. In practice, the mapping is mostly consistent across generations and configurations and is best described with an example (see Figure 8-7 for an illustration). Consider a two-socket system with two four-core processors that each support HT. The logical processors that are enumerated in `sysfs` will be enumerated as follows:

- Logical processors 0 to 3 represent the first thread on cores 0 to 3 on socket 0.
- Logical processors 4 to 7 represent the first thread on cores 0 to 3 on socket 1.
- Logical processors 8 to 11 represent the second thread on cores 0 to 3 on socket 0.
- Logical processors 12 to 15 represent the second thread on cores 0 to 3 on socket 1.

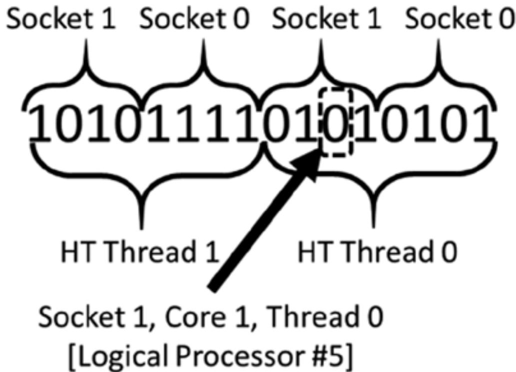

Figure 8-7. *Example logical processor bitmask with two-socket, four-core CPUs on Linux*

Windows has a related feature called C*ore* P*arking*. This feature is a bit different from the offlining support that is available in Linux, and it does not provide a mechanism to force certain cores to stay turned off. Core Parking is no longer recommended in server deployments by Microsoft[6] and has been disabled by default.

Although Windows does not have a way to force specific logical processors to turn off across the operating system, specific applications can be affinitized to a set of logical processors. In conjunction with OS requested C-states, the user can use affinity control to encourage the OS to not use specific cores, allowing them to enter into a deep C-state. The logical processor to physical core mapping can be decoded using the `coreinfo`[7] utility. Affinity control is beyond the scope of this book.

Package C-States

Unlike Core C-states, which are under the control of the operating system, package C-states today are autonomously managed by the CPU. See Chapter 2 for more details on package C-States. Package C-states save additional power when all of the cores are in a deep C-state by taking additional power savings steps that would not be possible if cores were active. These states are generally only possible at very low system utilizations, and their residencies can be monitored with software, as described in Chapter 7.

One of the biggest impacts of package C-states is that they increase the latency for external devices to communicate with DDR memory. In order to access memory, the CPU must wake from the package state, which typically takes tens of microseconds.

When it uses package C-states, the idle power of the Xeon E5 CPU is on the order of 10-15 W. With Core C-states enabled and package C-states disabled, the CPU power will increase by a moderate amount (the amount is dependent on the processor generation). Note that this power increase will be amplified by power delivery efficiency losses (see Chapter 4), since power delivery tends to be less efficient at lower power levels. A rough rule of thumb is a ~1.5–2 times increase.

---

[6]See `http://support.microsoft.com/kb/2814791`.
[7]See `https://technet.microsoft.com/en-us/sysinternals/cc835722.aspx`.

Package C-states can also impact platform power. This impact can be even larger than what is observed on the CPU die. The most notable impact comes from memory self-refresh. If opportunistic self-refresh is disabled, then disabling package C-states will result in memory only using CKE to save power. This can have a large impact on platform idle power savings, particularly on systems with a large memory capacity.

As described earlier in this chapter in the “Idle Workloads” section, one limitation with package C-states is that it is common for the software that continuously runs on them to result in an “active idle” state that is not truly idle. These states are most effective when the system is able to achieve a truly idle state across all sockets in the node. This is different from many consumer usage models, which are aggressively optimized and tuned for idle power due to battery life concerns. In order to take full advantage of idle power savings opportunities, additional work may be required to tune (or stop) software that prevents the system from becoming truly idle.

If a user has a system that spends a notable amount of time at idle, then enabling package C-States can save a notable amount of system power. This is particularly true of systems with large memory capacities. Package C-states are typically controlled by the BIOS. The OS does not have direct control over these states today. It is also possible to disable them by writing 0x0 into MSR 0xE2 [2:0] (CST_CONFIG_CONTROL).

Energy Performance Bias

A number of power management algorithms manage different power and performance tradeoffs in the system. Some of these algorithms are internal and their tunings are not externally visible. Energy Performance Bias (EPB) provides a mechanism for software to provide a hint to the CPU about how the user would like the system to make these tradeoffs. Today, this is an architectural MSR that provides 4 bits to select from up to 16 different operating modes. Smaller values represent more performance whereas larger values save more power. Rather than requiring users to attempt to tune 10 different features that could have different tunings on each project, this feature is intended to provide a simple interface for tuning CPU power management algorithms.

On Xeon E5/E7 processors starting with Sandy Bridge, EPB supported four modes of operation and the two least significant bits of the MSR were ignored. In practice, only two modes have ultimately shown significant value: a Performance mode (EPB values 0 to 3) and a Balanced Performance mode (EPB values 0 to 7). Two deeper modes (Balanced Energy and Energy Efficient) exist, but in practice, it has been difficult to distinguish them from the Balanced Performance mode. Table 8-1 provides an overview of some of the key features that are managed by EPB.

Table 8-1. *EPB Feature Control Summary*

Feature	Performance	Balanced Performance
Turbo demotion	Disabled	Enabled
Memory CKE	Disabled	Enabled
QPI L0p	Disabled	Enabled
Internal algorithms	Performance tunings	Power savings

Dynamic switching is another feature that was introduced on Sandy Bridge E5 that automatically transitions the system into the Performance mode when the CPU detects that high performance is desired. It is recommended that this remain enabled. Some users may desire to override some of these decisions that are typically controlled by EPB and Dynamic EPB Switching, and these limited opportunities will be discussed later in this chapter. Note that dynamic switching will not occur on systems that have disabled Turbo, so selecting the right EPB mode is more important on such systems. Users who configure the system to request Turbo 100% of the time (such as the Windows or Linux `performance` governors) will also switch the system into Performance EPB mode 100% of the time.)

EPB has not been optimized on Atom servers or on Xeon E3 servers like it has been on the E5/E7 product lines. The dynamic switching algorithm is also only productized on E5/E7.

When it was initially architected, EPB was placed in control of the operating system with 0x1B0 IA32_ENERGY_PERF_BIAS. On Windows, the Power Options control panel Performance mode will automatically select the EPB Performance mode. On Linux, the easiest option is to simply write directly to the MSR on all threads. EPB is not managed dynamically on Linux.

On Sandy Bridge, a configuration bit was added so that the BIOS could take control of EPB away from the operating system. This is controlled by the non-architectural MSR 0x1FC (MSR_POWER_CTL). If bit [25] is 0, then the EPB is controlled by the OS and MSR 0x1B0. Otherwise, the OEM is in control. In these cases, EPB may be controlled through some existing BIOS option to further simplify the process for end users.

One of the biggest impacts of EPB is the Turbo demotion algorithm. Turbo demotion has no impact for customers that have disabled Turbo. In such a situation, using the EPB Performance mode may be the best choice. However, such a configuration will result in CKE being disabled, which may be undesirable. Overrides are available for CKE to ensure that it will be used. This will be discussed later in the chapter.

For customers that are making use of Turbo, the Balanced Performance mode has been shown to effectively save power with minimal impact to performance on most workloads due to dynamic switching. One drawback to this situation occurs when there is a spike in system demand. Similar to the operating system, it will take time (tens to hundreds of milliseconds) to detect the spike in demand and react accordingly.

Note that recent versions of the Linux kernel will detect an EPB boot configuration of 0x0 (Performance) and reprogram it automatically to 0x6 (Balanced Performance). As such, you may want to configure this mode using the operating system instead of the BIOS.)

Hyperthreading

Hyperthreading (HT) allows two logical processors to share a single core. By sharing a core, they frequently reduce their thread performance in order to achieve higher overall throughput and higher system performance. The act of enabling HT has almost zero impact on power by itself. However, by having multiple threads share the resources of a core and increase the overall performance and throughput of the core, power is frequently increased. Running a core with HT enabled, but having the scheduler only use one of the two logical processors on that core, will result in an almost identical power consumption to disabling HT.

Although HT will frequently increase the power consumption of a core, it almost always does so in a power efficient manner. It is typically one of the most power efficient performance optimizations that exists; however, some workloads that do not benefit from HT do exist. Workloads that are very sensitive to memory bandwidth are a good example. Such workloads may see a slight efficiency loss by executing them with HT. However, in practice, disabling HT in order to improve power efficiency is very rarely an optimization that pays dividends.

HT can frequently be disabled in the BIOS. It is also possible to instruct Linux not to use HT threads using the same mechanism as core offlining. This provides *effectively* the same behavior as the BIOS disable. The `/affinity` flag can be used in Windows for a similar effect.

Prefetchers

Prefetchers can provide significant performance boosts on certain types of workloads. On others, they provide little value. Some workloads even lose performance with prefetchers, but this has become much less common over the years as memory bandwidth has improved and the state of the art has improved with prefetcher design.

Prefetchers almost always increase the amount of memory bandwidth being consumed by the system. Even workloads that see no performance benefit from prefetching may observe a measurable increase in memory bandwidth. This memory bandwidth will increase platform power. As a result, users may see power efficiency improvements by disabling prefetchers on workloads that exhibit little or no performance upside.

MSR 0x1A4 can be used to configure prefetchers on many server processors. Bits [3:0] are generally mapped to different types of prefetchers in the system. By setting these bits to a 1, the corresponding prefetcher(s) are disabled. Setting all four bits to 0xF is an effective way to disable prefetching. This can be performed at runtime in order to enable easy testing or deployment. Some BIOSes also provide an interface to disable prefetchers and use the same interface. MSR 0x1A4 is not an architectural MSR and therefore may change in the future.

PCIe

The primary power saving feature for PCIe is L1. L1 enabling has generally struggled in the server ecosystem over the years. It saves a relatively small amount of power (on the order of ~1 W for an x8 connection on the CPU, with additional savings on the connected device) at the expense of increased latency for PCIe devices that make use of the feature.

On some Haswell E5 platforms, PCIe L1 has a larger impact on idle power than previously observed on Sandy Bridge and Ivy Bridge. On some platforms, additional platform-level power optimizations depend on all links being in the L1 state. This can change L1 from a 1 W feature into a 10 W feature. Users should experiment with L1 to best understand the tradeoffs on a given system.

ASPM L1 enabling is controlled by the BIOS. However, the operating system can also be used to disable ASPM. Enabling ASPM from the OS if it was not enabled by the BIOS is generally not recommended.

In Linux, ASPM can be disabled using the `pcie_aspm=off` kernel parameter. ASPM can also be disabled using `sysfs`:

```
/sys/modules/pcie_aspm/parameters/policy
```

One can ascertain the current ASPM configuration using the following:

```
> lspci -vvvv | grep ASPM
```

In Windows, you can determine if ASPM is enabled or not with `powercfg /energy` report. ASPM can be configured with `powercfg` (assuming it was enabled by the BIOS). The following will disable ASPM:

```
powercfg -setacvalueindex scheme_current sub_pciexpress aspm 0
```

Customers who are concerned about PCIe latencies should investigate L1 disable. Others who are more power conscious should evaluate ASPM L1, particularly if their deployments are already making use of package C-states.

QPI

QPI supports two main power management features: L1 and L0p (as described in Chapter 3). QPI L1 is only utilized during package C-states and has zero cost due to the fact that the L1 wake-up is done in parallel with other longer latency operations. L0p is used at runtime and does result in some power and performance tradeoffs.

Starting with Ivy Bridge E5/E7, L0p was disabled automatically by dynamic switching or when the EPB performance mode was selected. L0p increases cross-socket data movement latencies by ~10 ns. This can result in a small decrease in performance (up to ~1%). Users that have disabled Turbo and desire maximum performance should consider setting EPB in performance mode.

QPI can be configured to run at lower frequencies. The QPI voltage is fixed and the power savings from running at lower frequencies tends to be relatively small. Bandwidth across the link is directly proportional to the QPI frequency, and there is also a small latency cost (<10 ns) when QPI is run at low frequencies. In practice, the power savings from running QPI at lower frequencies is not worth the performance and flexibility impact.

On some systems (particularly two-socket platforms), multiple links exist between sockets. It is possible to disable links in order to save notable power (on the order of ~10 W at low, but non-idle utilizations) at the expense of a significant decrease in cross-socket communication bandwidth. This can lead to severe performance loss on some workloads. However, a subset of workloads, such as those that primarily execute out of the cache or have extremely good NUMA optimizations and locality, may observe minimal performance loss. By monitoring the QPI link bandwidth, you may be able to determine whether this could be a good opportunity for power savings. It is not possible to enable or disable links at runtime.

Memory

Memory can be a large contributor to overall platform power in deployments with large memory capacities. Memory naturally has a wide power dynamic range, because a large percentage of the power is a function of the memory bandwidth (independent of power management actions like CKE and self-refresh). A good rough estimate of memory power is a simple linear function where both the slope and y-intercept are a function of the type of memory deployed (capacity, ranks, process generation, etc.). Memory power management and the power characteristics of different types of memory technologies are discussed in Chapter 3. In addition to the inherent power scaling that exists in memory, additional CPU-driven power management capabilities can be tuned and configured to provide different power and performance characteristics.

As a general rule, users with small memory capacities likely need not get overly aggressive with memory power management, whereas those who are loading up the DIMM slots should carefully consider their options here. “Small” is a difficult term to formally define in this case. With the transition to DDR4, power consumption has significantly improved, making aggressive power management somewhat less important. On low-power servers with TDPs that are less than about 50 W, the contribution of memory power is proportionally larger, and a quick look at the memory power management configuration is advisable. For higher power servers, users who are deploying x4 devices in 2DPC configurations (or larger) should evaluate their options. Users with small deployments, such as a 1DPC x8 population, may want to look at other opportunities for power savings first.

CKE

CKE provides moderate levels of power savings with minimal performance penalty. As a result, it is commonly used while applications are active to save power during short idle periods. Using CKE while performance benchmarking generally results in a peak performance decrease of ~1% and a minimal impact on response times because it only costs ~10 ns to wake up memory from this state. Memory latency benchmarks suffer from CKE though, particularly because the PPD variety forces a page close. This latency increase is not representative of the performance impact on real workloads.

On systems that support dynamic switching and EPB (such as Xeon E5/E7 class servers starting with Sandy Bridge), CKE is disabled automatically in the Performance mode (or after a dynamic switch). Using EPB Balanced Performance with dynamic switching enabled is effective at avoiding the 1% peak throughput decrease while saving power at lower system utilizations for systems that have Turbo enabled and have not configured their OS to request max frequency 100% of the time. Users who configure their system to use EPB Performance mode or request maximum frequency 100% of the time should consider enabling CKE at all times if they have a large memory topology and are willing to trade off ~1% peak throughput. This configuration must be done through the BIOS, and different BIOS OEMs may make such a configuration available through different mechanisms.

Some platforms have provided options for both APD and PPD modes.[8] In practice, the difference between these two modes of operation is not a first order impact on either power or performance. APD, for example, sounds like it should have slightly better performance characteristics (at slightly higher power), but in practice, it is largely not significant. As a result, tuning these modes is only recommended for users who are looking to fine-tune their system to a specific workload, and only after they exhaust other tuning opportunities.

On Avoton/Rangeley, EPB and dynamic switching do not control CKE. It must be statically configured by the BIOS. Due to the low SoC TDP power of these systems, smaller memory capacities will have a larger contribution to platform power, and users should consider their options with respect to CKE. One big exception here is with storage deployments that have a large number of HDDs. In this case, the memory power is frequently dwarfed by the drive power, making CKE less important overall.

Self-Refresh

Self-refresh provides much larger power savings than CKE at the cost of significant wake-up latencies (generally on the order of a few microseconds). As a result, it is typically targeted at large idle periods, such as when all cores are asleep. Aggressive use of self-refresh can result in performance loss and even an energy increase for completing a set of work.

Conceptually, there are two ways to use self-refresh. First, it can be used during deep package C-states. In this situation, the latency cost is effectively zero, since the wake-up can be done in parallel with other actions. We will refer to this as *ForceSR* for simplicity. Secondly, it can be used outside of package C-states. This is referred to as *Opportunistic Self-Refresh (OSR)*. Because self-refresh is typically "free" during package C-states, controls for this capability are not made available for the user. OSR configuration is separate from the Package State configuration.

Unlike CKE, OSR (and Force SR) is not automatically disabled by EPB Performance or dynamic switching. This decision must be made in the BIOS. By default, OSR is configured to be very unaggressive. It is intended to target very long idle periods where package C-states are not active, such as in a system with very good NUMA optimizations or when package C-states have been disabled. It can also be useful in saving power in multi-socket systems at idle that are running software that prevents high package C-states residency but does not actually frequently access memory on all channels/sockets during the spurious events that are preventing package C-states. Tuning the aggressiveness of the algorithm is possible through the BIOS, but this is not recommended.

Recall from Chapter 3 that the CK behavior is an option for how deep of a self-refresh to use. The clocks can stay active, which significantly shortens the wake latency but also reduces the power savings effectiveness. In practice, self-refresh with the CK active has minimal power savings benefits over CKE. As a result, this is the option that is less interesting than the Enable/Disable decision.

In practice, for most users, the base OSR configuration is effective at saving power without an observable impact to responsiveness or throughput. However, latency-sensitive users who are concerned about block times on the order of ~5-10 μs should disable OSR in addition to package C-states.

[8] Haswell E5/E7 only productized the APD state.

Patrol Scrub

Patrol Scrub is a reliability feature that steps through memory-reading each line and writing it back. The intention is to detect correctable errors (and correct them) before they can degrade into uncorrectable errors. The bandwidth cost of Patrol Scrub is quite small and does not materially impact the power of the system at runtime. However, Patrol Scrub does prevent OSR. Only a subset of channels on a socket is scrubbed at a time, though, so OSR is still possible on the remaining channels. Patrol Scrub is a very effective feature for avoiding uncorrectable errors, and it is not recommended that it be disabled. If a user observes that certain channels just won't enter OSR, the likely explanation is Patrol Scrub. Package C-states have been optimized to provide self-refresh (across all channels) while also maintaining a good average scrub rate.

NIC

Chapter 4 discusses some of the network interface card (NIC) power management options. Table 8-2 provides an overview of these capabilities. NICs can be connected into any system, and therefore the power management configuration is managed by the driver and not by the BIOS. This configuration can typically be managed at runtime.

Table 8-2. *NIC Optimization Summary*

Feature	Potential Savings	Cost	Description
Media Speed	Up to a few watts	Significant throughput loss and latency increase potential	The speed of NIC links can sometimes be reconfigured to save power at the expense of bandwidth. This can be done at runtime by software drivers, but it takes significant time to do so (during which time the network connection is blocked) and therefore it cannot be performed aggressively.
Energy Efficient Ethernet	~400 mW to ~2 W	<~16 μs	Idle power management feature for the network.
Interrupt Moderation	Platform dependent	Configurable, typically ~100-200 μs	Rate limits delivery of interrupts to the CPU, frequently resulting in power savings, improved throughput, or both, at the expense of latency.

Interrupt moderation is one of the most interesting features for the NIC. It is common for users to concern themselves with features like P-states, which can induce latency bubbles on the order of 10 microseconds, while ignoring interrupt moderation, which can cause 10 times larger latency increases. Although this feature does increase latency, it is effective at both saving power and improving network throughput. Latency-sensitive users should consider changing the latency tuning parameter or even disabling this feature, but they should be aware that this will increase the demand on the CPU.)

DMA coalescing is another NIC feature. This feature has shown minimal effectiveness in server deployments. It is not enabled by default and is not generally recommended in servers.

Storage

Storage power management is made up of two components: the storage controller PHY and the device (HDD/SSD). These power management capabilities are standardized across both server storage subsystems as well as those found in consumer devices, and are discussed in Chapter 4.

As described in Chapter 4, SATA devices support four power savings modes: Working, Idle, Standby, and Sleep. Both Standby and Sleep can take significant time (seconds) to wake, and therefore they may not be desirable for server deployments. The Idle state, on the other hand, has minimal latency costs, and therefore it can be left enabled. For storage deployments where long latency wake-ups are acceptable, allowing the Standby state, or explicitly using the Sleep state, can help achieve very low idle power. Both SATA and SAS drives can enter a power management state either autonomously (after a configurable timer expires) or based on a command from the host. Table 8-3 provides an overview of this control. Remember that with HDDs, power consumption is heavily dependent on the use of power management actions, whereas with SSDs, power will automatically scale with bandwidth consumption.

Table 8-3. *SATA/SAS Drive Power Management Configuration*

Type	Host Request	Timer Configuration
SATA	Set Features: Go To Power Condition	Set Features: Extended Power Condition
SAS	Start/Stop Unit (SSU) command	Power Condition Mode page

In Linux, the `sdparm` and `hdparm` tools can be used to control these options. The `sdparn --all` command is useful for discovering if the drive supports mode pages for power management. Not all drives support the full set of configuration options, and the `-enumerate` flag is useful for exploring common mode pages (independent of a specific drive). For `hdparm`, the `-B`, `-S`, and `-M` flags can be used to control the power management aggressiveness of some drives. Other flags exist for requesting a drive to enter into a low-power state.

Smartmontools is another set of utilities that can be used to manage drives.[9] `smartctl` is one frequently-used utility that is part of this package. These utilities are commonly available on Linux but can also be used on Windows.

In Windows, the idle time before spinning down a drive can be set with `powercfg` (setting it to a value of 0 will disable putting the drive to sleep):

```
powercfg /Change disk-timeout-ac 15
```

The SATA Aggressive Link Power Management (ALPM) power state of *Partial* is able to achieve low idle power (~100 mW) with a wake-up latency of <10 μs. The deeper states have minimal power savings benefits in most servers and can cause significant latency. Users who are very latency sensitive can consider disabling all these states, but Partial provides a good compromise between latency and power savings.

In Linux, the tlp and tuned-adm tools are available (depending on the distro) for managing a wide range of power management options, including link power management.[10]

■ **Note** The tlp and tuned-adm utilities are useful for managing a variety of power management options in Linux.

In Windows, the link power management is controlled by `powercfg` in the *disk* subgroup. These options are hidden (not named) today. You can explore the current configuration using the following command. By doing this, you can ascertain the GUID associated with the hidden features and configure it in a manner similar to some of the previous command lines.

```
powercfg -qh scheme_current sub_disk
```

Thermal Management

CPU thermal management is configured to protect the system and is not tunable by consumers. Platform thermal management is managed be proprietary OEM algorithms, and therefore the specifics are beyond the scope of this book. OEMs commonly make different thermal management options available. More aggressive thermal management algorithms result in higher system temperatures and the potential for brief thermal throttling events and slight reductions in Turbo performance. In practice, these algorithms can save significant platform power without materially impacting the performance of the system.

[9]See www.smartmontools.org.
[10]See http://linrunner.de/en/tlp/tlp.html.

Cooling a processor to very low temperatures tends to be cost prohibitive and unnecessary. Lower temperatures save leakage power and thereby improve Turbo performance of some workloads, but this effect is exponential and the benefits decrease rapidly as temperatures drops. The decrease in CPU leakage power typically is much smaller than the increase in cooling power.

Optimization at a Glance

This chapter has discussed a wide variety of different optimization opportunities, and it can be daunting to determine where to begin. What features should be enabled? Which should be disabled? Different users have different constraints and goals for what they are trying to achieve. This section provides a high-level summary of the various features that have been discussed. Before the optimizations are discussed, Table 8-4 provides a summary of some of the different types of impacts that these optimizations can have on the system. Tables 8-5 through 8-8 summarize a selection of the optimization opportunities, including the priority in which users may want to focus their efforts first.

Table 8-4. *Performance Metrics*

Metric	Description
Response time	Average time to complete a request from an external agent. Users of transaction processing workloads should note these.
Peak throughput	The feature may reduce (or improve) the peak throughput of the system. This could also be called "peak performance."
Execution block	This feature may result in short periods of time (microseconds, unless noted otherwise) in which core execution may be blocked.
Device block	This feature may result in short periods of time (microseconds, unless noted otherwise) in which an external device may be blocked from access to memory.

Table 8-5. *CPU Optimization Summary*

Feature	Power Impact	Primary Performance Impacts	Utilization Targets	Control	Priority
P-states	Tens of watts	Response time,execution block,device block	Low to moderate	OS, BIOS	High
Turbo	Tens of atts	Peak throughput, execution block, device block	High	OS, BIOS	High
UFS	Tens of watts	Latency blocks, execution block, device block	Focus on low utilizations	BIOS, MSR	Medium
Core C-states	Tens of watts	Response time, execution block	Low to moderate	BIOS, OS	High
Package C-states	Watts	Response time, execution block	Idle	BIOS, MSR	High
Core Disable	Watts	Peak throughput	All	BIOS, OS	Medium
EPB	Tens of watts	Response time, peak throughput	Moderate to high	BIOS, OS	High
HT	Function of performance change	Peak throughput	High	BIOS, OS	Medium
Prefetchers	Workload dependent	Peak throughput	All	BIOS, MSR	Medium
QPI frequency	Minimal	Peak throughput	All	BIOS	Low
QPI link disable	Watts	Peak throughput	All	BIOS	Low
PCIe ASPM L1	Watts	Response time, device block	Low	BIOS, OS	Medium

On many systems, the CPU is a significant component of the overall platform power. Table 8-5 provides some highlights from the CPU power optimizations that have been discussed. Note that the power impact shown here is for a typical high-TDP Xeon E5/E7 system. Not all systems will exhibit these impacts; the details are discussed in earlier sections.

Systems with large memory capacities or low-power CPUs may have large contributions for memory power. Table 8-6 provides a summary of some of the optimizations available for memory power savings. Users who are very latency sensitive will want to disable OSR, but typical users likely will not be exposed to it.

Table 8-6. *Memory Optimization Summary*

Feature	Primary Performance Impact	Utilization Targets	Control	Priority
CKE Enable	Peak throughput	All	BIOS	Medium
OSR	Response time, execution block, device block	Idle	BIOS	Low

Networking cards themselves do not contribute a large percentage of platform power in most systems. However, Interrupt Moderation is a key feature that provides tradeoffs between CPU utilization, power consumption, and latency. Tuning this feature to suit a user's needs can have a significant impact on the power/performance/latency characteristics of a system.) Table 8-7 provides a summary of some of the key NIC optimizations.

Table 8-7. *NIC Optimization Summary*

Feature	Power Impact	Primary Performance Impacts	Utilization Targets	Control	Priority
Media speed	Watts	Peak throughput, response time	All	Driver	Low
EET	Watts	Response time, device block	Idle	Driver	Low
Interrupt moderation	Watts to tens of watts	Peak throughput, response time, device block	All	Driver	High

In compute servers with only one or two drives, storage power is generally not a large contributor to overall platform power. In storage nodes, on the other hand, HDD and/or SDD power can dominate the power consumption of the platform. Table 8-8 provides a summary of the storage power optimization opportunities for a storage node. For compute nodes, it may be desirable to disable or "turn down" many of these features. Some can add considerable latency with minimal power savings. The Slumber state, for example, saves minimal power in a server environment but can result in millisecond wake latencies.

Table 8-8. *Storage Optimization Summary*

Feature	Power Impact (per drive)	Primary Performance Impacts	Utilization Targets	Control	Priority
PHY power state (Partial/Slumber)	Hundreds of mW	Response time, execution block, peak throughput	Idle	Driver	Medium
Device power savings	Watts	Response time, execution block, peak throughput	Idle	Driver	Medium

Summary

Power optimization can have a significant impact on both power consumption and performance in a platform. Users should start by characterizing their system behavior and power consumption so that they can decide which areas to focus on. There is no need to spend weeks attempting to optimize drive power if it is only consuming 5% of the overall platform power. Next, users need to identify a repeatable workload that can be used to best understand the power and performance tradeoffs of different optimizations. Once this is complete, users can identify targeted experiments on specific configuration changes based on the guidance provided in this chapter, and they can then identify the optimal configuration based on the their constraints and goals.

Finally, there are a few key things to remember when performing optimizations:

- P-states need not be an all-or-nothing decision. Enabling a small to moderate frequency range can save significant power with minimal exposure to performance problems. These decisions are best made in the OS and not with the BIOS.
- Higher frequencies can, in many cases, provide better platform power efficiency.
- C-states can provide significant performance improvements when paired with Turbo in addition to saving power.
- Interrupt moderation tunings can have a significant impact on response time, power efficiency, and throughput. Improved throughput and better power efficiency can be traded for faster response times.