Unit 6 – Design and Optimization of Embedded Real-Time Systems

Performance Optimization

- Designers must deal with increasing levels of complexity in both the targeted applications and the underlying embedded systems executing the applications. This is in part due to the level of integration and faster speeds of current embedded systems.
- Focusing on one aspect (e.g.: execution time) is not enough anymore. Performance as measured by power utilization must also be considered.
- The optimization process must integrate tuning steps for single core performance, multi-core performance, and power performance.
- Reference: Lori Matassa, Max Domeika, *Break Away with Intel Atom processors: A Guide to Architecture Migration, Intel Press*, 2012.

Optimization Process for Multi-core Processors

- Applications' performance is one aspect of software design; however, correctness and stability are usually prerequisite to extensive performance tuning efforts.
- Typical development cycle: design, implementation, debugging, and tuning. This is an iterative process that concludes when performance and stability requirements are met.
  - ✓ Tuning Phase: single-core optimization, multi-core optimization, and power optimization

SINGLE-PROCESSOR CORE TUNING

- Optimization of the application on one Intel Atom processor core. Focus is on increasing performance, which is typically the reduction of execution time.
- This tuning step isolates the behavior of the application from more complicated interactions with other threads or processes in the system.
- Foundations of performance tuning:
  - ✓ Pareto principle (80/20 rule): 80% of the time spent on application is in 20% of the code.
    - This observation helps prioritize optimization efforts to the areas of highest impact ≡ the most frequently executed portions of the code.
  - ✓ Amdhal's law: It provides guidance on the limits of optimization. If optimization can only be applied to 75% of the application, the maximum theoretical speedup is 4 times ($P = 0.75, N \to \infty$).
    - $S(N) = \frac{Serial\ execution\ time}{Parallel\ execution\ time} = \frac{1}{1-P+\frac{P}{N}}$, $\quad S(N)|_{N\to\infty} = \frac{1}{1-P}$
      - P: proportion of program that can be made parallel. $1-P$: proportion that remains serial, $P \in [0,1]$.
      - N: number of threads across which the parallel portion is split.
      - $S(N)$: theoretical speed up of the execution of the whole task.
    - There is always some fraction of the total operation that is inherently sequential and cannot be parallelized. This includes reading data, setting up calculations, control logic, storing results.

- Summary of tuning process:
  - ✓ Benchmark: Develop a benchmark that represents typical application usage.
  - ✓ Profile: Analyze and understand the architecture of the application.
  - ✓ Compiler optimization: Use aggressive optimizations if possible.
  - ✓ General microarchitecture tuning: Tune based upon insight from general performance analysis statistics that can be employed regardless of the underlying architecture.
  - ✓ Intel Atom processor tuning: Tune based on insight about the Intel Atom processor. These include statistics and techniques to isolate performance issues specific to the Intel Atom processor.

MULTI-CORE TUNING

- Optimization of the application taking advantage of parallel technology including SMT (simultaneous multi-threading, i.e., Intel Hyper-Threading Technology) and multiple processor cores. Focus is on increasing performance, which is typically the reduction of execution time.
- At the application level, two techniques allow you to take advantage of multiple processor cores:
  - ✓ Multitasking: execution of multiple OS processes on a system. Here, the OS governs much of the policy of execution.
  - ✓ Multithreading: execution of multiple threads. Assumes memory is shared. This is much more in control of the software developer.
- Tuning for multi-core processors is more complex due to possible thread interactions and shared cache issues. It is possible for one thread to cause another to miss in the cache on every access.

- Converting a serial application to take advantage of multi-threading requires an approach that uses a development cycle that consists of five phases:
  - ✓ Analysis: develop a benchmark that represents typical system usage and comprised by concurrent execution of processes and threads. Use a performance profiler (e.g.: Intel® VTune™ Performance Analyzer) to identify the performance hotspots in the critical path. Determine if these computations can be executed independently.
  - ✓ Design: Determine changes required to accommodate a threading paradigm (data/code restructuring) by characterizing the application threading model. Identify which variables must be shared and if the current design structure is a good candidate for sharing.
  - ✓ Implementation: Convert the design into code based on the selected threading model. Make use of the multithreading software development methodologies and tools.
  - ✓ Debug: Use runtime debugging and thread analysis tools such as Intel® Thread Checker.
  - ✓ Tune: Tune for concurrent execution on multiple cores with and without SMT.

POWER TUNING
- Optimization of the application focusing on power utilization. Focus is on reducing the amount of power used in accomplishing the same amount of work.
- This is a relatively new addition to the optimization process for Intel architecture processors.

Basics:
- At a fundamental level, power is a measure of the number of watts consumed in driving an embedded system.
- Power can be consumed by several components in a system: memory, hard drives, solid state drives, communications.
  - ✓ Two largest consumers of power in an embedded computing system: display and processor
- Power management features already exist in many OSes. They enable implementation of power policy where various components are powered down when idle for long periods, e.g.: turning off the display after a few minutes of idle activity.
- Power policy can also govern behavior based upon available power sources, e.g.: the embedded system may default to a low level of display brightness when powered by battery as opposed to being plugged into an outlet.

- Characterizing the power used by a system:
  - ✓ Thermal design power (TDP): Maximum amount of heat that a thermal solution must be able to dissipate from the processor so that the processor operates under normal operating conditions. Typically measured in watts.
  - ✓ "Plug load power": Power (watts) drawn from an outlet as the embedded system executes.
  - ✓ Battery power draw: An estimate of power (watts) drawn from a battery as the embedded system executes.

- Your project requirements will guide which of these power measurements to employ and what goals will be set with regards to them.
- The Intel Atom processor enables a number of power states:
  - ✓ C-states: different levels of processor activity. They range from C0 (processor fully active) to C6 (processor completely idle and many portions of the processor are powered down).
  - ✓ P-states: known as performance states, they are different levels of processor frequency and voltage.

Power measurement tools
- Two categories of tools to measure power utilization in an embedded system:
  - ✓ First category: direct measurement (employs physical probes). These probes can be as simple as a plug load power probe between the device and the outlet. Or it could require more extensive probes placed on the system board monitoring various power rails.
  - ✓ Second category: power state profiling tools, employs an indirect method of measuring power utilization. These class of tool measures and reports on the amount of time spent in different power states. The objective is to understand what activities are causing the processor to enter C0 and to minimize them.

POWER AND PERFORMANCE ANALYSIS TOOLS OVERVIEW

SINGLE CORE PERFORMANCE TOOLS

- They provide:
  - ✓ Insight on how an application is behaving as it executes on one processor core.
  - ✓ Different views on the application ranging from how the application interacts with other processes in the system to how the application affects the processor microarchitecture.

- Categories (or types):
  - ✓ ***System profilers***: Provide a summary of execution time across processes on the system.
  - ✓ ***Application profilers***: Provide a summary of execution times at the function level of the application.
  - ✓ ***Microarchitecture profilers***: Provide a summary of processor events across applications and functions executing on the system.

- How is data viewed?
  - ✓ ***Flat profile***: How much time your program spent in each function and how many times that function was called. All in all, it will tell you which functions burn most of the cycles. It also shows the correlation of processes and functions with the amount of time the profiler recorded in each. It does not show relationships between the processes and functions listed with any other processes or functions executing on the system.
  - ✓ ***Call graph profile***: It shows relationships between processes and functions. For each function, it shows which functions called it, which other functions it called, and how many times. There is also an estimate of how much time was spent in the subroutines of each function. It also shows contributions to the measured times between the caller functions and the called functions. This can suggest places where you might want to try to eliminate function calls that use a lot of time.

- Profilers obtain information by sampling or tracing the system while the application is executing. Three techniques:
  - ✓ ***OS-provided API***: OS provides capability to periodically sample and record information on executing processes.
  - ✓ ***Software instrumentation***: Application has code added to trace and record statistics.
  - ✓ ***Hardware performance monitoring counters***: Employed by microarchitecture profilers. Provides information on microarchitecture events such as branch mispredictions and cache misses.

- Table I describes several tools used in single core performance analysis.

TABLE I. SINGLE CORE PERFORMANCE TOOLS

Tool	Type	Description
Sysprof	System profiler	Easy to use, start and stop profiler. Linux-hosted and targeted profiler. Provides system-wide flat profile and call graph information if debug information is present
GNU gprof	Application profiler	Ubiquitous, widely available single application profiler. Requires recompilation to add instrumentation. Provides flat profile and call graph profile.
Oprofile	Microarchitecture profiler	Linux-targeted microarchitecture profiler. Enables event-based sampling using hardware performance monitoring counters.
Intel® VTune™ Performance Analyzer	Microarchitecture profiler	Windows and Linux-targeted microarchitecture profiler. Enables event-based sampling using hardware performance monitoring counters. Powerful GUI enables easy visualization.
Intel® Performance Tuning Utility	Microarchitecture profiler	Similar to VTune™ Performance Analyzer with enhanced event-based profiling features. Basic block view

SYSTEM PROFILING: ***Sysprof***

- System-wide profiler in Linux. It is designed to profile all processes in the systems. It provides information across the kernel and user level processes. Very useful in a scenario with lots of interconnected components.
- It helps in finding the functions in which a program spends most of its time.

Installation:	`sudo apt-get update` `sudo apt-get install sysprof`
Usage:	`sudo -i` `sysprof`

- If an application is being profiled , it must be started independently of ***Sysprof***. The application itself does not require special instrumentation.
- Fig. 1 shows the user interface.
  - ✓ To begin profiling, press the Start button.
  - ✓ To stop profiling (and show the collected results), click on the `Profile` button.
  - ✓ Three sections:
    - Functions (top left): list of functions where the greatest amount of time was measured during profiling. Time per individual function includes the time spent in any function called as a result of the function, such as descendants in the call chain. Time is reported in two forms:
      - Self-time: execution time inside the function, does not include called functions.
      - Total time: amount of time inside the function and all its descendants in the call chain.
    - Callers (parents): List of functions that call the highlighted function in the Functions box. The time listed here is relative to the highlighted function: the sum of self-times (or total-times) of the callers is equal to the self-time (or total-time) of the highlighted function in the Function box.
      - Self-time: time spent in the caller function.
      - Total time: time spent calling the highlighted function (on the Function box).
    - Descendants (children): It shows a portion of the call graph of the highlighted function. To follow a path of the graph call further, click on the triangle (▸), which will show another level of the call graph. At each node of the call graph that represents the function, time is reported for it in self-time and cumulative time.

- Self-time: already described.
- Cumulative time: time in the function and all of its descendants and it is a fraction of the time spent by a caller higher in the call graph.

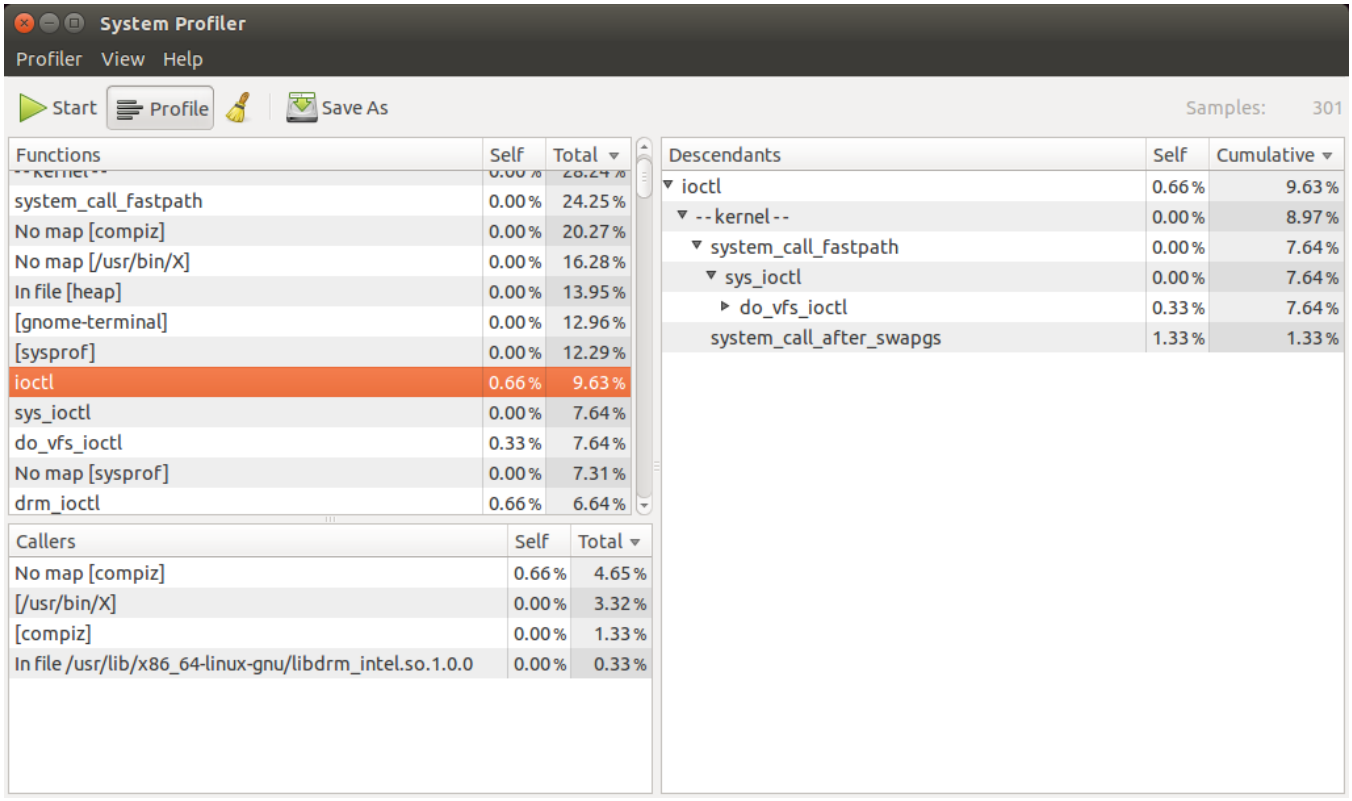

Figure 1. Sysprof example

APPLICATION PROFILING: *GNU gprof*

- Application-based profiling tool that serves as the output and reporting tool for applications that have been compiled and instrumented by `-pg` option (supported by GNU `gcc`, it results in instrumentation being added to the application to collect profile information).
- The instrumented application generates a profile data file (`gmon.out` is the default profile name) when executed.
- We can then use *GNU gprof* to process the profile data and generate reports such as ordered listing of the functions that consume the largest amount of execution time.

Operation

- *GNU gprof* is usually already installed in Ubuntu Linux OS.
  - ✓ To double-check that it is installed, in the Terminal type: `gprof` ↵
  - ✓ If you get an error like: `a.out No such file or directory`, then this would mean that the tool is already installed.
  - ✓ Otherwise, you can install with the following command: `apt-get install binutils` ↵

- For testing, we use the `tst_gprof.c` application, which is specifically written to explain *gprof* usage. It includes the `main()` function and calls to two other functions. One of these functions include a call to other function as well (see Fig. 2).

```
#include<stdio.h>
void func4(void) {
   int count;
   printf("\n Inside func4() \n");
   for (count=0;count<=0XFFFF;count++);
}

void func3(void) {
   int count;
   printf("\n Inside func3() \n");
   for (count=0;count<=0XFFFFFFF;count++);
}

void func2(void) {
   int count;
   printf("\n Inside func2() \n");
```

```
   for(count=0;count<=0XFFF;count++);
   func3();
}

void func1(void) {
   int count;
   printf("\n Inside func1() \n");
   for(count=0;count<=0XFFFFFF;count++);
   func2();
}

int main(void) {
   int count;

   printf("\n main() starts...\n");
   for(count=0;count<=0XFFFFF;count++);

   func1();
   func4();
   printf("\n main() ends...\n");
   return 0;
}
```

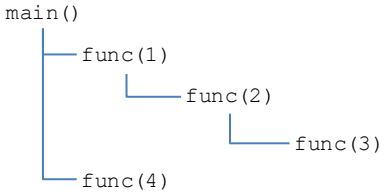

Figure 2. *tst_gprof* application: relationship between the functions.

- ✓ Compilation: `gcc -Wall -pg tst_gprof.c -o tst_gprof` ↵
  - This will generate the binary file named `tst_gprof`. Instrumentation was added to the application.
- ✓ Execute the program: `./tst_gprof` ↵
  - Generated file: `gmon.out`. This file contains all the information that ***gprof*** needs to generate human-readable profiling data.
- ✓ Use the ***gprof*** tool: `gprof {executable name} gmon.out > {name of text file with the profiling data}`
  - `gprof tst_gprof gmon_out > profile.txt` ↵

- The information on the generated text file is divided into a ***flat profile*** and a ***Call graph profile***.
  * If the program does not have any functions (just `main()`), nothing will be shown.
  - ✓ ***Flat profile***: (Dell Inspiron laptop): The description of each column is available in the generated text file.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 95.52      0.87     0.87        1   869.25   869.25  func3
  5.55      0.92     0.05        1    50.54   919.78  func1
  0.00      0.92     0.00        1     0.00   869.25  func2
  0.00      0.92     0.00        1     0.00     0.00  func4
```

  - `% time`: percentage of the total running time of the program used by this function.
  - `cumulative seconds`: A running sum of the # of seconds accounted for by this function and those listed above it.
  - `self seconds`: number of seconds accounted by this function alone.
  - `calls`: number of times this function was invoked.
  - `self ms/call`: average number of ms spent in this function alone per call.
  - `total ms/call`: average number of ms spent in this function and its descendants per call.
  - `name`: function name.

  - ✓ ***Call graph profile***: (Dell Inspiron laptop): The meaning of each item is available in the generated text file.

```
granularity: each sample hit covers 2 byte(s) for 1.09% of 0.92 seconds
index % time    self  children    called     name
                0.05    0.87       1/1          main [2]
[1]    100.0    0.05    0.87       1          func1 [1]
                0.00    0.87       1/1          func2 [3]
-----------------------------------------------
                                               <spontaneous>
[2]    100.0    0.00    0.92                 main [2]
                0.05    0.87       1/1          func1 [1]
                0.00    0.00       1/1          func4 [5]
-----------------------------------------------
                0.00    0.87       1/1          func1 [1]
[3]     94.5    0.00    0.87       1          func2 [3]
                0.87    0.00       1/1          func3 [4]
-----------------------------------------------
                0.87    0.00       1/1          func2 [3]
[4]     94.5    0.87    0.00       1          func3 [4]
-----------------------------------------------
                0.00    0.00       1/1          main [2]
[5]      0.0    0.00    0.00       1          func4 [5]
-----------------------------------------------
```

- The table describes the call tree of the program, sorted by the total amount of time spent in each function and its children. The lines full of dashes divide the table into entries, one for each function. Each entry has a unique index `[i]` next to every function name. Each entry includes several lines:
  - Current function: Line with the index number on the left.
  - Function's parents (function that call the current function): Lines above current function
  - Function's children (function that this function calls): Lines below current function.

- For the current function, the associated line lists the following fields:
  - `% time`: Percentage of the total time spent in this function and its children.
  - `self`: Total amount of time (s) spent solely in this function.
  - `children`: Total amount of time propagated into this function by its children (time spent by the children).
  - `called`: # of times the function was called. If function called recursively: non-recursive calls+ # of recursive calls
  - `name`: name of the current function.

- For the function's parents, the associated lines list the following fields:
  - `self`: Amount of time that was propagated directly from the function into this parent (time spent by the child)
  - `children`: Amount of time that was propagated from the function's children into this parent (time spent by the child's children).
  - `called`: Number of times this parent called the function '`/`' the total number the times the function was called. Recursive calls are not included in the number after '`/`'.
  - `name`: Name of the parent along with the parent's index number. If the parents of the function cannot be determined, the word '`<spontaneous>`' is printed in this field and all the other fields are blank.

- For the function's children, the associated lines list the following fields for each child.:
  - `self`: Amount of time that was propagated directly from the child into the function (time spent solely by the child)
  - `children`: Amount of time that was propagated from the child's children to the function (time spent by the child's children).
  - `called`: Number of times the function called this child '`/`' the total number the times the child was called. Recursive calls by the child are not listed after '`/`'.
  - `name`: Name of the child.

- For the given example, the total time (100%) is the time spent in `main()`. Time spent in each function is (note that most of the time is spent in `func3`):
  - `func1`: 100%. Here, time spent in `func2` is 94.5%, and time spent in `func3` is 94.5%
  - `func4`: 0%. It is important to note that there is time spent in `func4`, however the resolution of the results is such that the total amount of time in `func4` appears as 0.00 s (0%).

- For a full list of *gprof* options, go to the documentation.

MICROARCHITECTURE PROFILING: *Oprofile*

- Command line-based profiler that provides access to the performance monitoring counters.
- *Oprofile* targets Linux systems and requires a kernel driver that acts as a daemon to collect the profile information.
- No instrumentation of recompilation of the applications is required.
- Command-line utilities that comprise *Oprofile*:
  - ✓ `opcontrol`. Configures the collector, initiates and terminates collection
  - ✓ `opreport`. Displays profile, merging available symbolic information where possible.
  - ✓ `opannotate`. Displays profile information correlated with source and assembly code.
  - ✓ `oparchive`. Saves profile for offline viewing and analysis.
  - ✓ `opgprof`. Translates profile into *gprof*-compatible file.
- Table II summarizes the steps for employing *Oprofile* to collect and output a profile of an application reporting clock cycle information. These commands should be executed with root privileges.

TABLE II. OPROFILE PROFILE GENERATION

Step	Command line or Description
1. Initialize the *oprofile* daemon	`opcontrol -init`
2. Configure profile collection	`opcontrol -setup -event="default"`
3. Start profile collection	`opcontrol -start`
4. Start activity	Begin activity to profile
5. Stop profile collection	`opcontrol -stop`
6. Produce report	`opreport -g -symbols`

MULTI CORE PERFORMANCE TOOLS

- Specific tools for analyzing performance related to multi-core processor are still somewhat few in number.
- System profilers can provide information on processes executing on a system; however, interactions in terms of messages and coordination between processes are not visible.

INTEL® THREAD PROFILER

- It identifies thread-related performance issues and is capable of analyzing OpenMP and POSIX multi-threaded applications.
- Key capabilities when profiling an application:
  - ✓ Display of a histogram of aggregate data on time spent in serial of parallel regions.
  - ✓ Display of a histogram of time spent accessing locks, in critical regions, or with threads waiting at implicit barriers for other threads.
- Intel® Thread Profiler employs ***critical path analysis*** where events are recorded including spawning new threads, joining terminated threads, holding synchronization objects, waiting for synchronization objects to be released, and waiting for external events.
- An execution flow is created that is the execution through an application by a thread, an each of the listed events above can split or terminate the flow.
- Critical path: longest flow through the execution from the start of the application until it terminates. Any improvements in threaded performance along this path would increase overall performance of the application.

CRITICALBLUE PRISM

- Prism's analyses are based on dynamic tracing approach: traces of the user's software application are extracted either from a simulator of the underlying processor core or via an instrumentation approach where the application is dynamically instrumented to produce the required data. Once a trace has been loaded into Prism the user can start to analyze the application behavior in a multi-core context.
- Prism provides the user with specific insight relevant in a multi-core processor context. Examples of the views and analysis available in Prism:
  - ✓ Histogram showing activity over time by individual function and memory.
  - ✓ Dynamic call graph showing function inter-relationships and frequency.
  - ✓ Data dependency analysis between functions on sequential code.
  - ✓ "what-if scheduling" feature to explore the impact of: i) executing functions in separate threads, ii) varying the numbers of processor cores employed, iii) removing identified data dependencies, and iv) cache misses on multi-core execution performance.
    - This feature can be used to explore the benefit of Intel® Hyper-Threading Technology on multi-core execution performance.
  - ✓ Data race analysis between functions on multithreaded code.

POWER PERFORMANCE TOOLS

- Two types:
  - ✓ Probe-based profiling: power is measured via physical probes. Typically, there is a mechanism to correlate the power readings with points in the application. Examples:
    - TMS320C55x Power Optimization DSP Starter Kit: It integrates NI Power Analyzer to provide a graphical view of power utilization over time.
    - Intel Energy Checker SDK: It measures power from the AC adaptor and enables correlation with specific regions of application code.
  - ✓ Power state-based profiling: these tools rely upon software interfaces into the platform's power states, which provides the number of times transitions occur between the platform power states. Examples:
    - PowerTOP: Linux tool that targets idle mode optimization techniques. It executes on the target device with an operating mode similar to the common Unit tool ***top***, where the tool provides a dashboard-like display. The display provides real-time updates as your applications execute on the target device. There are 6 categories of information:
      - C state residency information: Average amount of time spent in each C state and the average duration (%) that is spend in each C state.
      - P state residency information: Percentage of time the processor is in a particular P state.
      - Wakeups per second: number of times per second the system moves out of an idle C state.
      - Power usage: Estimate of power currently consumed and the amount of battery life remaining.
      - Top causes for wakeups: Rank-ordered list of interrupts, processes, and functions causing the system to transition to C0.
      - Wizard mode: suggestions for changes to the operating system that could reduce power utilization.