Controlling Processor C-State Usage in Linux

A Dell technical white paper describing the use of C-states with Linux operating systems

Stuart Hayes | Enterprise Linux Engineering

November 2013 | Revision 1.1

Contents

Figures

Introduction

This paper briefly describes when and how to take advantage of C-states on systems running Linux on x86 computers.

Background

In many computers, processors can use a significant amount of power when they are active. However, there are many times when the processors in the system are not busy running useful code. There are several methods designed to lower power usage on a processor when its full capacity isn't needed. One of these is the ability to lower the frequency at which a processors is running when it isn't being heavily used (each possible frequency is referred to as a processor performance state, or "P-state"). Another is to put various processor subsystems to sleep completely when the CPU is idle (these are called processor (CPU) power states, or "C-states").

However, while it does save power, it takes time to enter and exit C-states. Generally the higher the C-state, more power is saved when the processor is idle, but it takes longer to get the CPU back up again when it is needed. The C-states are:

C0	The CPU is actively running code
C1	The CPU uses the HLT instruction when idle—the clock is gated off to parts of the core, but it is relatively quick to wake up
C1E	This is actually just C1, except when C1E is enabled, the CPU lowers the CPU's speed & voltage when it is in C1
C2 & up	The CPU will shut off various parts of the core for greater power savings, at the cost of taking longer to wake up

In most Linux distributions today, C-states are enabled by default, which is great for most users. The operating system will choose which C-state to use based on processor utilization and other factors. Some users, though, want very low latency, and are very sensitive to how quickly a CPU can run code when needed, so user control is desired.

Checking C-State Usage

There are several ways to see how much idle time is being spent in the various C-states.

First check the kernel messages from boot ("dmesg |grep idle" or "grep idle /var/log/messages", for instance) to see which idle driver is in use.

If the acpi_idle driver is being used, the quickest method is to look at /proc/acpi/processor/CPU0/power, if it is available. It will show how many times each C-state has been entered, and how much time has been spent in each C-state. This method is not very user-friendly, though, because it has to be checked separately for each processor, it only shows cumulative totals for usage (so you have to check several times to see if a given C-state is currently in use).

Another option when using acpi_idle is the “acpitool” program, but this not included in all Linux distributions and may have to be downloaded separately.

If the system is using Intel processors, the program “i7z” can be used to check C-state usage regardless of which idle driver is being used (it accesses processor registers directly to see which C-states are being used). Another option for use with Intel processors is “powertop.” Note that programs such as i7z, which directly access processor registers and do not use ACPI, may not number states C2 and higher the same as they are numbered in ACPI.

```
Cpu speed from cpuinfo 2699.00Mhz
True Frequency (without accounting Turbo) 2699 MHz

Socket [0] - [physical cores=8, logical cores=8, max online cores ever=8]
  CPU Multiplier 27x || Bus clock frequency (BCLK) 99.96 MHz
  TURBO ENABLED on 8 Cores, Hyper Threading OFF
  True Frequency 2798.96 MHz (99.96 x [28])
  Max TURBO Multiplier (if Enabled) with 1/2/3/4/5/6 cores is  35x/35x/34x/32x/32x/32x
  Current Frequency 1913.42 MHz (Max of below)
        Core [core-id]  :Actual Freq (Mult.)      C0%   Halt(C1)%  C3 %   C6 %  Temp
        Core 1 [0]:       1566.42 (15.67x)         1    99.9       0       0    37
        Core 2 [2]:       1650.17 (16.51x)         1     100       0       0    35
        Core 3 [4]:       1835.05 (18.36x)         1     100       0       0    36
        Core 4 [6]:       1813.66 (18.14x)         1     100       0       0    36
        Core 5 [8]:       1779.50 (17.80x)         1     100       0       0    32
        Core 6 [10]:      1712.99 (17.14x)         1     100       0       0    34
        Core 7 [12]:      1913.42 (19.14x)         1     100       0       0    32
        Core 8 [14]:      1726.95 (17.28x)         1     100       0       0    32
  CPU Multiplier 27x || Bus clock frequency (BCLK) 99.96 MHz
  TURBO ENABLED on 8 Cores, Hyper Threading OFF
  True Frequency 2798.96 MHz (99.96 x [28])
  Max TURBO Multiplier (if Enabled) with 1/2/3/4/5/6 cores is  35x/35x/34x/32x/32x/32x
  Current Frequency 3051.25 MHz (Max of below)
        Core [core-id]  :Actual Freq (Mult.)      C0%   Halt(C1)%  C3 %   C6 %  Temp
        Core 1 [1]:       3051.25 (30.52x)      5.37    93.9       0       0    34
        Core 2 [3]:       2919.84 (29.21x)         1    99.7       0       0    34
        Core 3 [5]:       1957.27 (19.58x)         1     100       0       0    35
        Core 4 [7]:       1902.06 (19.03x)         1     100       0       0    33
        Core 5 [9]:       1904.37 (19.05x)         1     100       0       0    34
        Core 6 [11]:      1844.87 (18.46x)         1     100       0       0    31
        Core 7 [13]:      1866.32 (18.67x)         1     100       0       0    37
C0 = Processor running without halting20x)          1     100       0       0    36
C1 = Processor running with halts (States >C0 are power saver)
C3 = Cores running with PLL turned off and core cache turned off
C6 = Everything in C3 + core state saved to last level cache
  Above values in table are in percentage over the last 1 sec
[core-id] refers to core-id number in /proc/cpuinfo
'Garbage Values' message printed when garbage values are read
  Ctrl+C to exit
```

Figure 1 – i7z showing C-state usage when processors limited to C1

Disabling C-States for Low Latency

Most newer Linux distributions (Red Hat Enterprise Linux 6, Update 3, for example), on systems with Intel processors, use the “intel_idle” driver (probably compiled into your kernel and not a separate module) to use C-states. This driver uses knowledge of the various CPUs to control C-states without input from system firmware (BIOS). This driver will mostly ignore any other BIOS setting and kernel parameters, so if you want control over C-states, you should use kernel parameter “intel_idle.max_cstate=0” to disable this driver. Note that the intel_idle driver will only take control for CPUs it specifically knows about, so if you have, for example, a very new system with an older linux distribution, this may not be relevant.

You might also need to disable C1E in BIOS setup, if you are looking for low latency. As mentioned above, when C1E is enabled, the processor will try to lower processor clock speed and voltage when it

enters the C1 C-state, which might result in higher latency. Note that in the 3.9 and later linux kernels, the intel_idle driver treats C1 and C1E as separate states, so if you are using the intel_idle driver with these later kernels you can control whether C1E is used without disabling it in BIOS setup.

When the intel_idle driver is disabled, the Linux kernel will use the acpi_idle driver to use C-states. System firmware (BIOS) provides a list of available C-states to the operating system using an ACPI table. So, once intel_idle is disabled, users who want low latency can disable C-states by going into system (BIOS) setup during boot and telling BIOS to disable C-states (which takes them out of the ACPI tables).

Disabling C-states in this way will typically result in Linux using the C1 state for idle processors, which is fairly fast. If BIOS doesn't allow C-states to be disabled, C-states can also be limited to C1 with the kernel parameter "idle=halt", assuming the intel_idle driver is disabled (kernel parameter "idle=halt" should automatically disable cpuidle, including intel_idle, in newer kernels).

If a user wants the absolute minimum latency, kernel parameter "idle=poll" can be used to keep the processors in C0 even when they are idle (the processors will run in a loop when idle, constantly checking to see if they are needed). If this kernel parameter is used, it should not be necessary to disable C-states in BIOS (or use the "idle=halt" kernel parameter). Take care when keeping processors in C0, though--this will increase power usage considerably. Also, hyperthreading should probably be disabled, as keeping processors in C0 can interfere with proper operation of logical cores (hyperthreading). (The hyperthreading hardware works best when it knows when the logical processors are idle, and it doesn't know that if processors are kept busy in a loop when they are not running useful code.)

And, to reiterate—there are a number of kernel parameters and BIOS settings that deal with C-states, but most of these will be ignored if intel_idle is in use. Disabling intel_idle with kernel parameter "intel_idle.max_cstate=0" will result in more intuitive control of C-states, and there should not be any disadvantage to disabling it on systems that provide correct C-state information to the operating system via ACPI.

Dynamic Control of C-States

The methods to limit C-states above will all be permanent (until the system is rebooted). If you would like to have a system have extremely low latency during certain hours, but want more power savings at other times, there is a method to dynamically control which C-states are used.

To dynamically control C-states, open the file /dev/cpu_dma_latency and write the maximum allowable latency to it. This will prevent C-states with transition latencies higher than the specified value from being used, as long as the file /dev/cpu_dma_latency is kept open. Writing a maximum allowable latency of 0 will keep the processors in C0 (like using kernel parameter "idle=poll"), and writing a low value (usually 5 or lower) should force the processors to C1 when idle. The exact value needed to restrict processors to the C1 state depends on various factors such as which idle driver you are using, which CPUs you are using, and possibly the ACPI tables in your system. Higher values could also be written to restrict the use of C-states with latency greater than the value written. The value used should correspond to the latency values in /sys/devices/system/cpu/cpuX/cpuidle/stateY/latency (where X is the CPU number and Y is the idle state)—CPU idle states that have a greater latency than written to /dev/cpu_dma_latency should not be used.

One simple way to do this is by compiling a simple program that will write to this file, and stay open until it is killed. An example of such a program is below, and can be compiled by cutting and pasting the code into a file called setcpulatency.c, and running "make setcpulatency".

So, to minimize latency during certain hours, say from 8AM until 5PM, a cron job could be set up to run at 8AM. This cron job could run setcpulatency in the background with an argument of 0, with a cron table entry like this:

```
00 08 * * * /path/to/setcpulatency 0 &
```

Then, at 5PM, another cron job could kill any program that's holding /dev/cpu_dma_latency open:

```
00 17 * * * kill -9 `lsof -t /dev/cpu_dma_latency`
```

Of course, this is just an example to show how C-states can be dynamically controlled… the crond service is often disabled in low latency environments, but these steps could be taken manually or run by other means.

```
#include <stdio.h>
#include <fcntl.h>

int main(int argc, char **argv) {
      int32_t l;
      int fd;

      if (argc != 2) {
            fprintf(stderr, "Usage: %s <latency in us>\n", argv[0]);
            return 2;
      }

      l = atoi(argv[1]);
      printf("setting latency to %d us\n", l);
      fd = open("/dev/cpu_dma_latency", O_WRONLY);
      if (fd < 0) {
            perror("open /dev/cpu_dma_latency");
            return 1;
      }
      if (write(fd, &l, sizeof(l)) != sizeof(l)) {
            perror("write to /dev/cpu_dma_latency");
            return 1;
      }
      while (1) pause();
}
```

Figure 2 – setcpulatency.c

Summary

Normally C-States should be enabled by default, which is what is best for most users.

For users who want low latency under all circumstances, C-states should be disabled completely. To do this, use kernel parameter "intel_idle.max_cstate=0". In addition, to allow use of the C1 state,

disable the C1E state in BIOS, and use kernel parameter “idle=halt” (or disable C-states in BIOS in lieu of “idle=halt”). To keep the processors in C0, “idle=poll”.

For users who want low latency sometimes, but want the power savings associated with C-states at other times, dynamic control of C-states is possible.

There are several ways to check C-state usage, which should be used to verify that any attempt to control C-state usage is working correctly.