Expertise
and insight
for the future

Vladislav Govtva

Intel Xeon Server CPU Maximum Wake Latency Measurement

Metropolia University of Applied Sciences

Bachelor of Engineering

Electronics

Bachelor's Thesis

18 February 2019

Author Title Number of Pages Date	Vladislav Govtva Intel Xeon Server CPU maximum wake latency measurement 35 pages + 17 appendices 18 February 2019
Degree	Bachelor of Engineering
Degree Programme	Electronics
Professional Major	Electronics
Instructors	Matti Fischer, Principal Lecturer Artem Bityutskiy, Lead Developer

Modern CPUs implement mechanisms for managing power consumption when being idle by means of clock- or power-gating various internal blocks. These idle states are called "C-states". If a CPU is in an idle C-state to save power but there is a task ready to be executed, the CPU exits the C-state, although with delay. This delay is referred to as "C-state wake latency".

There are typically multiple C-state levels. And each level has its own wake latency. Based on those latencies Linux will choose which level to use in time-sensitive applications.
The aim of this research is to measure this latency with different levels of idle states.
The paper will explore idle power states with different generations of Intel Server CPUs, how those states affect wake latency, as well as how Linux performs power management.
There are requirements for the C-state wake latency measurement. The test must measure with greater than 95% precision and with sufficient resolution to measure 1-2 µs wake latency. The measurement method should not require expensive or unobtainable equipment, such as custom-made motherboards with measurement points.

The test developed for measuring C-state wake latency uses a special Network Interface Controller (NIC). NICs' architecture and high resolution clock allow it to send a network packet at scheduled time with 32 nanosecond granularity. The NIC keeps track of those network packets, which allows the CPU to stay idle until a packet is sent. When the packet is sent, NIC notifies the CPU by issuing an interrupt. This interrupt causes the CPU to wake up and take a timestamp with special software. The difference between the interrupt time and the timestamp is C-state wake latency.

The results showed that there is significant difference between generations of Intel Server CPUs and Linux's expectations for certain C-state's maximum wake latencies were incorrect.

Keywords	CPU, C-state, Idle, Intel, Linux

Contents

List of Abbreviations

TSC	Time Stamp Counter – A CPU counter that consistently increments, regardless of CC or PC state.
CPU	Central Processing Unit
PCH	Peripheral Control Hub
PCI	Peripheral Component Interconnect
NIC	Network Interface Controller
MAC	Media Access Control Address
HW	Hardware
BIOS	Basic Input/Output System
SW	Software
OS	Operating System

Acknowledgements

I wish to thank my thesis supervisor Matti Fischer for being a supervisor for this project. His willingness to give his time so generously has been very much appreciated. I would also like to Artem Bityutskiy for all his assistance. His time and effort which he charitably put into helping me with this project were crucial to its success.

I would also like to give special thanks to Intel employees who provided me with indispensable technical information: Len Brown and Erik Mann.

1 Introduction

In modern servers, which are becoming a bigger part of our everyday life, there are three aspects which are sought after. The aspects are: performance, Quality of Service, and power efficiency. Performance allows to execute increasingly complex tasks within shorter period of time. Quality of Service (QoS) provides the user an experience for which they may be more willing to pay. And power efficiency lowers expenses for the business that employs massive data centres.

Two of the above mentioned qualities, namely QoS and power efficiency, have an important common component – wake latency from idle state. “Wake Latency” in this paper refers to the time difference between a CPU receiving an interrupt while idle and the first line of code it executes. For QoS, wake latency partly affects how quickly user’s request gets processed, or in worst case, a user my not be able to use the service at all, e.g. in networking applications a crucial packet can get dropped if the latency is too high.

Power efficiency is always a good goal. For individuals using laptops or desktop computers, better efficiency results in longer battery life and better performance to cost ratio. Power efficiency in server CPU mainly concerns business owners, it affects their bottom line. When there are thousands and tens of thousands of CPUs in a datacentre, their cumulative power consumption becomes one of the biggest contributors to their expenses. [1] [2, p. 18]

That is why modern server CPUs implement power saving techniques, to improve power performance. The techniques can be summarized as: core voltage gating, core clock gating, completely turning off parts of the CPU, and combination of all of the above. The benefit of these techniques results in significant power savings, the consequence is increased wake latency.

The goal of this research is to isolate and measure wake latency in Intel Xeon CPUs, Intel CPUs specifically designed for server applications, and compare them to values written in Linux intel_idle driver. Linux intel_idle driver was used for reference because most data centres use Linux kernel running on Intel Xeon CPUs. Any information about wake latency of those systems comes from intel_idle driver.

The reason for measuring the wake latency is to validate the values in intel_idle. The values in the driver are calculated by Intel Corporation. Because the values are calculated, and not measured, there is a necessity to test them.

There is also an additional reason for investigating wake latency. Wake latency is a part of total interrupt latency, which also includes factors such as: PCI topology of the system, PCH, BIOS, and other factors. But measuring total response latency is not a topic of this research. Intel_idle driver specifies wake latency of their CPUs, not the entire system. And in simple systems the CPU wake latency is the largest components of response latency and can be used as a short cut for estimating total response latency of that system.

This latency is important to know when deciding which states are permissible for a specific application. If, for example, there is an application where a server has to start processing a signal within 50 µs of its arrival, any C-state with maximum latency longer than 50 µs should not be used. This scenario is common in telecommunications and networking applications.

There are also examples where large servers with a lot of computational power are necessary but not utilized all the time. Typically, servers used for research, where there are no tight timing constraints, fit this description. The work loads are usually intense but

few in number and far between each other. In this case, the deepest C-state available will be best suited, since those will result in biggest savings.

2 Intel Xeon CPU

2.9 million server units were shipped worldwide between second quarter of 2017 and same quarter 2018. [3] A large number of those systems were shipped with Intel Xeon CPUs inside. That is why Intel CPUs are the focus of this paper. Better understanding of their power management subsystem can be beneficial to all users.

2.1 Power Dissipation in Integrated Circuits

This work is primarily interested in wake latency, but since wake latency is a direct result of power saving techniques, a high level picture of power dissipation helps to put the issue in perspective. There are two primary sources of power dissipation in integrated circuits: logic power, leakage power. These two sources are going to be discussed in this chapter.

The first factor to cover is logic power. Logic power is power that is dissipated as a result of a transistor switching state, going from “ON” to “OFF” and vice versa, which happens during computation. This power can be expressed with an equation [2]:

$$P \sim C \times V^2 \times f \times AR \quad (1)$$

where:
P = power dissipated
C = circuit capacitance
V = voltage
f = frequency

AR = active resistance

Modern CPUs aim to increase the frequency in the Equation 1, in order to increase the number of calculations in a unit of time. But this increase in performance results in increase in power consumption. The reverse is also true, if the frequency is decreased, we get less performance and power dissipation. If frequency becomes 0, it would result in no power dissipation.

Another way to affect logic power is to modulate voltage and the effect will be quadratic. This means that reducing voltage by half can have a bigger effect on power dissipation than reducing frequency by half. If voltage is lowered by 50% the effective power is lowered by 75%, and with frequency the change would be one to one. But increasing voltage by a factor of 2, results in power quadrupling.

The reason why voltage is changed is because in order to achieve higher frequency, more voltage is required.

Second factor is leakage power. Leakage power, is the power that is dissipated inside the CPU to keep the transistors powered on, and memory state saved. The formula for estimating power dissipation from leakage is [2]:

$$P \sim V * (e^{V} * e^{t}) \quad (2)$$

where:
P – Power dissipated due to leakage
V – Voltage
t – Time

Notice, that the Equation 2 lacks any physical information about the circuit. That is because those variables are constants. This formula is used for estimation. It is useful because it accentuates the relationship between leakage power and CPU voltage, or time.

The higher the voltage the higher the power dissipation, exponentially. Leakage power can altogether be avoided by having 0 V, or in other words turning off the circuit.

2.2 Intel Server CPU Architecture & Modern CPU Power.

Over its long history, Intel had many different microarchitectures, but we will focus on the latest server CPU architecture, Skylake-SP.

There are five main components in a CPU:

- Core – the unit responsible for operation and logic of the CPU.
- Uncore – mesh which allows communication between CPUs, I/O, and Memory controllers.
- Cache Memory – ultra fast, low capacity memory, extremely close to CPU cores.
- I/O - UltraPath Interconnect, responsible for intra Socket communications (on multi-socket systems), PCIe.
- The Memory Controller (MC) - controllers responsible for Read/Write to RAM memory.

It is important to know these components and what they do in order to understand the impact of each individual power state. Different power states progressively disable parts of the CPU, in order to avoid unnecessary power dissipation. The effect of turning off those parts was shown by Intel using thermal imaging [4, p. 14].

Intel CPUs implement hardware power management component (HWPM) called punit – a dedicated component responsible for managing voltage, frequency, and off lining components. This HWPM takes requests from the OS to enter a specific C-state and depending on conditions grants it. The HWPM also puts the package into PC-state when all cores have met the requirements.

2.3 Intel Server CPU Power Management & Hardware Level Power Management

As mentioned in Section 2.1 there are two ways that a CPU dissipates power. The first one was active power, when CPU is doing work. To regulate how much power is dissipated Intel Xeons have execution power saving states – P-states, which control the voltage and frequency. The power dissipated during active power dissipation is doing useful work.

The second source of dissipated power was leakage. For regulating this power Intel CPUs implement idle power saving states – C-states. In contrast to P-states, C-states are for the moments when the CPU is idle, no work is being done, and there is no benefit to dissipating power. To avoid this unnecessary power dissipation parts of the CPU are completely turned off, which creates latency when that those parts are needed again. There are other states as well. Here is a full list of them:

- P-states – changing frequency and voltage of cores, to reduce power dissipation or increase performance
- C-state – voltage and frequency gating for CPU cores.
- PC-state – voltage gating components of the CPU
- S-state – sleep state, equivalent to halt
- D-state – device power management

In this research we focus on C-states and PC-states, because in servers S-states are rarely used, P-states do not have wake latency (the CPU is executing instructions in any P-state) and D-states are typically not as impactful.

Table 1 lists commonly implemented core C-states. But it is not necessary that all of them are present in all platforms. For example, Skylake Xeon CPUs do not have CC3 or CC7-CC10, because those states were omitted.

There is a significant step up in latency after C1E. There could be many reasons why, but this information is only available inside of Intel. Something that is noticeable is that CC3 is the first state that uses clock gating, and flushes caches. Restoring cached information, and restarting the core clock can be a complicated process.

PC-states are global package states which introduce further power savings, with additional costs in wake up latencies. One big difference from CC-states is that, in order to achieve any PC-state all cores must be in the same CC-state as the desired PC-state or deeper. For example, PC3 is possible if, and only if all cores are in CC3 or deeper. Table 2 contains commonly used PC states.

Figure 1 also shows the order of entering and exiting PC-states. This hierarchy may vary across generations and specific skews. Depending on circumstance a CPU can switch from PC6 to PC2 and then back. Such a circumstance would be when a PCI device issues a Direct Memory Access (DMA) read request. In this case a CPU will bring uncore online, and allow the PCI device to read from memory, after this the CPU will return to PC6 or remain in PC2 until next interrupt.

Table 1 Core C-state description

State	Status
CC0	Normal operating state of the CPU where code is being executed
CC1	All threads on that core execute HLT or MWAIT (C1/C1E) instruction. The core is in low power mode with low voltage and frequency. When returning to CC0, frequency and voltage are restored to the state before idle state.
C1E	Same as CC1, but when exiting this state, the CPU start operating in LFM
CC3	The contents of L1, L2 caches are flushed to LLC, and the core's voltage is lowered, and clock is stopped.
CC6	The core saves its architectural state to SRAM, and the core is completely powered down.
CC7 – CC10	These states are rarely used in Xeon CPUs, but if these states are implemented, they exhibit the same behaviour as CC6.

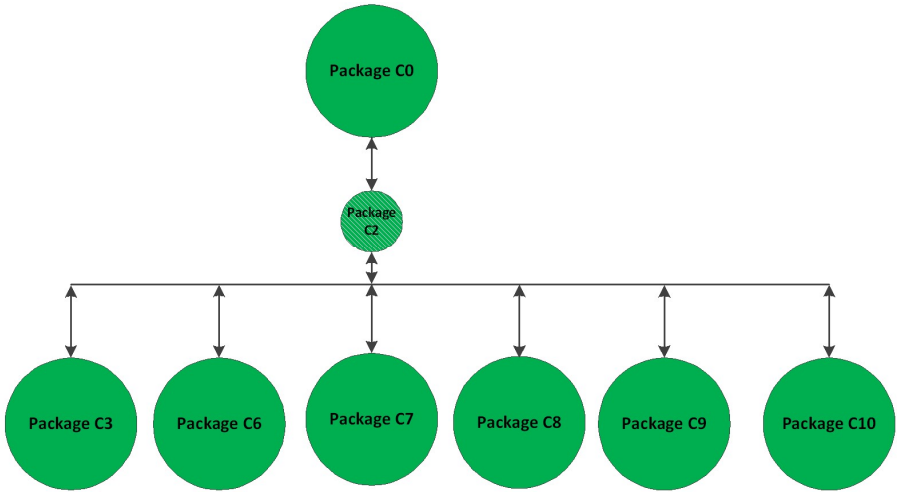


Figure 1 PC-state transitions [5, p. 76]

It is important to note that a CPU has a special register call Time Stamp Counter (TSC). In Intel Xeon CPU, starting from Nehalem generation of CPUs, two generations prior to Ivy Bridge, TSC became C-state invariant, meaning that it never stops counting, and it also counts at a constant frequency.

Table 2 Package C-state description

State	Status
PC0	This is the normal operating state for the processor. The processor remains in the normal state when at least one of its processor IA cores is in the C0 or C1. Individual processor IA cores may be in deeper power idle states while the package is in C0 state.
PC2	Special package C-state not explicitly available to SW. There are two scenarios when this state is used: • When deeper PC-states are off limits, and PC2 is the deepest PC-state • When the CPU is in PC3 state and a memory access request is received. In this case the CPU transitions to PC2, and completes all outstanding memory access requests, and then transitions back to PC3, or PC0 if there is a cause to wake from idle state.
PC3	Package low power state
PC6	LLC Cache is flushed, and turned off
PC7 – PC10	Typically present only on Desktop CPUs, not on CPU meant for servers

2.4 Linux and intel_idle Driver & cpuidle Driver

For this experiment Linux is the OS of choice. Linux's open source code allows to perform the experiment with greater degree of control compared to Linux's competitors. Another reason for choosing Linux is how wide spread Linus is as a server or data centre OS.

There are three major components in Linux's idle loop. There is kernel scheduler, idle governor, and a driver. The driver implements how to enter idle state. Different manufacturers have different implementation of idle driver. Kernel scheduler is responsible for scheduling idle state. Whenever it notices that there is an available moment for going into idle state, the scheduler calls "do_idle()" function (Appendix 3). In that function, kernel disables interrupts on the CPU that's executing that thread (Appendix 3, line 33), before continuing to make decisions about which idle state to request.

Then idle governor is called to prepare the system before idle state. Idle governor calls a function called "cpuidle_enter_state()" which is responsible for calling the correct function from the driver to enter a desired C-state (Appendix 1, line 40). The driver used with Intel Xeon's is called "intel_idle" driver. In this research we are only concerned with intel_idle driver.

Idle governor checks available information about future and past events, then it checks if there are any restriction on wake latency in Power Management QoS system, and based on all of this information makes a decision to request a specific idle state. After the state is chosen, governor calls corresponding function to enter that state. This is where "intel_idle()" is called. "Intel_idle()" is responsible for calling mwait with a correct hint (Appendix 2, line 37).

Mwait is a special instruction in CPUs which essentially instructs the CPU to enter power saving mode. Mwait can be called in a variety of ways. Mwait can be called with a time hint, to tell the CPU for how long it should be idle. Mwait can also be called with a hint for C-state. Intel_idle passes the information from idle governor to the hardware by calling mwait with hints to desired C-state.

Using a special tool, we can see exactly when mwait function is called. The tool is called “ftrace”. One of the features of ftrace is that it allows to trace exactly what functions were called by the CPU and in what order. In Listing 1 contains the ftrace output close to intel_idle:

```
01)|    call_cpuidle() {
02)|      cpuidle_enter() {
03)|        cpuidle_enter_state() {
04)|          sched_idle_set_state();
05)|          intel_idle() {
06)|            leave_mm();
07)|            mwait_idle_with_hints.constprop.2();
08)|          }
09)|          sched_idle_set_state();
10)|          arch_local_irq_enable() {
11)|
12)|            do_IRQ() {
13)|              irq_enter() {
```

Listing 1. ftrace output

Line “07” in function trace is crucial, because this is where the CPU goes into idle state, and turns off the core. The next relevant lines are 9-12 and those lines show that some code is executed after idle state and before handling interrupts. For greater degree of accuracy in measuring wake latency our timestamping function should fit between lines “08” and “09”.

Intel_idle driver also contains information about wake latencies of different Intel CPUs including Intel Xeon CPUs. The values for CPU tested can be seen in Table 3.

Table 3 Intel_idle Wake Latency values

Core Architecture family	Ivy Bridge, µs	Haswell, µs	Broadwell, µs	Skylake, µs
C1	1	2	2	2
C1E	10	10	10	10
C3	59	33	40	NaN
C6	80	133	133	133

2.5 Wake Up from Idle

When an interrupt is received the CPU raises a flag. The context manager then checks if the interrupt handling is enabled, and if it is, the context is switched to interrupt handling. If the interrupts are disabled, the process continues as normal, until the interrupts are enabled. If the CPU was in idle state and the interrupts are disabled, the first instruction after wake up will be the next instruction after mwait. Linux idle loop utilizes this mechanism to perform post idle state tasks before enabling interrupts and switching context to take care of the interrupt. This is exactly where a time stamp for wake latency should be taken.

The interrupt is going to be set up by an external source, the device that does it will be Intel i210 NIC.

3 Additional Equipment

3.1 Intel Network Interface Controller, Intel i210 NIC

Intel i210 NIC (datasheet [6]) is a network device designed by Intel. It is a feature labelled 802.1Qav [6, pp. 1,7,314-320] - Forwarding and Queuing Enhancements for Time-Sensitive Streams. This feature is intended to be used in time sensitive application like audio and video streams, where in very precise transmissions are required.

This feature includes a capability to schedule a packet to be sent at a specific time. Figure 7-11 in the datasheet [6], page 315, shows how this is achieved. There are four queues on the device which hold descriptors with Launch Time - time at which a packet should be sent - and pointer to data in host's memory. When the Launch Time is met, the card issues DMA read request and retrieves data into its own internal First in First out (FIFO) queue, and sends it from there.

There are parameters such as Fetch Delta which are used to specify when to pre-fetch the data into the cards FIFOs so that it would be ready by scheduled launch time. This

delta is useful in cases where the packets to be sent are large and require significant time to fetch.

There is another feature of the network card, which is related to packet scheduling. The card keeps only control message of the packet in its own memory. When it is time to send the packet the network card needs to retrieve the packet from host memory. Depending on the size of the packet, the time might vary. A way to deal with that issue is to configure a pre-fetch time. To achieve that, a user would need to know how long it takes to retrieve some amount of information from host's memory. For that reason Intel i210 keeps track of the longest 64-bit transaction. This time, the longest transaction, will be refered to as DMA (Direct Memory Access) time.

In this research this feature will be used to measure PC6 to PC2 transition. That will be the longest transition when retrieving a packet from an idle system, because it will include the time uncore comes online and become available.

3.1.1 Network Interface Controller Interrupts

When using 802.1Qav mode Intel i210 NIC sends out an interrupt for transmit packages. Table 4, from the NIC datasheet [6, p. 19], describes the transmission flow, in total 14 Step process. An interrupt is raised by the network interface controller at the end of this process.

Table 4 Intel i210 NIC Transmit data flow [6, p. 19]

Step	Description
1	The host creates a descriptor ring and configures one of the I210's transmit queues with the address location, length, head and tail pointers of the ring (one of 4 available Tx queues).
2	The host is requested by the TCP/IP stack to transmit a packet, it gets the packet data within one or more data buffers.
3	The host initializes descriptor(s) that point to the data buffer(s) and have additional control parameters that describe the needed hardware functionality. The host places that descriptor in the correct location at the appropriate Tx ring.
4	The host updates the appropriate queue tail pointer (TDT)
5	The I210's DMA senses a change of a specific TDT and as a result sends a PCIe request to fetch the descriptor(s) from host memory.
6	The descriptor(s) content is received in a PCIe read completion and is written to the appropriate location in the descriptor queue internal cache.
7	The DMA fetches the next descriptor from the internal cache and processes its content. As a result, the DMA sends PCIe requests to fetch the packet data from system memory.
8	The packet data is received from PCIe completions and passes through the transmit DMA that performs all programmed data manipulations (various CPU off loading tasks as checksum off load, TSO off load, etc.) on the packet data on the fly.
9	While the packet is passing through the DMA, it is stored into the transmit FIFO. After the entire packet is stored in the transmit FIFO, it is forwarded to the transmit switch module.
10	The transmit switch arbitrates between host and management packets and eventually forwards the packet to the MAC.
11	The MAC appends the L2 CRC to the packet and sends the packet to the line using a pre-configured interface.
12	When all the PCIe completions for a given packet are done, the DMA updates the appropriate descriptor(s).
13	After enough descriptors are gathered for write back or the interrupt moderation timer expires, the descriptors are written back to host memory using PCIe posted writes. Alternatively, the head pointer can only be written back.
14	After the interrupt moderation timer expires, an interrupt is generated to notify the host device driver that the specific packet has been read to the I210 and the driver can release the buffers.

The interrupt is raised after the packet is sent to MAC, Steps 10-11. Then The NIC updates all the information about the transmission, Step 12, and then it raises an interrupt, Step 14. According to the lead architect for Intel i210 NIC, Erik Mann, the time between Step 11 and Step 14 is 1.8 µs, later referred to as “NIC Interrupt Offset”. This number later comes up, because during this moment nothing is happening on the host.

3.1.2 Using Network Interface Controller as a Clock Source

In the context of the experiment i210 serves 2 purposes: external interrupt source and external time source. The former is the main feature, which allows the experiment to achieve better precision of measurement. The latter is a complementary. Taking the time from the same source as the interrupt source removes the necessity to synchronize times between the host and the external interrupt source.

Using i210 as a clock source has its challenges. The network card has its own clock, which is 64 bit clock with nanosecond precision. Taking a timestamp on i210 has a delay.

This delay will be referred to as Latch Offset. The name comes from the mechanism used by i210 to take a timestamp. The steps necessary to read time from i210 are:

1. Read SYSTIMR register on i210, this latches the time when the register is read
2. Read SYSTIML register on i210 to get 32 least significant bits of 64-bit timer
3. Read SYSTIMH register on i210 to get 32 most significant bits of 64-bit timer

It is difficult to measure at what point exactly the time latches on step 1, but it is easy to measure how long the entire read operation takes. This is done in characterization, covered later. But characterization obtains the average time to read contents of that register, or latch the time, and dividing that value by two will give Latch Offset.

4 Experiment

The general concept of the experiment is to use an external interrupt source, to raise an interrupt at a precise, known time. It is important that an external interrupt is used because in case a CPU has an internal algorithm which knows about the schedule wake-up, the CPU might start the wake-up process sooner, thus lower the perceived wake latency.

As soon as the CPU wakes up from idle state, it will start executing from the point where it went idle. At that point we take a time stamp, and record the wake latency. After this, interrupts are enabled, and the CPU OS continues to operate as usual. The experiment is fully automated and done with software tailor made for this purpose.

4.1 Wake Latency Tracer

WAke **L**atency **Tr**acer (waltr) is a tool for measuring C-state wake latency. It is currently designed to work with Intel i210 NIC only. There are three main components to waltr:

- **Armer** – sets up a unix domain socket, an interface which allows to interact with network stack from a normal program, and schedules packets to be set. Armer can schedule packets to be sent at any time in the future.

- **Waltr.c** – a kernel module is responsible for taking timestamps before and after “intel_idle();”, which are later used to calculate wake latency, and write them to a location accessible from userspace.

- **Waltr** – a python tool to controls the entire measurement process, aggregates all the data, and calculates statistical parameters: mean, median, standard deviation, and minimum, maximum.

Armer and Waltr scripts do not require extensive coverage because their purpose is to set up a measurement. The scripts themselves do not affect the final outcome of an

individual measurement. In addition, Armer and Waltr can operate remotely, when the system under test will be less busy. Waltr.c module, on the other hand, needs some explanation because it is directly involved in the measurement and has time sensitive components, which affect the final outcome of the experiment.

4.1.1 Waltr Module

Waltr module is a linux kernel module, designed to interpose its own function instead of “intel_idle()”. The module must be capable of taking a timestamp as close to the point where CPU goes into idle state, and right after. And after those timestamps are acquired waltr module must communicate those to user space.

Waltr module, when initialized, finds the address of “intel_idle()” function and inserts a jump to its own internal function in the first few lines of “intel_idle()”. Waltr’s internal function is called “intel_idle_wrapper()”. “Intel_idle_wrapper()” wraps “intel_idle()” function with other calls to get system information and timestamps. First, before calling “intel_idle()”, waltr takes a snapshot of MSR registers and a timestamp, which represents time before idle. After “intel_idle()” we take another timestamp straight away, and later we take a second snapshot of MSR registers. Listing 2 shows the result after waltr module was initialized, as seen in ftrace (with comments):

```
01)|    call_cpuidle() {
02)|      cpuidle_enter() {
03)|        cpuidle_enter_state() {
04)|          sched_idle_set_state(); # intel_idle would be called after this
point
05)|          read_cstate_msrs [waltr]() { # Take snapshot of state registers
06)|            read_cstate_msrs [waltr]();
07)|            ktime_get_real_ts64();
08)|            latch_nictime [waltr](); # This is where time is taken from
NIC
09)|            ktime_get_real_ts64();
10)|            leave_mm();
11)|            mwait_idle_with_hints.constpr # Actual mwait instruction
12)|            ktime_get_real_ts64();
13)|            latch_nictime [waltr](); # Time stamp after idle, also from
NIC
14)|            ktime_get_real_ts64();
15)|          }
16)|          sched_idle_set_state();
17)|          arch_local_irq_enable() {
18)| # Context switch
19)|            do_IRQ() { # Our transmit interrupt is processed
20)|              irq_enter() {
```

Listing 2 ftrace output when intel_idle is interposed with waltr

Compared to idle trace from Listing 1 there are now timestamping functions (Listing 2, lines 5-9, 11-14) as the last function before mwait (Listing 2, line 11) and right after.

4.2 Measurement Method

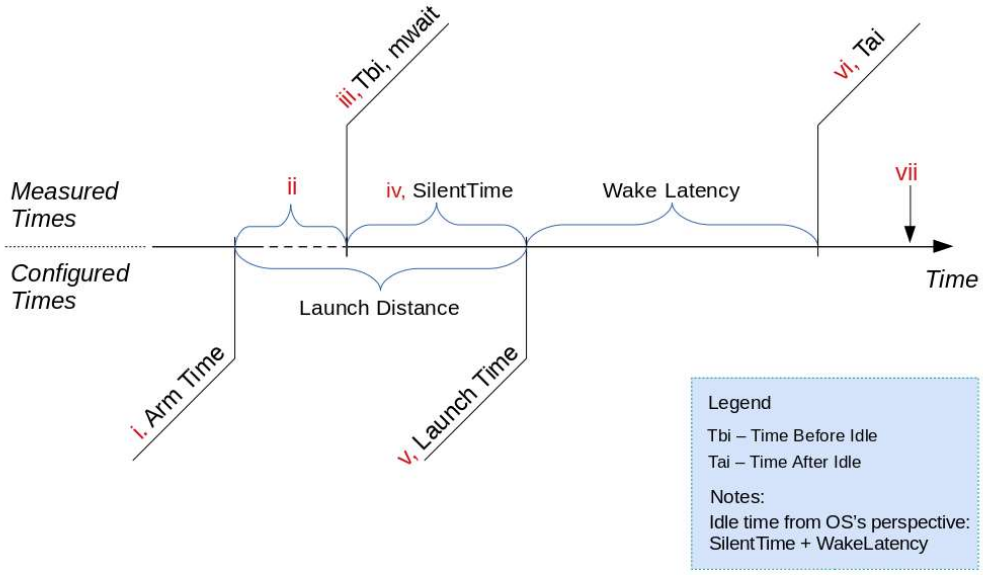


Figure 2 Waltr single data point measurement

A diagram of a single measurement can be seen in Figure 2. Data is logged after each measurement cycle, which makes parsing it simple. This allows to perform the experiment as many times as needed, and obtaining any amount of data. Here are the steps of the experiment:

i. Armer (waltr) arms a network packet to be sent at some point in the future. That point is Launch Time. After this waltr sleeps until after Launch Time.

ii. At this point there might be some activity on the system, which is not under our control. At the very least, the time it takes for a packet to go through the entire network stack may vary, so we cannot rely on this time to consistent. Another example of uncontrolled activity would be a network packet received, if the system is not isolated from any external network.

iii. Time before Idle, mwait – Here we take a time stamp before intel_idle is executed, and then mwait is called by the OS, which puts the CPU in idle state.

iv. Silent Time – A period of time that the CPU had after mwait and before receiving an interrupt. The CPU was in idle state, but not necessarily in any one CC- or PC-state, for example, it could have switched to CC6 then to CC1 and then back to CC6.

v. Launch Time – This is the time when the NIC starts sending a packet, which wakes up the CPU up to PC2, if the CPU was deeper than PC2. After NIC is done retrieving the packet from hosts memory, it takes 1.8µs to process the data, and raises the interrupt. Launch time is used as the time when CPU starts to wake up.

vi. Time After Idle – this is where the CPU starts executing instructions, and as the first action, it takes a timestamp. And waltr logs the output to temporary location.

vii. Waltr wakes up from idle state, parses the output and logs it to final output file.

This pattern is repeated thousands of times to find the maximum wake latency.

It is important to note that Launch Distance is configurable, but we cannot control Silent Time. This is because the OS might have other tasks scheduled between Armer and Tbi, denoted with step “ii”, and that is outside of user’s control. This is actually to the benefit of the test. Because it adds natural randomness to the test, increases likelihood of catching unforeseen scenarios. Launch distance was not kept constant in the experiment to increase the range of possible Silent Times.

The final calculation for wake latency is therefore:

$$WL_{C3.C6} = (WT - LO) - LT - IO \quad (3)$$

$$WL_{C1.C1E} = (WT - LO) - LT - DMA - IO \quad (4)$$

Where:
$WL_{C1,C1E}$ = Wake Latency for C1 and C1E
$WL_{C3,C6}$ = Wake Latency for C3 and C6
WT = Wake up time
LT = Launch Time
LO = Latch Offset
DMA = DMA read time
IO = Interrupt offset

4.3 Errors and Inaccuracies

Errors and Inaccuracies related to the method can be classified in two different categories: errors related to delayed timestamping and inaccuracy of NIC's interrupt timestamp.

Before going over the categories, additional information is required. Figure 3 shows the order of events that happen on CPU and NIC both between Time before Idle (Tbi) and Time after idle (Tai) timestamps. The figure has additional detail compared to Figure 1. Additional detail is necessary to better illustrate the origin of inaccuracies in the method used.

In the figure we see the sequence of events and their dependencies, as depicted with blue arrows. First action shown is taking a timestamp, Tbi, from the NIC (as explained in Chapter 3.1.2), this is done in order to filter out events where the "launch time" happened before CPU went to idle state. The timestamping is done on the NIC to keep everything in the same time domain. The Tbi is also interlaced by TSC timestamps to keep track of how long that took.

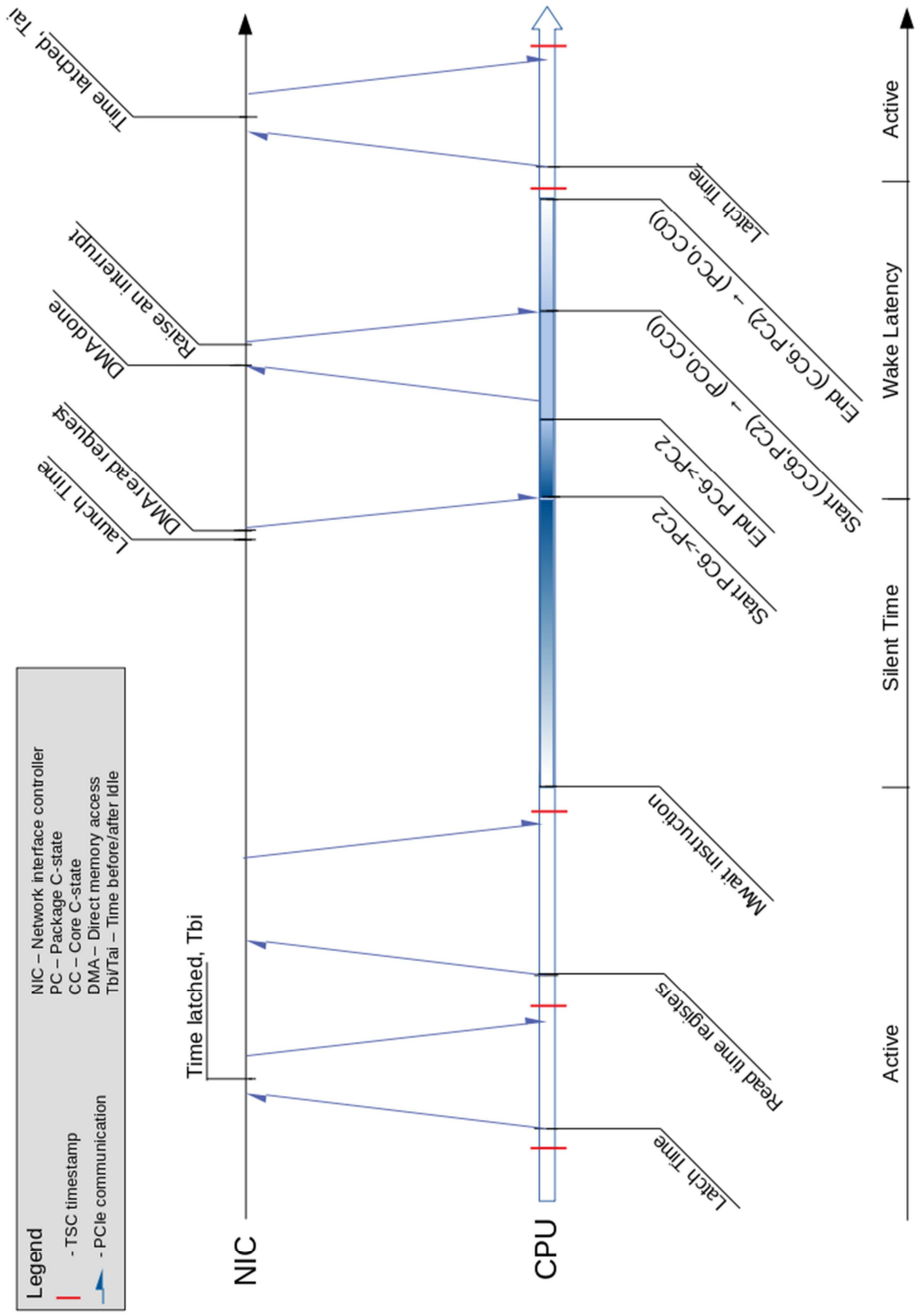


Figure 3 Tbi - Tai interval in detail

4.3.1 Timestamping Errors

Because the TSC and Tbi timestamps are in different domains, it is impossible to tell when Tbi happens exactly from the point of view of CPU. Which makes it difficult to get the time mwait was called in NIC domain. But it is possible to get a close approximation with two methods. First, if an assumption is made that the TSC read before Tbi happens almost at the same time, then the offset between clocks would be the difference between those time stamps. And the formula for time when mwait was called in NIC domain would be:

$$MWait = HTbi2 + (Tbi - HTbi1) \quad (5)$$

where:
Mwait = time when mwait is called
Tbi = Time before idle, NIC domain
HTbi1 = TSC timestamp before Tbi
HTbi2 = TSC timestamp closest to mwait

The mwait time error in this case is exactly the time between calling “latch_time()” instruction and the time latching on NIC. If the entire latch operation takes approximately 1.4µs then the error in this case would be sub 1.4µs. Since the use case for mwait time if filtering out data points when the interrupt was raised before calling mwait, then 1.4us window is sufficiently small.

The same arrangement is seen after idle around Tai, but in that case a different approach was used because the event being measured happened before TSC timestamp. In this case, an assumption is made that time latching happens in the middle of TSC timestamps. The following formula show to calculation of time when the CPU started executing instructions:

$$WT = Tai - (Htai2 - Htai1)/2 \quad (6)$$

where:
WT = wake up time
Tai = NIC Time after idle
Htai1 = TSC time before Tai
Htai2 = TSC time after Tai

In this case maximum error is half the difference between TSC timestamps, or in other words, hald the time it takes to latch the time on NIC, which was measured to be typically 1.4 µs. Therefore, the error for wake up time is ±0.7 µs.

4.3.2 Interrupt Time Error

In an ideal set up the wake up latency should be calculated with:

$$WL = WT - IT \quad (7)$$

where:
WL = Wake Latency
WT = Wake up Time, time when the first instruction was executed
IT = interrupt time, the time when an interrupt was received by the CPU, in the same domain as WT.

But in the method used in the experiment, the interrupt time is more complicated.

Figure 3 also illustrates a gap between “Launch Time” and “Raise an Interrupt”, which is used to calculate wake latency in Equations 3 and 4. The time difference between those

two events is labelled as Interrupt Offset. The way to account for this offset differs depending on which package C-state the CPU was in at Launch Time:

$$IO_{C6,C3} = (DMA_{start} - LT) + IO_{NIC} \quad (8)$$

$$IO_{C1E,C1} = DMA + IO_{NIC} \quad (9)$$

where:
$IO_{C6,C3}$ = total interrupt offset for C3 and C6 C-states
$IO_{C1E,C1}$ = total interrupt offset for C1 and C1E C-states
IO_{NIC} = Interrupt offset from DMA finished
DMA_{start} = DMA read request time
DMA = DMA read time
LT = Launch Time

The difference is a result of PC6 to PC2 transition during the first DMA read request. This measurement is part of the total wake latency, and therefore should be included in the final wake latency measurement. But when the PC-state is PC2 or shallower, DMA read time should be excluded.

4.4 Experiment Configuration and Setup

4.4.1 Hardware

The experiment requires two servers, or a server and a switch connected by an Ethernet cable. The server which is being tested should have Intel i210 NIC installed, it will be referred to as “localhost”. When the NIC attempts to send a packet, it first checks if there is a listener on the other end of the Ethernet cable, if there is no reply to its messages, the NIC does not even attempt to send a packet, and does not raise an interrupt. This is the purpose of the other machine, be it another server or a switch.

If it is possible to use two servers, the experiment can be controlled remotely. In that case most processes will not be on a system tested and data logging will be done remotely. The benefit of this approach is less noise on the system under test.

Six different hosts were tested, with four different generations of Intel Xeon CPUs. The results of for the four different generations are presented because there is little variation within the same generation. In the table below is the information about the CPUs tested:

Table 5 CPUs tested

Core Architecture family	Ivy Bridge	Haswell	Broadwell	Skylake
Number of Sockets	2	2	2	2
CPU model(s)	Intel(R) Xeon(R) CPU E5-2697 v2	Intel(R) Xeon(R) CPU E5-2697 v3	Intel(R) Xeon(R) CPU E5-2699A v4	Intel(R) Xeon(R) Platinum 8170M CPU
C-states	CC1,C1E, C3,C6	CC1,C1E, C3,C6	CC1,C1E, C3,C6	CC1,C1E,C6
PC-states	PC2,PC3,PC6	PC2,PC3,PC6	PC2,PC3,PC6	PC2,PC6

4.5 System Under Test Characterization

Characterization is a process where we define constants for the system under test. The constants are: latch time, reading time, TSC, and MSR registers, and getting time of day since epoch. When characterized, each of the above mentioned actions are performed thousands of times and the average is taken.

The result of characterization allows us to account for the impact of our own measurement tool. For example, before we call mwait, we first have to latch the time, which typically takes 1.3 µs, after that we have to read the time which takes 2.6 µs. So at this point our mwait instruction is 3.9 µs ahead from the moment time registers were latched. Knowing this we can calculate Silent Time more precisely.

More of these corrections will be mentioned in the remaining part of the thesis as they come up. What is important to keep in mind, is that every system is characterized individually, and all data is corrected using corresponding characterization values.

4.6 Configuration in Software and Operating System

A few steps are required in OS before running the test. The first and most important one is turning off all virtual cores except for one. This simplifies the experiment by making sure that all instructions are roughly executed in the desired order. Another benefit to turning off cores is no race condition for timestamps. The fact that virtual cores are turned off does not affect the wake latency because when a CPU is turned off in Linux OS, it is equivalent to requesting the deepest C-states, making sure no interrupts are assigned to it, and forbidding the OS from using that core. There is a possibility that any particular physical core can be slower than any other core in the same CPU, but this requires a lot of testing in order to find the slowest one.

Next important step is to configure the network stack to allow the network card to send a packet at the desired time. This is done through configuring the queueing discipline in Linux. A queueing discipline is a set of rules which orders the packets that are queued to be transmitted and then it offloads the packets to driver in the desired order and at desired time.

The queueing discipline used in this experiment is called Earliest Time First (ETF). ETF orders the packets based on the time they are supposed to be sent off. An additional feature in this queueing discipline which was developed with i210 in mind is “offloading”. Offloading is a feature which, if hardware supports it, offloads the packet to the network card some time before send off. That time is configured by a variable. For example, if a packet is supposed to leave at time X, ETF can be configured to send the information

about this packet to hardware 500 µs before send off. Which means that the actual send of time is controlled by the hardware on the NIC, not the OS. As a result a greater accuracy is achieved.

In this experiment, the network stack is configured so that the information about a packet and its send of time are offloaded to the NIC half a second before send of time. Which in practice means that as soon as we create a packet and send it through a network stack, it is immediately offloaded to the NIC.

The last step necessary is time synchronisation between NIC and host. This is a constraint placed by ETF queueing discipline. The time on the network card is International Atomic Time (TAI) and time in OS is Real-Time Clock, the difference between those, in 2019, is 37 seconds. A special program is started in the background called “phc2sys” which periodically synchronizes the time between the card and the host. Any other time synchronization should be turned off, otherwise, proper synchronization would be more difficult to achieve.

4.7 Collected Data

The experiment is performed as follows:

1. the system is configured and prepared
2. Launch Distance is set to 300 µs.
3. 1500 data points are obtained, with the method from chapter 4.2
4. Launch Distance is increased by 10%
5. Next set of 1500 data points is obtained
6. Continue until Launch Distance exceeds 8 ms

Two variables tested are: CPU core family and deepest C-state. C-states tested are C1, C1E, C3, and C6. CPUs tested are all generations from Xeon Ivy Bridge to Xeon Skylake, with exception of C3 on Skylake, because it is not available.

As a result of the testing, 15 combinations of C-state and CPU were tested, total of 525 data sets were obtained, with 790 000 data points recorded for the raw dataset. There is no real limit to the number of data points to obtain from the experiment, except for the law of diminishing returns, the more data points are obtain the less frequent are the new maximum data points.

Each C-state will be presented separately to showcase the difference in generations of Intel server CPUs. An example of measurement can be seen in Figure 4 Broadwell C6 Wake Latency Measurement showing the results for C6 wake on Broadwell machine. The points are arranged in chronological order. The orange line represents the Launch Distance used for that section of the graph. The scale on the left is for wake latency, and on the right for Launch distance.

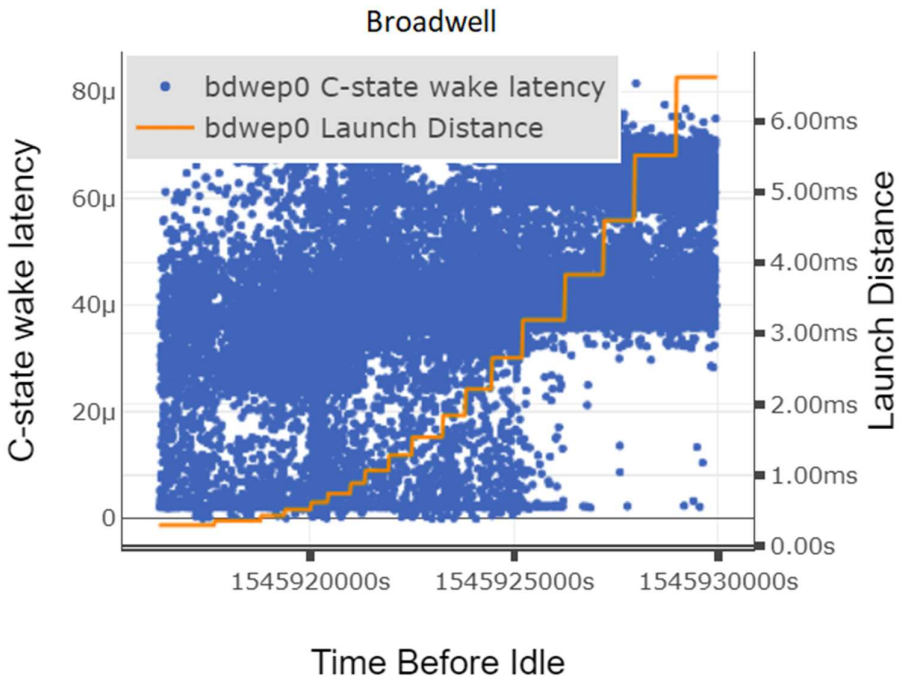


Figure 4 Broadwell C6 Wake Latency Measurement

On the plot there is a cloud of data points ranging from ~2us up to maximum of ~82us. The latter is the main focus of this thesis, and the entire figure can be summarized as that single value. Figures of other machines and C-states can be found in Appendix 5.

4.7.1 Results for C6 Measurements

All systems performed within limits when exiting from C6. Table below summarizes results in Appendix 5.1:

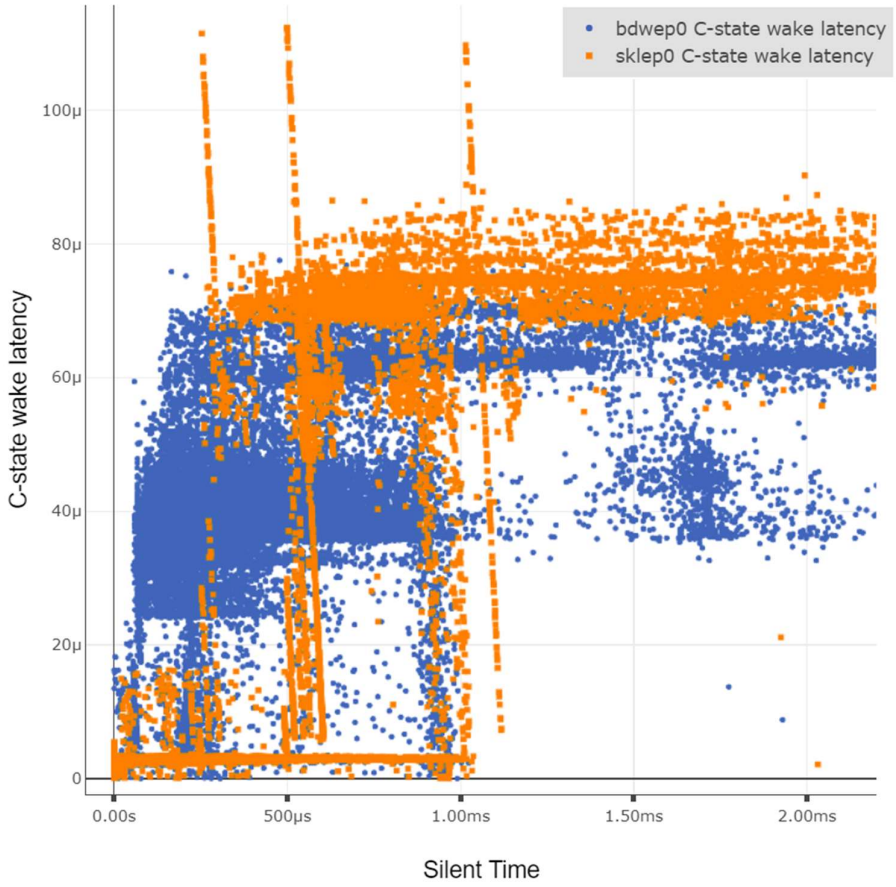
Table 6 C6 wake latency

CPU architecture	Maximum wake latency from C6, ns
Ivy Bridge	87062
Haswell	69727
Broadwell	81642
Skylake	112276

Interesting difference was found in behaviour between Broadwell and Skylake, Figure 5 and Figure 6 respectively. In order to better illustrate the difference between these generations of CPUs their wake latencies are compared with respect Silent Time (defined in 4.2). The reason for that, is Identical Silent Time is a better representation of the state of the system when it received an interrupt, because it is definitely known that mwait was called, and C6 was requested.

Additionally, there are special hardware counters to keep track of C-state residency. Comparing those two with regards to Silent Time gives even more insight. Former is depicted in Figure 5, latter in Figure 6.

Figure 5 Skylake vs Broadwell wake latency



On Figure 5 on x-axis we see silent time, and corresponding wake latency on y-axis. wake latency for Broadwell (bdwep0 on Figure 5) started to increase almost instantly, and increases with a constant slope up until 250us of Silent Time. This slope may suggest that the CPU began to enter deep C-state, but the interrupt was received and the CPU started to wake up again. Skylake CPU (sklep0 on Figure 5) on the other hand does

not show high latency values until 250 µs. This can suggest that the punit delays the earliest entry by 250 µs.

This theory is also supported in Figure 6. On the x-axis we see Silent Tim and on y-axis – value in C6 Residency Counter. We see discrete lines for both CPUs (the same graph as for Broadwell can be observed in Ivy Bridge and Haswell). This suggests that there are only few discrete point when the punit begins to enter idle after receiving mwait. For Broadwell those points are approximately 60 µs and 200 µs. For Skylake though, the points are 250 µs, 500 µs, and 1.1ms.

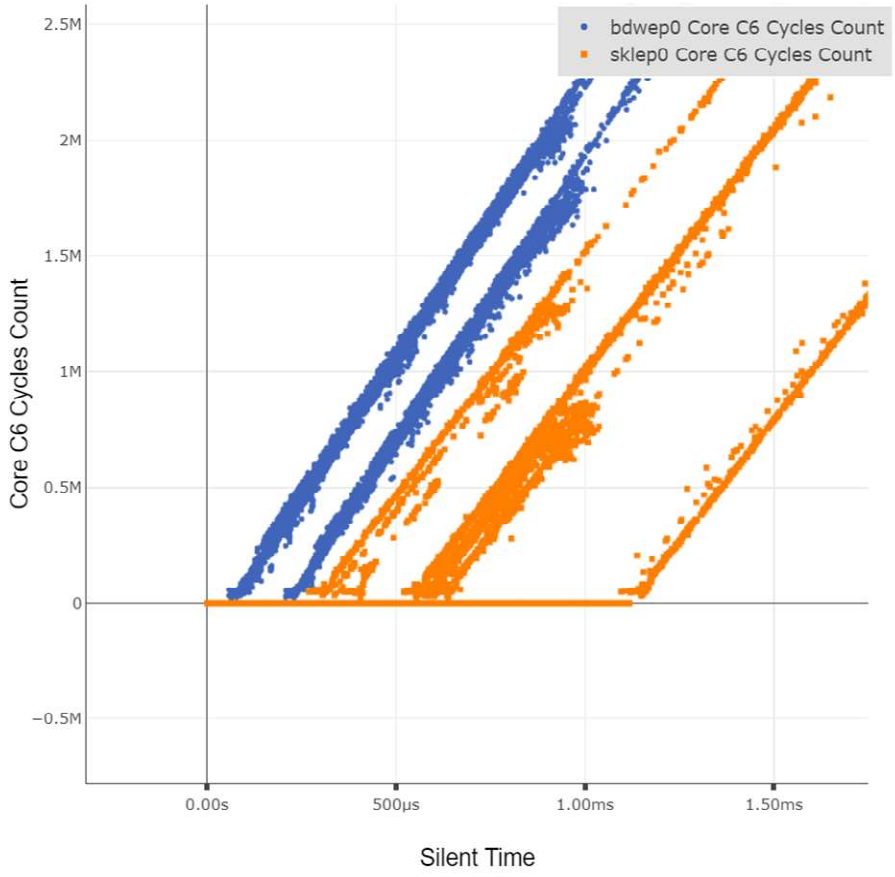


Figure 6 Broadwll vs Skylake CC6 cycles

4.7.2 Results for C3 Measurements

Neither Core C3 nor Package C3 are available in the late 2018 generation of Intel CPUs Skylake Xeon Core. That is why it is not present in the figures below. The difference between the C6 and C3 can be found in C-states description table and PC-state description table in Chapter 2.3

Table 7 C3 Maximum Wake Latency

CPU architecture	Maximum wake latency from C3, ns
Ivy Bridge	78807
Haswell	59617
Broadwell	71803
Skylake	NaN

In Table 7 we see that the wake latency for Broadwell Core and Haswell Core systematically exceeds those specified in intel_idle driver. This is an important finding, and will be addressed in Chapter 5.

4.7.3 Results for C1E Measurements

Table 8 C1E Maximum wake latency

CPU architecture	Maximum wake latency from C6, ns
Ivy Bridge	21765
Haswell	15911
Broadwell	18661
Skylake	16141

Here again, like with C3, we see that intel_idle C1 wake latency value was exceeded across all generations. These results include RTD, because additional testing showed that in some cases RTD is affected by C-state.

4.7.4 Results for C1 Measurements

C1 is difficult to measure, the expected wake latency for C1 is 2 µs which is too small to measure with the current setup. But even if that is the case, anomalies where the exit latencies are greater than 2 µs can still be detected. So this test is still valuable. And indeed, we see that there are cases where we have significantly large wake latency.

Table 9 C1 Maximum wake latency

CPU architecture	Maximum wake latency from C6, ns
Ivy Bridge	19270
Haswell	14956
Broadwell	15153
Skylake	15471

All generations failed to meet their expected maximum wake latency. All the configurations were checked and it is quite likely that this is indeed the time it took for the CPU's to wake up. These results include RTD, because additional testing showed that in some cases RTD is affected by C-state.

4.8 Results Summary

Table 10 summarizes all maximum values obtained in the experiments.

Table 10 Results Summary

Core Architecture family	Ivy Bridge, ns	Haswell, ns	Broadwell, ns	Skylake, ns
Wake Latency Max., C6	82794	69727	74895	108573
Wake Latency Max., C3	78807	59617	71803	NaN
Wake Latency Max. C1E	21765	15911	18661	16141
Wake Latency Max., C1	19270	14956	15153	15471

The results show that there are inaccuracies in intel_idle driver. In C1, C1E, and C3 have points which exceed the expected maximum latency.

5 Making Use of the Results

It is important to keep in mind that the goal was to measure the wake latency of a CPU, not interrupt response latency. CPU wake latency should only be dependant of the CPU alone, and it is a subset of interrupt response latency. Interrupt response latency can depend on number of factors. Whether the system is idle or busy. Whether an interrupt source was external, and if so, are there physical factors affecting latency, such as distance. Are there any other components introducing latency between external interrupt source and CPU? And many other conditions. This distinction is important when trying to make use of the results, because the latencies mentioned above are smaller than real world response latency.

The results show that Intel CPU's wake latency as it is stated in intel_idle driver does not match with certain C-states. In some applications such as automotive, audio and sound, and telecommunications wake latency determines which power saving strategy can be used. For example, for best quality of service C6, C3, and C1E are disabled in telecommunication services due to their high latency, which can result in packet drop.

To make proper use of the results, one must also look at the frequency at which latencies close to maximum occur. For example, if the maximum measured point occurs once every million points, this is something to take into consideration. Furthermore, the measurement method also aims to find the maximum, not measure frequency of its occurrence. Therefore, the frequency in these results is exaggerated.

6 Conclusion

The method implements using an external interrupt for wake up, which makes it possible to implement the same method on other systems or CPU manufacturers with minimum manufacturers. And as mentioned in Chapter 4 the fact that the interrupt is external allows the experiment to be valid even in systems with some optimization algorithms. In addition to that the method uses the same clock source, i210 internal timer.

There is strong evidence suggesting that the method is sufficiently accurate. As we see in Chapter 4.1.1 the module is installed in such a way that the timestamps are taken as the first few steps after idle, when we start executing code. The placement of the timestamps and their granularity result in the smallest footprint on the final measurement, in other words, we are not measuring ourselves. Te Network card specifies that when used in Qav mode the accuracy is sub 0.5 µs when it comes to delay from Launch Time to packet in MAC. [6]

This research can serve as a base for further research into the behaviour of the CPU related to wake latencies. For example, measuring difference between pure Core C-state and Core C-state with Package C-state, to see how Package C-state influence wake latency. Also investigating the effect of Efficiency Power Bias (EPB), which is a commonly available setting.

Another example would be measuring the effect of more complicated integrated circuit on wake latency, by measuring a range of CPU's within the same generation. Intel offers a range of CPUs with a varying features, core count, and power.

This research also provides a method that can be used for future generations of Intel CPUs, and with minor adjustments can be use in non-Intel CPU's.

The goal of this research was to find maximum wake latency for Intel Xeon CPUs and compare the results to values specified in Linux's intel_idle driver. This goal was achieved. As a result of extensive tests, with thousands of data points, the research concluded that C1, C1E, and C3 figures for maximum wake latency were not correct the last four generations of Intel Xeon Cores (as of end 2018 and beginning of 2019). Further research can be done to investigate the cause of these longer latencies, for example, isolating specific BIOS configuration which affect idle performance.

7 References

[1] N. Rasmussen, "Determining Total Cost of Ownership for Data Center and Network Room Infrastructure," 8 June 2011. [Online]. Available: https://www.apc.com/salestools/CMRP-5T9PQG/CMRP-5T9PQG_R4_EN.pdf. [Accessed 21 January 2019].

[2] C. Gough, I. Steiner and W. Saunders, Energy Efficient Servers, Barkeley, CA: Apress, 2015.

[3] IDC, "Worldwide server market Revenue Grew 43.7% Year Over Year to a Record $22.5 Billion During the Second Quarter of 2018, According to IDC," 5 September 2018. [Online]. Available: https://www.idc.com/getdoc.jsp?containerId=prUS44259518. [Accessed 22 January 2019].

[4] S. Jahagirdar, "hotchips.org," 27-29 August 2012. [Online]. Available: http://www.hotchips.org/wp-content/uploads/hc_archives/hc24/HC24-1-Microprocessor/HC24.28.117-HotChips_IvyBridge_Power_04.pdf. [Accessed 20 January 2019].

[5] intel.com, *Intel's 8th and 9th Generation Core Family datasheet Vol. 1,* https://www.intel.com/content/www/us/en/products/docs/processors/core/8th-gen-core-family-datasheet-vol-1.html, 2019.

[6] I. N. D. (ND), Intel® Ethernet Controller i210, Intel, 2018.

[7] Intel, "intel.com," Intel, 10 July 2017. [Online]. Available: https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview. [Accessed 17 January 2019].

[8] S. Jahagirdar, V. George, I. Sodhi and R. Wells, *Power Management of the Third Generation of Intel Core Micro Architecture formerly codenamed Ivy Bridge,* Hot Chips, 2012.

Appendix 1. Cpu .c

Location in the kernel tree: drivers/cpuidle/cpuidle.c

```
int cpuidle_enter_state(struct cpuidle_device *dev, struct cpuidle_driver
*drv,
                        int index)
{
        int entered_state;

        struct cpuidle_state *target_state = &drv->states[index];
        bool broadcast = !!(target_state->flags & CPUIDLE_FLAG_TIMER_STOP);
        ktime_t time_start, time_end;
        s64 diff;

        /*
         * Tell the time framework to switch to a broadcast timer because our
         * local timer will be shut down.  If a local timer is used from an-
other
         * CPU as a broadcast timer, this call may fail if it is not availa-
ble.
         */
        if (broadcast && tick_broadcast_enter()) {
                index = find_deepest_state(drv, dev, target_state->exit_la-
tency,
                                           CPUIDLE_FLAG_TIMER_STOP, false);
                if (index < 0) {
                        default_idle_call();
                        return -EBUSY;
                }
                target_state = &drv->states[index];
                broadcast = false;
        }

        /* Take note of the planned idle state. */
        sched_idle_set_state(target_state);

        trace_cpu_idle_rcuidle(index, dev->cpu);
        time_start = ns_to_ktime(local_clock());

        stop_critical_timings();
        entered_state = target_state->enter(dev, drv, index);
        start_critical_timings();

        sched_clock_idle_wakeup_event();
        time_end = ns_to_ktime(local_clock());
        trace_cpu_idle_rcuidle(PWR_EVENT_EXIT, dev->cpu);
```

```
/* The cpu is no longer idle or about to enter idle. */
sched_idle_set_state(NULL);

if (broadcast) {
        if (WARN_ON_ONCE(!irqs_disabled()))
                local_irq_disable();

        tick_broadcast_exit();
}

if (!cpuidle_state_is_coupled(drv, index))
        local_irq_enable();

diff = ktime_us_delta(time_end, time_start);
if (diff > INT_MAX)
        diff = INT_MAX
```

Appendix 2. intel_idle.c

Location in the kernel tree: drivers/idle/intel_idle.c

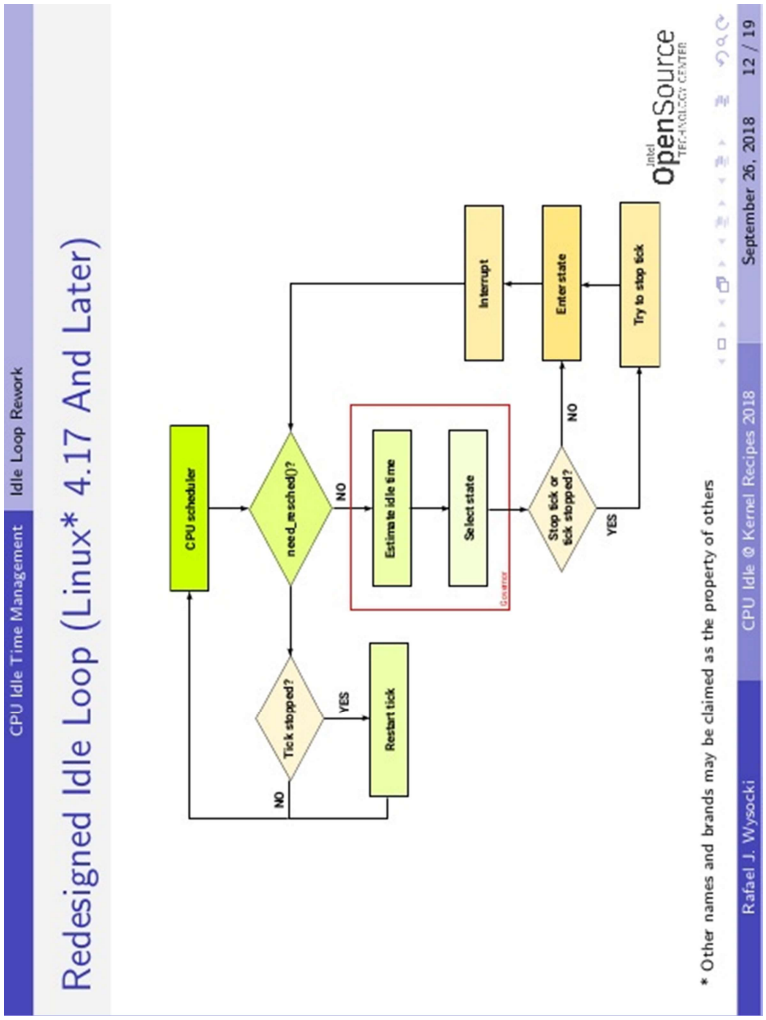
```
/**
* intel_idle
* @dev: cpuidle_device
* @drv: cpuidle driver
* @index: index of cpuidle state
*
* Must be called under local_irq_disable().
*/
static __cpuidle int intel_idle(struct cpuidle_device *dev,
                                struct cpuidle_driver *drv, int index)
{
        unsigned long ecx = 1; /* break on interrupt flag */
        struct cpuidle_state *state = &drv->states[index];
        unsigned long eax = flg2MWAIT(state->flags);
        unsigned int cstate;
        bool uninitialized_var(tick);
        int cpu = smp_processor_id();
        /*
         * leave_mm() to avoid costly and often unnecessary wakeups
         * for flushing the user TLB's associated with the active mm.
         */
        if (state->flags & CPUIDLE_FLAG_TLB_FLUSHED)
                leave_mm(cpu);

        if (!static_cpu_has(X86_FEATURE_ARAT)) {
                cstate = (((eax) >> MWAIT_SUBSTATE_SIZE) &
                                MWAIT_CSTATE_MASK) + 1;
                tick = false;
                if (!(lapic_timer_reliable_states & (1 << (cstate)))) {
                        tick = true;
                        tick_broadcast_enter();
                }
        }

        mwait_idle_with_hints(eax, ecx);

        if (!static_cpu_has(X86_FEATURE_ARAT) && tick)
                tick_broadcast_exit();

        return index;
}
```

Appendix 3. kernel_shed.c

```
/*
* Generic idle loop implementation
*
* Called with polling cleared.
*/
static void do_idle(void)
{
        int cpu = smp_processor_id();
        /*
         * If the arch has a polling bit, we maintain an invariant:
         *
         * Our polling bit is clear if we're not scheduled (i.e. if rq->curr
!=
         * rq->idle). This means that, if rq->idle has the polling bit set,
         * then setting need_resched is guaranteed to cause the CPU to
         * reschedule.
         */

        __current_set_polling();
        tick_nohz_idle_enter();

        while (!need_resched()) {
                check_pgt_cache();
                rmb();

                if (cpu_is_offline(cpu)) {
                        tick_nohz_idle_stop_tick_protected();
                        cpuhp_report_idle_dead();
                        arch_cpu_idle_dead();
                }

                local_irq_disable();
                arch_cpu_idle_enter();

                /*
                 * In poll mode we reenable interrupts and spin. Also if we
```

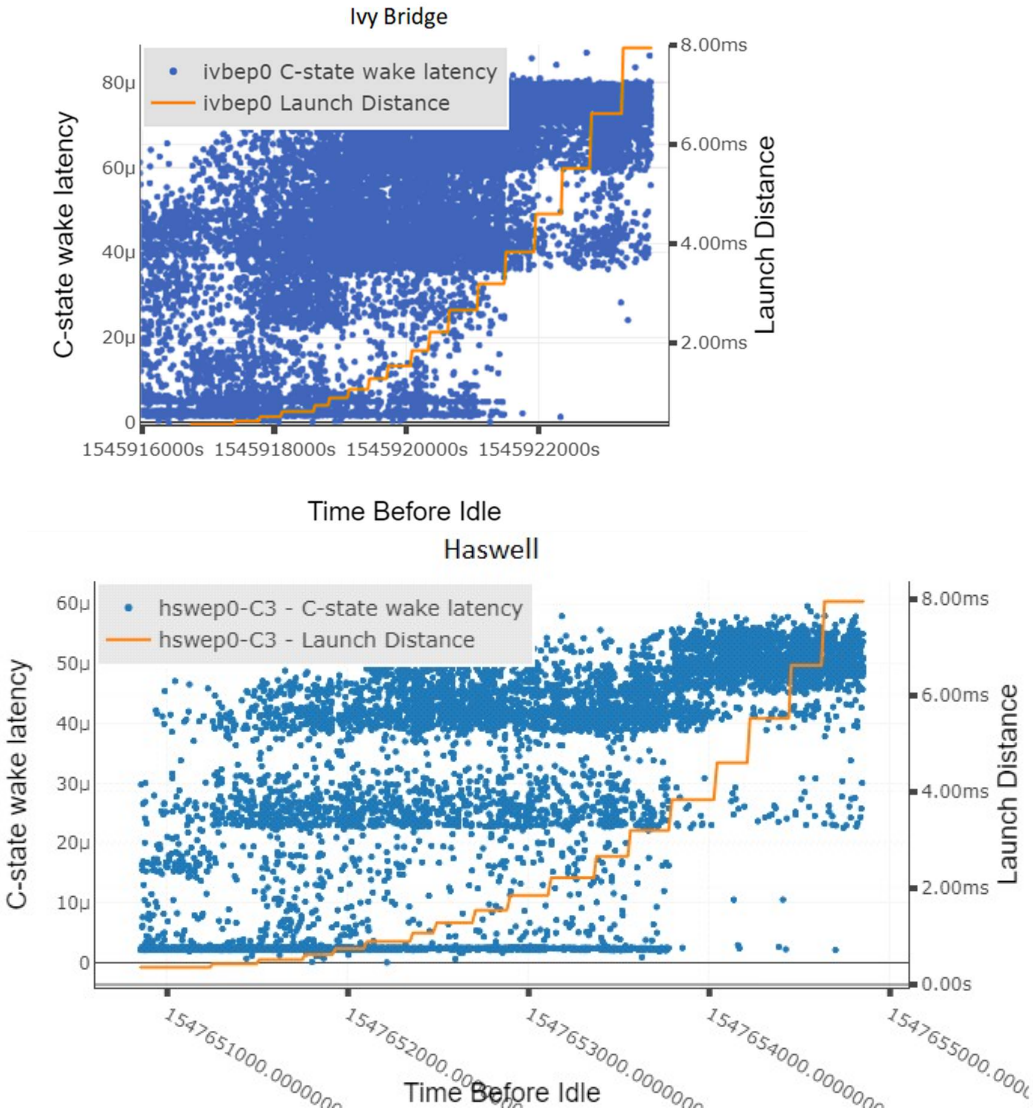
```
                 * detected in the wakeup from idle path that the tick
                 * broadcast device expired for us, we don't want to go deep
                 * idle as we know that the IPI is going to arrive right away.
                 */
                if (cpu_idle_force_poll || tick_check_broadcast_expired()) {
                        tick_nohz_idle_restart_tick();
                        cpu_idle_poll();
                } else {
                        cpuidle_idle_call();
                }
                arch_cpu_idle_exit();
        }

        /*
         * Since we fell out of the loop above, we know TIF_NEED_RESCHED must
         * be set, propagate it into PREEMPT_NEED_RESCHED.
         *
         * This is required because for polling idle loops we will not have had
         * an IPI to fold the state for us.
         */
        preempt_set_need_resched();
        tick_nohz_idle_exit();
        __current_clr_polling();

        /*
         * We promise to call sched_ttwu_pending() and reschedule if
         * need_resched() is set while polling is set. That means that clear-ing
         * polling needs to be visible before doing these things.
         */
        smp_mb__after_atomic();

        sched_ttwu_pending();
        schedule_idle();

        if (unlikely(klp_patch_pending(current)))
                klp_update_patch_state(current);
}
```

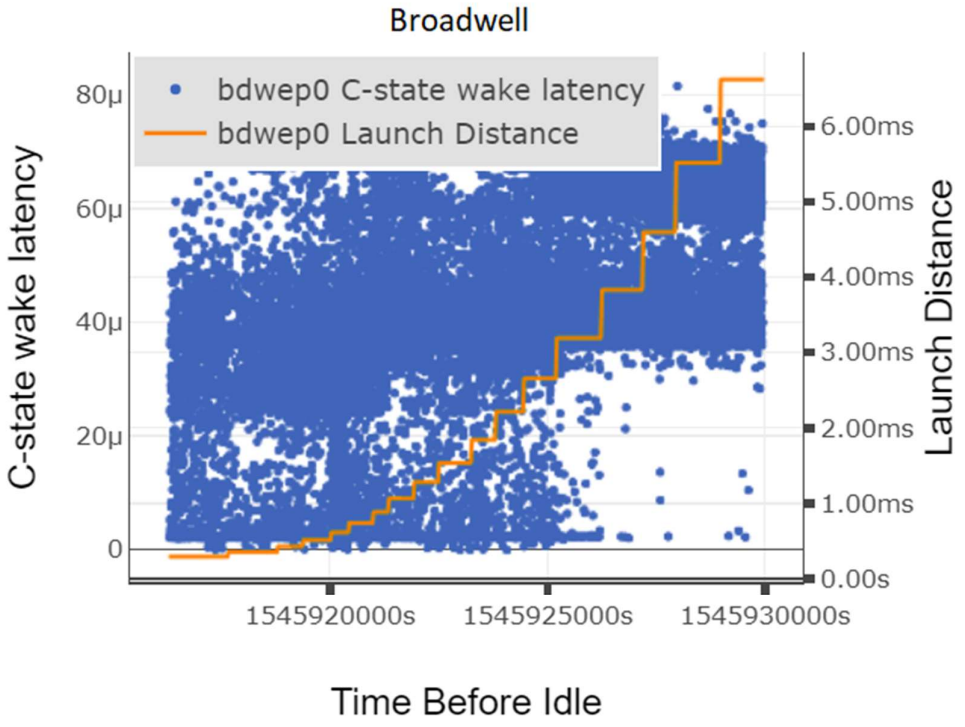
Appendix 4. Idle Loop Figure

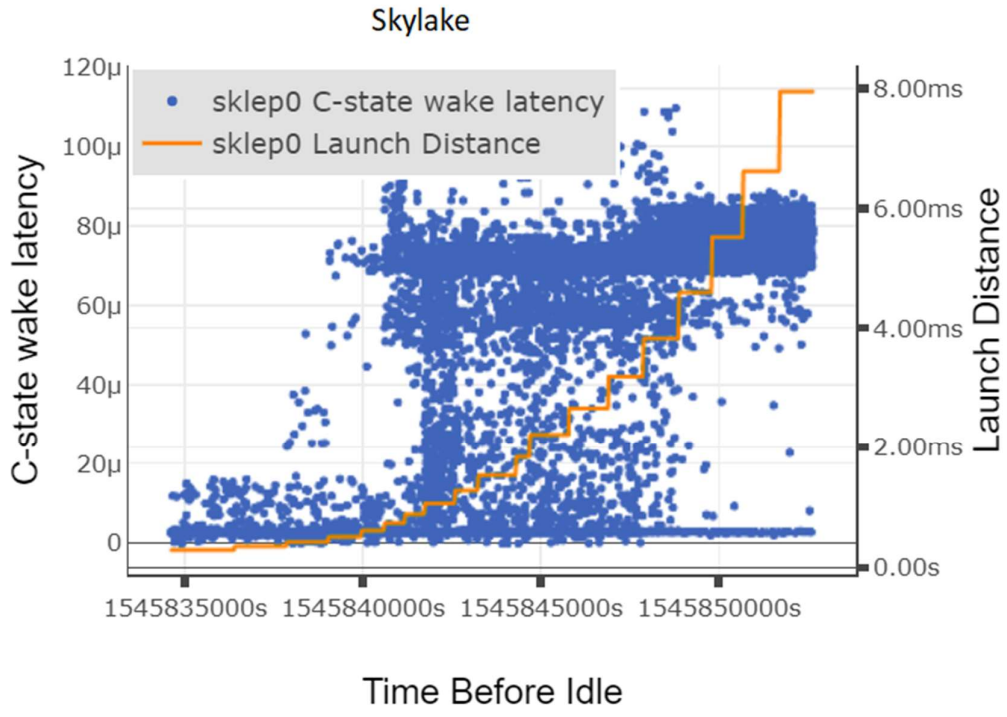


Appendix 5. C-State Wake Latency Figures

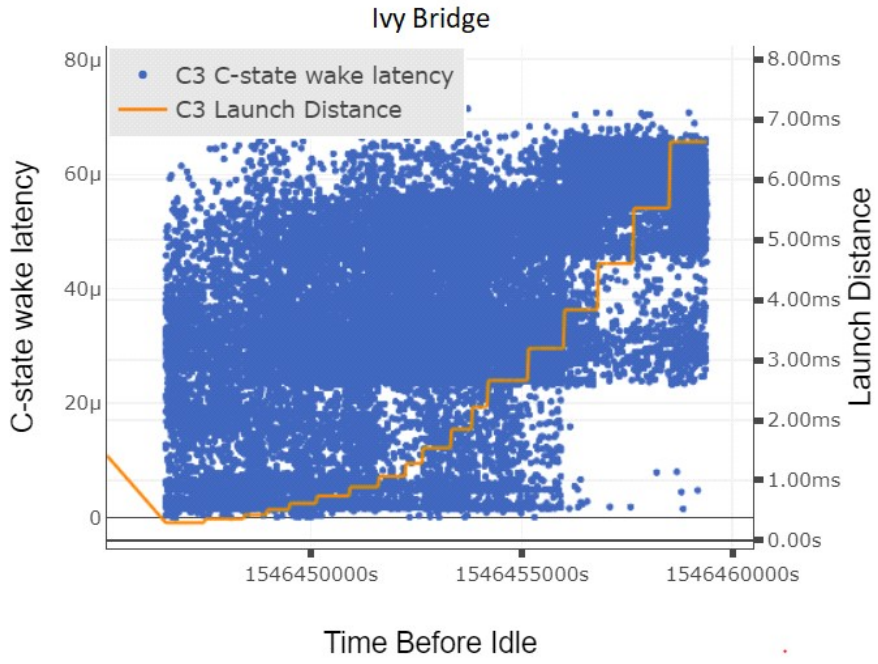
Appendix 5.1. C6 Wake Latency Figures

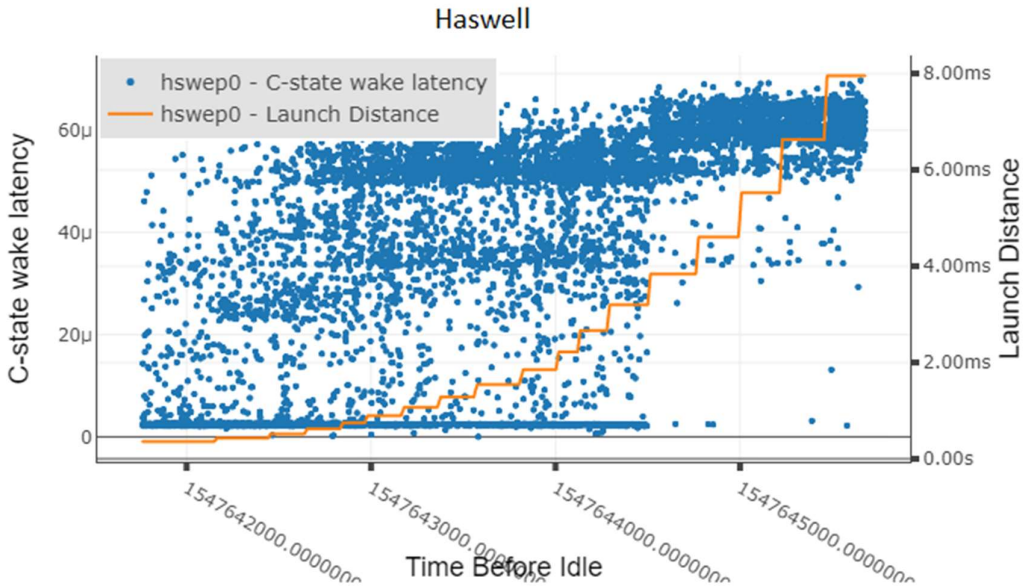


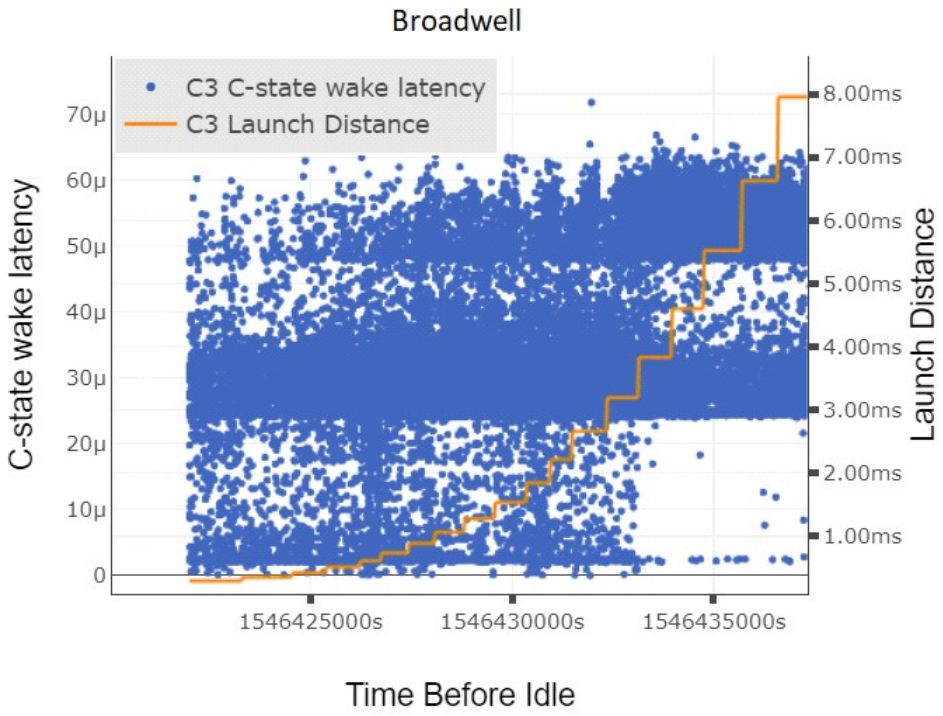

Broadwell
bdwep0 C-state wake latency
bdwep0 Launch Distance
C-state wake latency
Launch Distance
80µ
60µ
40µ
20µ
0
6.00ms
5.00ms
4.00ms
3.00ms
2.00ms
1.00ms
0.00s
1545920000s
1545925000s
1545930000s
Time Before Idle


Skylake
sklep0 C-state wake latency
sklep0 Launch Distance
C-state wake latency
Launch Distance
120µ
100µ
80µ
60µ
40µ
20µ
0
8.00ms
6.00ms
4.00ms
2.00ms
0.00s
1545835000s
1545840000s
1545845000s
1545850000s
Time Before Idle

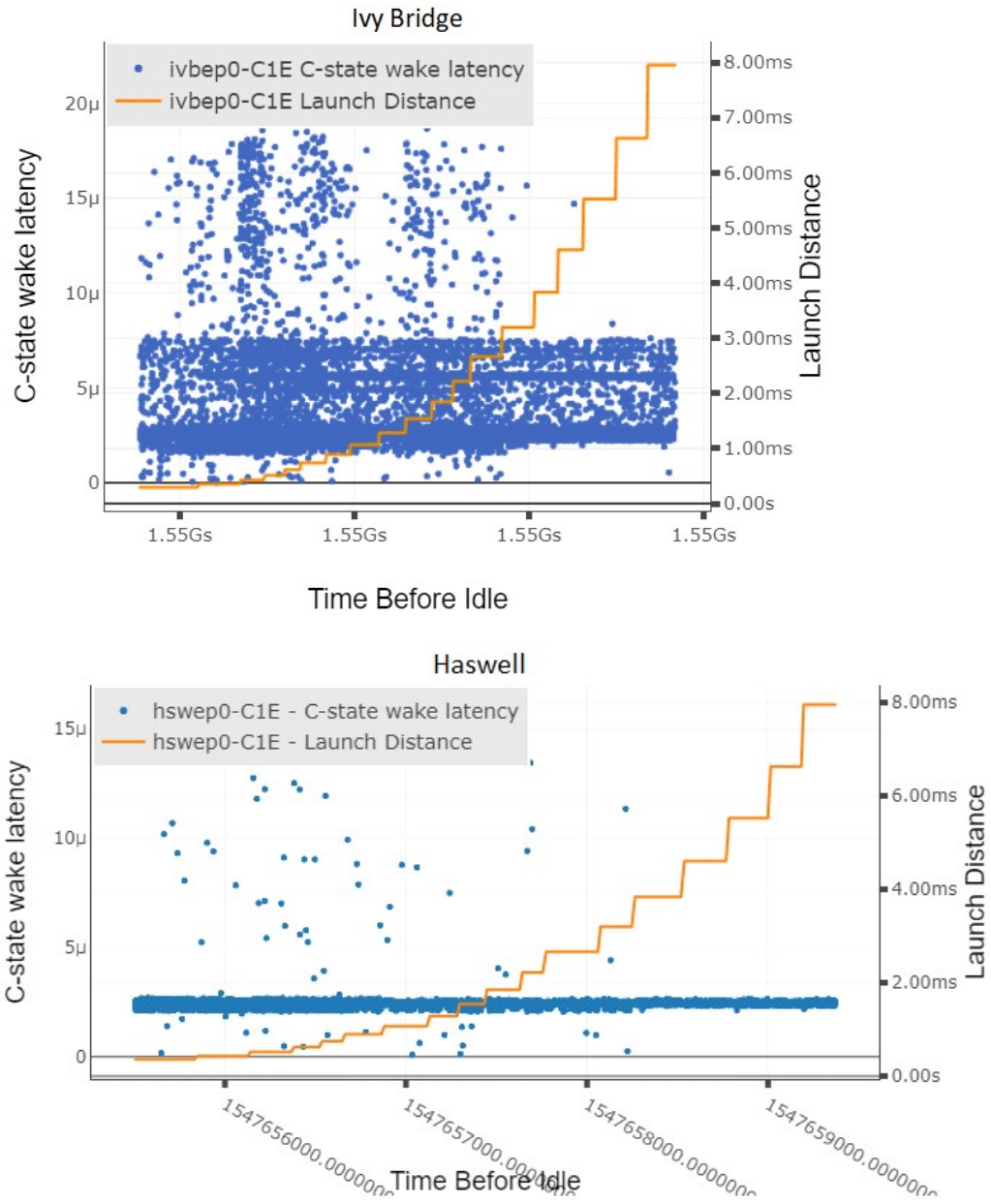
Appendix 5.2. C3 Wake Latency Figure

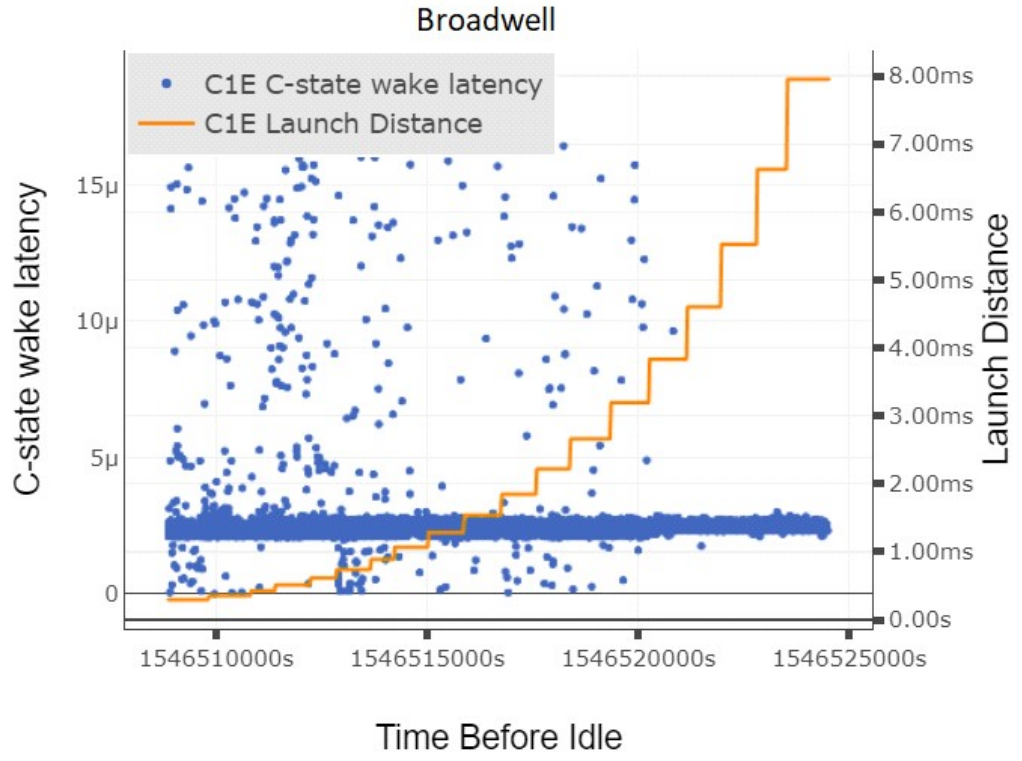


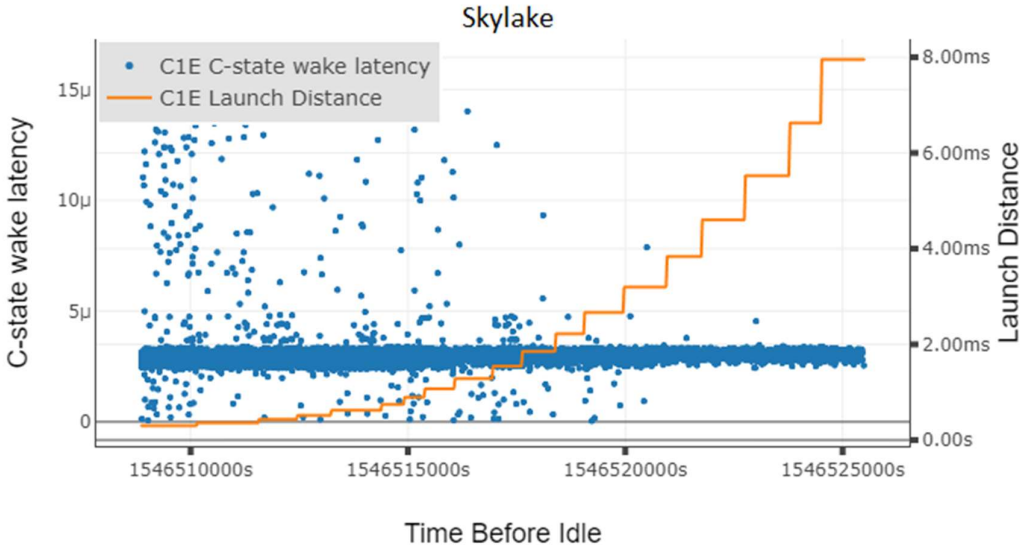



Broadwell
C3 C-state wake latency
C3 Launch Distance
C-state wake latency
70µ
60µ
50µ
40µ
30µ
20µ
10µ
0
Launch Distance
8.00ms
7.00ms
6.00ms
5.00ms
4.00ms
3.00ms
2.00ms
1.00ms
1546425000s
1546430000s
1546435000s
Time Before Idle

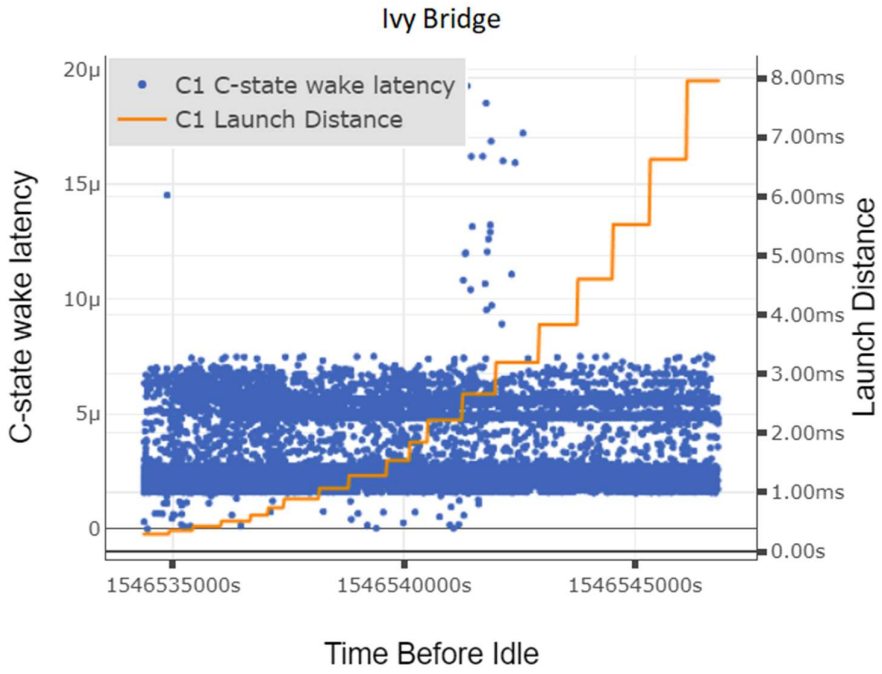
Appendix 5.3. C1E Wake Latency

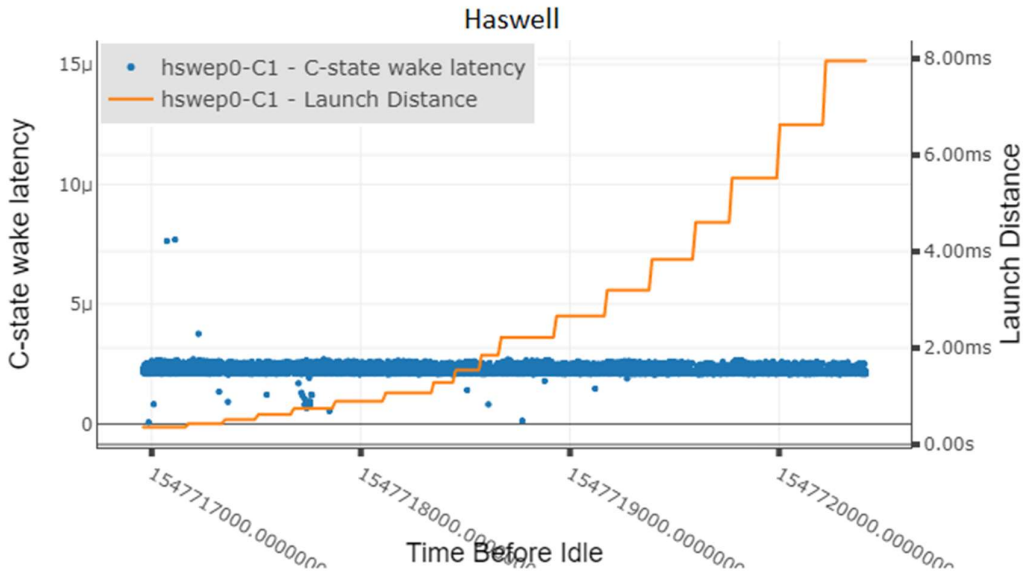


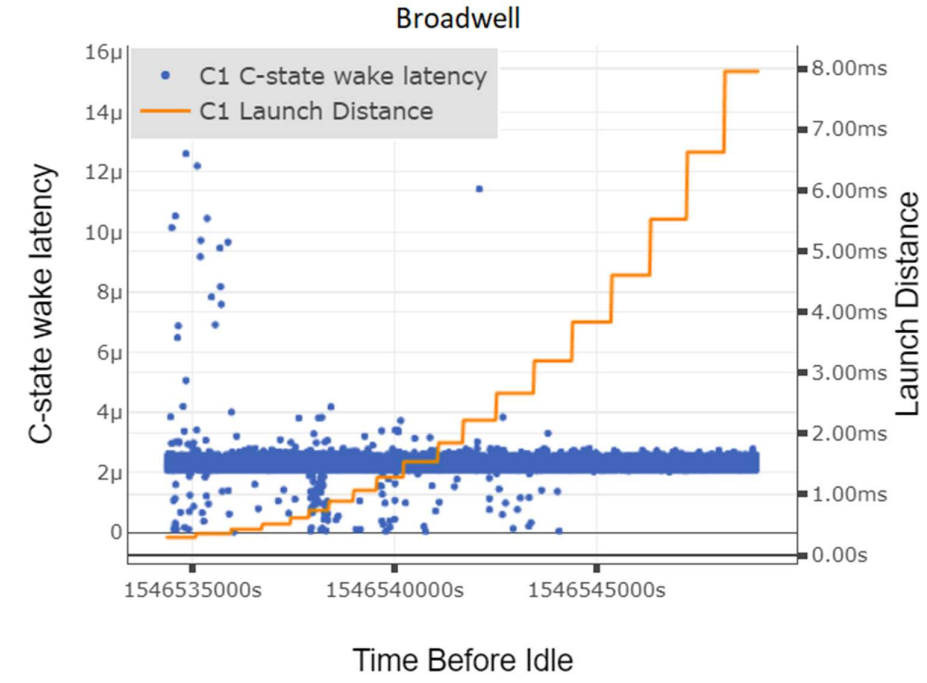

Broadwell
C1E C-state wake latency
C1E Launch Distance
C-state wake latency
15µ
10µ
5µ
0
8.00ms
7.00ms
6.00ms
5.00ms
4.00ms
3.00ms
2.00ms
1.00ms
0.00s
Launch Distance
1546510000s
1546515000s
1546520000s
1546525000s
Time Before Idle

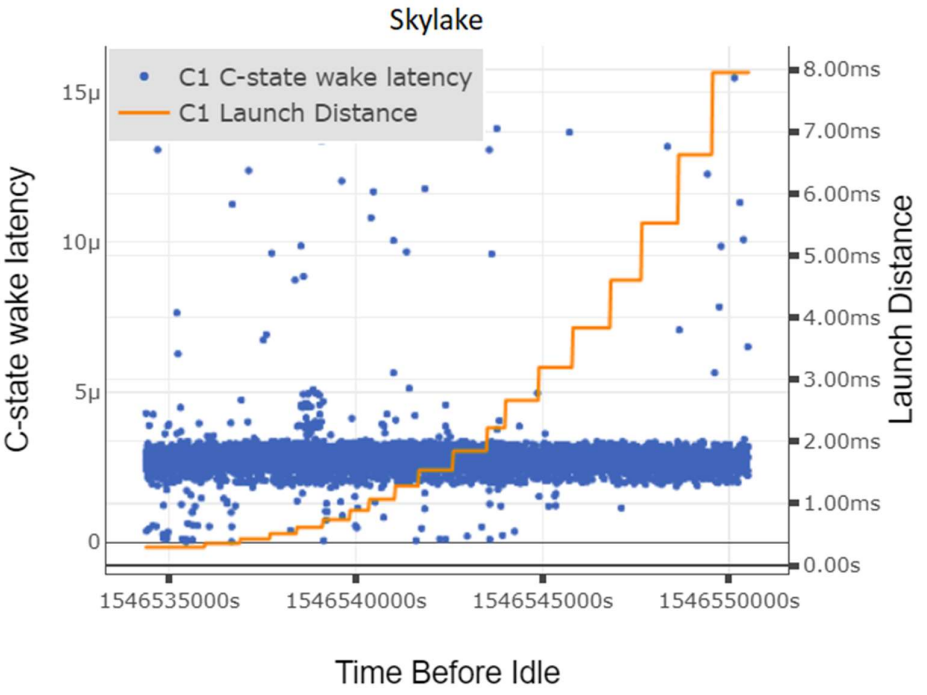

Skylake
C1E C-state wake latency
C1E Launch Distance
C-state wake latency
15µ
10µ
5µ
0
8.00ms
6.00ms
4.00ms
2.00ms
0.00s
Launch Distance
1546510000s
1546515000s
1546520000s
1546525000s
Time Before Idle

Appendix 5.4. C1 Wake Latency






Broadwell
C1 C-state wake latency
C1 Launch Distance
C-state wake latency
Launch Distance
16µ
14µ
12µ
10µ
8µ
6µ
4µ
2µ
0
8.00ms
7.00ms
6.00ms
5.00ms
4.00ms
3.00ms
2.00ms
1.00ms
0.00s
1546535000s
1546540000s
1546545000s
Time Before Idle


Skylake
C1 C-state wake latency
C1 Launch Distance
C-state wake latency
Launch Distance
15µ
10µ
5µ
0
8.00ms
7.00ms
6.00ms
5.00ms
4.00ms
3.00ms
2.00ms
1.00ms
0.00s
1546535000s
1546540000s
1546545000s
1546550000s
Time Before Idle