

# Investigating Power Supply Efficiency and Potential Savings of Standby Mode

Bachelor Thesis

Author: Elena Desiato

Tutors: Dr. Romain Jacob, Pascal Emmenegger

Supervisor: Prof. Dr. Laurent Vanbever

December 2025 to April 2026

## **Abstract**

Power Supply Units (PSUs) in routers and servers are frequently overprovisioned and underutilized. Two primary approaches to minimize power conversion losses are to employ PSUs with a lower capacity and to place redundant PSUs into standby mode. This thesis utilizes servers to explore PSU efficiencies at lower loads to provide insight into the first strategy and estimates potential savings from the second.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Task and Goals . . . . .	1
1.3	Overview . . . . .	2
<b>2</b>	<b>Background and Related Work</b>	<b>3</b>
2.1	Power Supply Units . . . . .	3
2.1.1	PSU Efficiency . . . . .	3
2.1.2	PSU Efficiency Standards . . . . .	3
2.1.3	Redundancy and Load Balancing . . . . .	5
2.2	Related Work . . . . .	5
<b>3</b>	<b>Methodology</b>	<b>6</b>
3.1	Measurement Sources . . . . .	6
3.2	Obtaining Internal Measurements . . . . .	7
3.2.1	BMC And IPMI . . . . .	7
3.2.2	Generating Artificial Loads . . . . .	7
3.2.3	get_measurements.py . . . . .	7
3.3	Obtaining External Measurements . . . . .	9
3.4	Cutting Power to One PSU . . . . .	10
3.5	Synchronization of Measurements . . . . .	10
3.6	Computing Efficiency . . . . .	10
3.7	Computing Power Savings . . . . .	11
3.8	Experimental Setup Details . . . . .	11
3.8.1	Non-cycled experiments . . . . .	11
3.8.2	Cycled experiments . . . . .	11
3.8.3	External Measurement Parameters . . . . .	12
3.8.4	Internal Measurement Parameters . . . . .	12
<b>4</b>	<b>Evaluation</b>	<b>13</b>
4.1	Verifying PSU-internal Measurements . . . . .	13
4.1.1	Comparing Internal and External Measurements . . . . .	13
4.1.2	Comparing Internal Input and Output Power Estimate . . . . .	16
4.1.3	Critical Assessment . . . . .	17
4.2	PSU Efficiency . . . . .	17
4.2.1	Efficiency Curves . . . . .	17
4.2.2	Critical Assessment . . . . .	18

4.3	Power Savings from Standby Mode . . . . .	21
4.3.1	Power Savings . . . . .	21
4.3.2	Critical Assessment . . . . .	22
<b>5</b>	<b>Outlook and Summary</b>	<b>24</b>
5.1	Outlook . . . . .	24
5.2	Summary . . . . .	24
	<b>References</b>	<b>25</b>
<b>A</b>	<b>My Appendix</b>	<b>I</b>
A.1	stress-ng commands to generate loads . . . . .	I
A.2	code snippets: get_measurements.py . . . . .	II
A.3	Code computing Load and Efficiency . . . . .	VI
A.4	Code computing Power Savings . . . . .	VI
A.5	Full Plot comparing Internal and Output Power . . . . .	VII

# Chapter 1

## Introduction

### 1.1 Motivation

According to the IEA Energy and AI report from 2025 [1], the electricity consumption of data centers is expected to more than double to 945 TWh by 2030, mainly driven by the demands of artificial intelligence. The increasing energy demands of data centers and the data transmission networks that serve them underscore the necessity of making them more efficient. Specifically, we take a closer look at the efficiency of Power Supply Units (PSUs), which are key components in both servers and network devices.

Although our main focus is on power consumption in routers and networks, the architecture of routers is very similar to that of servers. Therefore, we want to analyze PSUs in the context of servers, as they offer more predictable control over power draw than most networking hardware.

As Jacob et al. state in [2], the loads of their analyzed router PSUs range from 10% to 20%, and a router has multiple PSUs for redundancy. For that reason, we want to analyze the efficiency of PSUs at lower loads and the potential savings of using standby mode in particular.

### 1.2 Task and Goals

Previous work [3] has indicated that PSU-internal measurements in routers might be inaccurate. Therefore, before using internal measurements to compute efficiency, we must first analyze their validity and accuracy in the server setting. Then, we can proceed to gain a better understanding of efficiency curves in the analyzed PSUs, especially at lower loads, which are most relevant due to the overprovisioning of PSUs. Finally, knowing that PSU efficiency varies with load, we attempt to quantify potential power savings from forcing one PSU to supply the entire required power, and therefore operating at a higher load, while the other is on standby in case the first fails.

In short, the three research questions (RQs) we aim to answer are:

1. How trustworthy are PSU-internal measurements for the analyzed PSUs?
2. How efficient are those PSUs, especially at lower loads?
3. Are there potential power savings from having one PSU in standby mode?

## 1.3 Overview

Chapter 2 outlines the relevant background on PSUs and lists related work. Chapter 3 explains the design, methodology, and details of the performed experiments. The obtained data is evaluated and critically assessed in Chapter 4. Chapter 5 provides a brief summary of our findings, and future approaches are discussed.

## Chapter 2

# Background and Related Work

### 2.1 Power Supply Units

Power Supply Units (PSUs) are used in most electronics, from microwaves to laptops to industrial servers. Typically, wall outlets in Switzerland provide electricity at 230 Volts alternating current (AC). Most electronic devices require direct current (DC) at a significantly lower voltage, for example, 12 Volts for many servers. PSUs convert and step down the input voltage from the outlet to the voltage required by the server.

PSUs have a specific maximum output power, called the *capacity*  $C$ . A PSU's *load*  $L$  is defined as the ratio of output power  $P_{out}$  and capacity:

$$L = \frac{P_{out}}{C}$$

#### 2.1.1 PSU Efficiency

During the conversion process, PSUs lose energy as some electricity is converted into heat instead of usable power. *PSU efficiency*  $\eta$  characterizes how much energy is lost and is defined as the ratio of the output power delivered to the server  $P_{out}$  and the input power drawn from the wall socket  $P_{in}$ :

$$\eta = \frac{P_{out}}{P_{in}}$$

The efficiency of a PSU depends on the load a PSU is under, so usually we look at the entire PSU efficiency curves  $\eta(L)$ . These can vary depending on the PSU model, hardware degradation, and other factors such as temperature. Generally, the efficiency is highest at around 50% load. Figure 2.1 provides two examples of an efficiency curve for the *Corsair RM850x* PSU at different input voltages taken from a technological review [4]. Note that this is a different model from our PSUs, as they do not have any publicly available datasheet or review with efficiency curves.

#### 2.1.2 PSU Efficiency Standards

There are various certification programs for PSU efficiency. The most commonly used one is the 80 Plus<sup>®</sup> standard [5], where to qualify for a certain rating, a PSU must meet specific efficiency targets at specific loads, typically 20%, 50% & 100%, although a target at 10% has been added for the highest rating. Other less popular standards, such as the Cybenetics ETA, extend the 80 Plus<sup>®</sup> standard to cover the entire operating range of PSUs. The targets for various 80 Plus<sup>®</sup> certifications can be found in Figure 2.2.

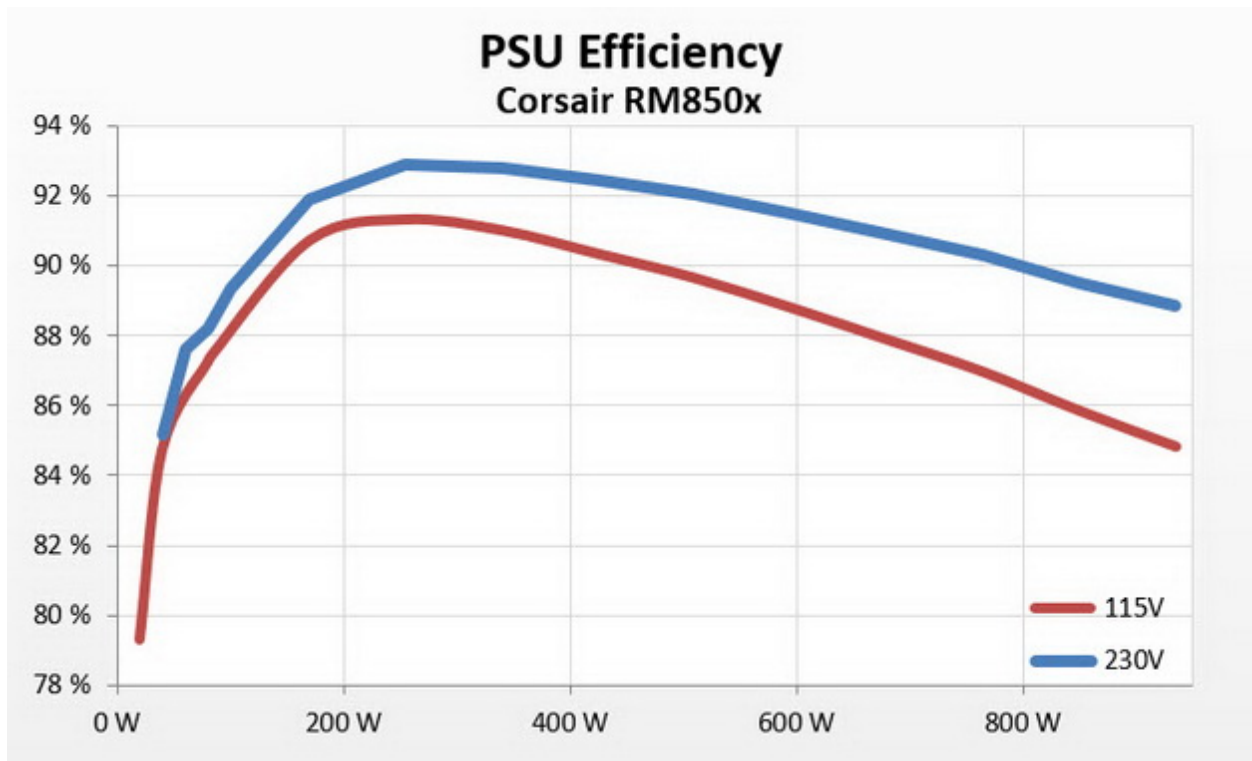


Figure 2.1: Example efficiency curves from a review for the *Corsair RM850x* PSU [4].

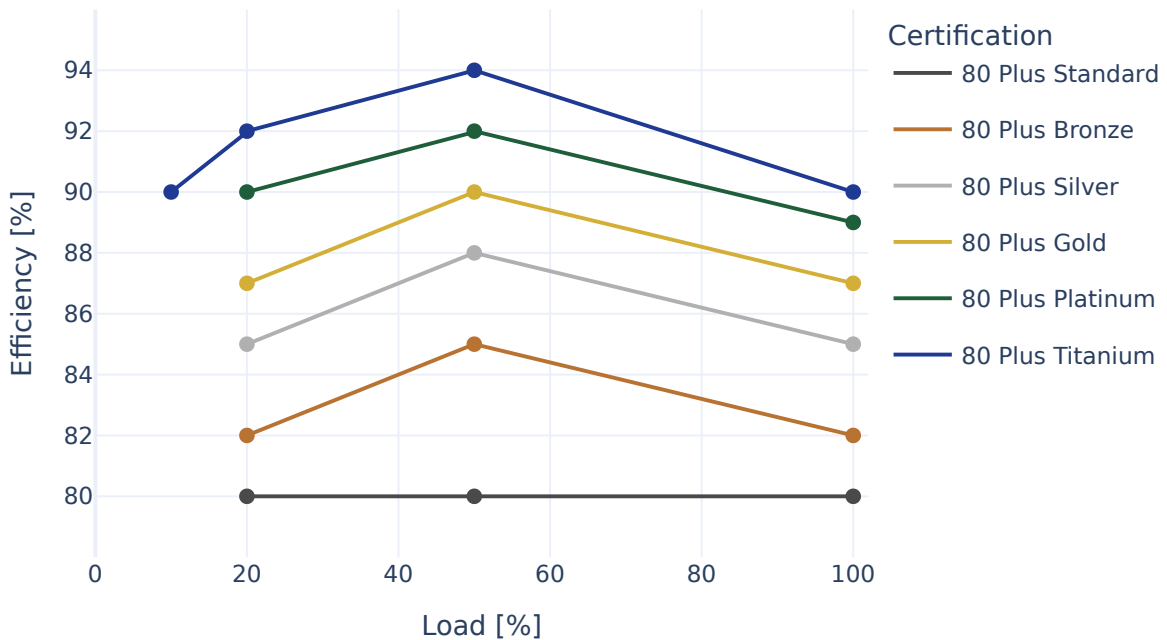


Figure 2.2: Different 80 Plus<sup>®</sup> certifications and their efficiency targets at certain loads.

### 2.1.3 Redundancy and Load Balancing

Modern servers use multiple PSUs instead of only one to achieve redundancy. If one PSU fails, the others can keep the server powered while the failed PSU is replaced. Let  $N$  be the number of PSUs required to handle server load. Common redundancy models include  $N+1$  (1 extra PSU) and  $N+N$  ( $N$  extra PSUs). There is a trade-off between the first, which is more cost-efficient, and the second, which is more fault-tolerant.

Redundant PSUs can be active, meaning the server draws power through them, or passive, meaning they are in standby mode unless the other PSUs fail. In configurations with multiple active PSUs, the server must distribute the load across all of them (load balancing). The availability of specific redundancy and load-balancing configurations depends on the server model and the manufacturer's firmware implementation.

## 2.2 Related Work

This work builds on previous student work [3], which examined network routers and found that router PSUs are underutilized and that there are potential power savings from using more efficient PSUs or switching from two PSUs to one PSU. This work builds on previous work by focusing on servers instead. On servers, we have better control over power draw, which we use to better understand PSU efficiency curves and estimate the potential benefits of standby mode.

# Chapter 3

## Methodology

This chapter describes the design of the experiments performed. We start by discussing internal and external measurement sources in Section 3.1 and how to gather data from them in Sections 3.2 and 3.3, including how to remotely cut off power to one PSU (Section 3.4) and how measurements from different sources are synchronized (Section 3.5). Then, in Sections 3.6 and 3.7, we discuss the computation of PSU efficiency and power savings from standby mode, along with the required assumptions. Finally, we examine the details of the experimental setup in Section 3.8.

### 3.1 Measurement Sources

As discussed in Section 2.1, PSUs convert high-voltage AC input into lower-voltage DC output. For our experiments, we obtained internal measurements from the PSU itself for input power and output current, from which we derived an estimate of the output power. We also utilized external measurements for input power. The various measurement sources are shown in Figure 3.1.

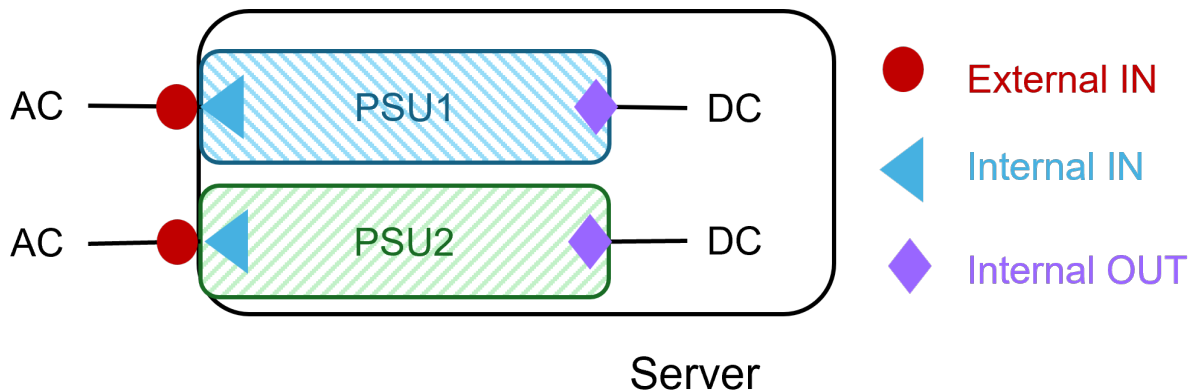


Figure 3.1: Diagram illustrating external input, internal input and internal output measurement sources.

## 3.2 Obtaining Internal Measurements

This section discusses the process of obtaining measurements from internal PSU sensors, such as PSU input power, PSU temperature, and PSU output current. It also describes the method used to artificially generate varying load levels on the target server.

### 3.2.1 BMC And IPMI

A Baseboard Management Controller (BMC) is a microcontroller on the motherboard of most modern servers. Among other functions, it commonly enables remote monitoring of the server out of band, i.e., without relying on the main CPUs. The BMC receives data from various sensors in the server, the availability and details of which vary across different manufacturers.

The Intelligent Platform Management Interface (IPMI) is an industry standard that defines a common interface to the BMC, regardless of the exact hardware, BIOS, or Operating System. For the Intel<sup>®</sup> servers used in this project, the Intel<sup>®</sup> Server Management Guide [6] confirms the existence of a BMC that supports the IPMI v2.0 standard and allows out-of-band (OOB) communication. The BMC has its own dedicated MAC and IP addresses. OOB communication is crucial for these experiments, as the measurements must be taken independently of the load the server's CPUs are under.

`ipmitool` [7] is a command-line tool that implements the IPMI protocol. It can be used locally or remotely over the network to interface with the BMC. It is available for various Linux distributions and, if not already included, can be easily installed through package managers. The measurements collected for this project use `ipmitool` over LAN. Therefore, the target's BMC must be reachable over the network from the device collecting the measurements, the device running the measurements must have `ipmitool` installed, and an `ipmitool` user must be set up with the appropriate permissions.

### 3.2.2 Generating Artificial Loads

Although standardized PSU efficiency testing, as in the 80 Plus<sup>®</sup> certification program, is based on electronic load banks, this is beyond the scope of this study. Instead, we rely on artificial loads generated by software, specifically those generated by the `stress-ng` tool [8]. `stress-ng` is a Linux command-line tool for stress-testing a computer system that can be installed with package managers. For these experiments, it must be installed on the target system.

`stress-ng` has many capabilities, but we will limit ourselves to slowly increasing CPU load by increasing a number of stressors performing matrix multiplication and using virtual memory stressors to utilize RAM. Through experimentation, we have found that this provides the most stable power draw at discrete intervals and that any additional or alternative stressors do not further increase the power draw. We also ensured the commands were adjusted to the number of CPU cores on each server. All `stress-ng` commands that are run to generate various loads can be found in Appendix A.1.

### 3.2.3 `get_measurements.py`

The Python script `get_measurements.py` was used to generate workloads on the target system and obtain internal measurements.

## Script Parameters

The script takes in various parameters either as command-line arguments or by using the `--config` flag and passing a configuration `.json` file. The most important ones are listed in Table 3.1.

Table 3.1: Configuration Parameters

Parameter	Description
<code>test_duration</code>	Duration of each test in seconds.
<code>num_measurements</code>	Number of measurements per test/command.
<code>warmup_time</code>	Delay to begin measurements after starting load in seconds.
<code>psu_cycle</code>	Whether the PSU should be cycled.
<code>phase_duration</code>	Duration of each phase in seconds when PSU is cycled.
<code>num_measurements_per_phase</code>	Number of measurements per phase when PSU is cycled.
<code>warmup_each_phase</code>	Whether phases 2 and 3 also get warmup time or just phase 1.
<code>runs</code>	Number of experiment (all commands once) repetitions.
<code>random_shuffle</code>	Whether to randomly shuffle stress commands list.

There are also configuration parameters that specify the BMC IP, the IPMI user, the Smart Plug's IP, and SSH configuration for the target server and the Raspberry Pi. These must be defined, and the corresponding IPMI user's password must be passed to the script as the environment variable `IPMI_PASS`.

## Overview

`ipmitool` over LAN is used to obtain hardware and sensor data from the target machine's BMC, which requires root permissions on the controller. `ipmitool sdr` is used to collect sensor data from all available sensors, all of which are put into `.csv` files. Before starting the experiment, the script also runs `ipmitool fru` on the target to obtain information on Field Replaceable Units, which includes information about the target PSUs.

Generating different loads on the target machine is done by using `ssh` to run `stress-ng` commands on the target machine; hence, SSH keys must be set up beforehand.

Based on the `runs` argument, the script will perform multiple runs. Each run will perform measurements on loads generated by each `stress-ng` command exactly once, by default in a randomized order unless `random_shuffle` is set to `false`.

If `psu_cycle` is `false`, the script will not cycle one of the PSUs. For each test using a single command, `stress-ng` stressors are started on the target server, and there's a delay of `warmup_time` before measurements begin. There are `num_measurements` measurements taken, evenly spaced over the `test_duration`, not including the warm-up time.

Setting `psu_cycle` to `true` performs measurements that involve a PSU being power cycled by using the Smart Plug's HTTP interface. In this case, each test has three phases: first, both PSUs are powered; then one PSU is disconnected; and finally, it is reconnected. Each phase has a length of `phase_duration`, and `num_measurements_per_phase` measurements are evenly spaced during it. If `warmup_each_phase` is `true`, the script waits for `warmup_time` before measurements in all three phases. Otherwise, there is only a warm-up period before the first phase.

Synchronization with external measurements is facilitated by including two obvious synchronization spikes at the beginning and end of each run. Each synchronization spike consists of 10 seconds at idle load, followed by 10 seconds at high load, then 10 seconds of idle load again.

In summary, this script performs a single experiment with multiple runs. Each run is framed at the start and end by obvious synchronization spikes and performs a test for each listed load-generating command exactly once. Each test involves starting a load on the target server, potentially waiting for a warm-up time, and then taking periodic measurements of sensor data. The key parts of the script can be found in Appendix A.2.

### Setup Requirements

The following are all requirements to successfully run this script.

#### Controller:

- `ipmitool` must be installed.
- Must have root permissions. The script will prompt for password input when first needed.
- All configuration parameters (BMC host, IPMI user, Smart Plug IP, SSH alias for target server, and SSH alias for Raspberry Pi) must be known and passed to the script.

#### Target server:

- `stress-ng` must be installed.
- The target BMC must be configured to have a known IP address that is accessible from the controller. An `ipmitool` user must also be configured. Both the IP and user need to be passed to the script via the configuration file or command-line arguments. The corresponding password is passed to the script as an environment variable `IPMI_PASS`.

#### Other:

- An SSH key pair must be generated for the controller and target server, including an entry in the SSH config file of the controller specifying the SSH configuration. The hostname alias is passed to the script.
- An SSH key pair must be generated for the controller and the Raspberry Pi, including an entry in the SSH config file of the controller specifying the SSH configuration. The hostname alias is passed to the script.
- The Raspberry Pi must be configured to act as an Access Point, to which the Smart Plug is connected, and must be reachable from the controller. The Smart Plug's local IP address must be known and passed to the script.

## 3.3 Obtaining External Measurements

Using the Autopower system [9] from previous work, we obtained external input power measurements for the PSUs. Using a Raspberry Pi, Autopower collects power readings from a power meter connected to a PSU at a specified sampling frequency. These readings are in milliwatts and include a timestamp per measurement sample. This data is periodically sent to a server, and .csv files can be downloaded from a web-based management interface.

### 3.4 Cutting Power to One PSU

For the third research question, it is useful to be able to remotely cut power to one of the two PSUs connected to the server. To achieve this, a myStrom WiFi Switch (Smart Plug) is inserted between the PSU plug and the outlet. The Smart Plug can be controlled via HTTP requests to an API, which requires it to have a known IP address on a WLAN network. The university network is not suitable due to security measures that prevent a direct connection, so we set up a Raspberry Pi as a WLAN access point, connected the Raspberry Pi to the university network via Ethernet, and connected the Smart Plug to the Raspberry Pi's network using the MyStrom Smartphone Application. Then, the Smart Plug can be toggled on and off by sending HTTP requests from the Raspberry Pi, which can be accessed through an SSH connection.

### 3.5 Synchronization of Measurements

As we have measurements from different sources, we have to synchronize them. We take the approach of synchronizing on the data itself by forcing an obvious power spike on which we can align timestamps of internal and various external measurements. In particular, we take the median of a number of baseline idle power measurements, then detect the first measurement that exceeds the baseline by at least a specified delta, defined per server. Then we align the timestamps of all first-detected measurements from the various measurement sources.

After synchronizing the timestamps, we use the first and last internal measurement timestamps for each test to create an interval. As our final per-test measurement values, we take the medians of all measurements that fall into the corresponding time interval. We use the median for improved robustness to outliers.

### 3.6 Computing Efficiency

To compute efficiency values, we require input and output power. External measurements only provide a way to measure the input power, so in any case, we require internal measurements to measure the output power of the PSU. To remain consistent, we also made the choice to use internal input power measurements  $P_{in}$ .

For our servers, the only internal sensors provided linked to output power are sensors that measure the output current  $I_{out}$  as a percentage of the maximum. Therefore, we can only estimate the output power. In particular, we have to assume the maximum output current  $I_{max}$  and PSU capacity  $C$  based on the PSU label and assume that the output voltage is fixed at 12V. Then the output power is estimated as  $P_{out} = I_{out} * (I_{max}/100) * 12V$  and we can use the definitions of load and efficiency to compute their values:

$$L = \frac{P_{out}}{C} = \frac{I_{out} * (I_{max}/100) * 12V}{C}$$

$$\eta = \frac{P_{out}}{P_{in}} = \frac{I_{out} * (I_{max}/100) * 12V}{P_{in}}$$

The corresponding code can be found in Appendix A.3.

## 3.7 Computing Power Savings

Adjusting the load balancing or redundancy configurations of our servers is not possible due to firmware limitations. We can only connect or disconnect PSUs and observe that if both PSUs are connected at idle load (around 10 percent of one PSU’s capacity), both servers automatically place one PSU on standby while the other provides all the power. Eventually, as a sufficiently high load is reached on one PSU, around 30-35%, the server will utilize both PSUs and share the load almost equally.

Now we want to estimate the potential power savings from forcing one PSU to remain passive even as the load increases, rather than having the second PSU automatically become active. This is possible because the total load never reaches 50% of one PSU’s capacity; therefore, one PSU can certainly handle it. First, we estimate the standby power consumption of each PSU by observing the power drawn by passive PSUs when the load is low. There are no cases where only PSU2 is on; therefore, we only do this for PSU1 and assume that standby power is the same for both analyzed PSUs and is constant and independent of load.

Second, we look at various high-load cases with load levels  $L$  and compute the difference in the total input power of both PSUs when both are on versus the input power of the remaining PSU, and the input power of the standby PSU after one PSU is forcibly disconnected during a cycled experiment. To determine the total input power drawn by both PSUs, we take the average of the medians of the phases before the PSU is disconnected and after the PSU is reconnected.

$$P_{\text{savings}}(L) = (P_{\text{in, one active}}(L) + P_{\text{in, standby}}) - P_{\text{in, both active}}(L)$$

As we only require data on the input power, we use external measurements, which are more likely to be accurate. To determine whether a PSU is considered on standby and not actively supplying power, we use a fixed threshold of 10 Watts. This is sufficient on our server, as the standby power draw is significantly lower and the power draw when the PSU is active is markedly higher.

The Python code for this is available in Appendix A.4.

## 3.8 Experimental Setup Details

We ran experiments on two servers *snowball* and *delirium*, both with exactly two PSUs but slightly different hardware. *snowball* has two PSUs of the *DPS-750XB A* model with a capacity of 750W and a maximum output current of 62.0A on the main rail, according to the information label on the PSU itself. *delirium* has two PSUs of the *DPS-500WB-1 A* model with a capacity of 450W and a maximum current of 36.26A on the main rail. Both servers are Intel<sup>®</sup> servers running Ubuntu 24.04.3 LTS, and all PSUs were manufactured by Delta Electronics.

### 3.8.1 Non-cycled experiments

These experiments aim to validate PSU-internal measurements and assess PSU efficiency. Therefore, to remove the impact of the server’s load balancing, one PSU is unplugged, leaving the remaining PSU to power the server. The external measurement setup is connected to the plugged-in PSU. Experiments are performed twice per server, once per PSU.

### 3.8.2 Cycled experiments

Both PSUs are plugged in and connected to external measurement devices. PSU1 is plugged in via the Smart Plug, so it can be disconnected.

### 3.8.3 External Measurement Parameters

Using the Autopower web interface, the sampling frequency of external measurements was set to 2Hz or one sample every 500 ms.

### 3.8.4 Internal Measurement Parameters

Values chosen for the key parameters in the non-cycled experiments (RQ1 & RQ2) can be found in Table 3.2 and those for the cycled experiments (RQ3) can be found in Table 3.3.

Table 3.2: Parameter Values for non-cycled experiments (RQ1 & RQ2)

Parameter	Value
<code>psu_cycle</code>	false
<code>test_duration</code>	20s
<code>num_measurements</code>	10
<code>warmup_time</code>	0s
<code>random_shuffle</code>	true
<code>runs</code>	10

Table 3.3: Parameter Values for cycled experiments (RQ3)

Parameter	Value
<code>psu_cycle</code>	true
<code>phase_duration</code>	10s
<code>num_measurements_per_phase</code>	5
<code>warmup_time</code>	30s
<code>warmup_each_phase</code>	true
<code>random_shuffle</code>	true
<code>runs</code>	10

Based on initial testing, we found that when running the chosen `stress-ng` commands, the power draw nearly instantaneously reaches a target power level that remains stable over time. This justifies shorter test durations, fewer measurements, and no warm-up time. The only exception is temporary transient effects during the start of the second phase of cycled experiments, after disconnecting one PSU, which is why these require a warm-up time prior to each phase’s measurements. These effects will be discussed in more detail in Section 4.3.2. Finally, it is important that `random_shuffle` is set to `true` to avoid ordering bias.

# Chapter 4

## Evaluation

This chapter will discuss the results of experiments designed to explore each research question individually, including a critical analysis. First, Section 4.1 assesses the validity of internal PSU measurements, and Section 4.2 uses those measurements to estimate efficiency curves. Finally, Section 4.3 utilizes external measurements to approximate power savings from standby mode.

### 4.1 Verifying PSU-internal Measurements

In this section, we are interested in evaluating the accuracy of PSU-internal measurements. First, we compare internal and external measurements of the input power. Then we compare internal input power and our estimate of the output power based on internal output current measurements.

#### 4.1.1 Comparing Internal and External Measurements

After synchronizing the internal and external measurements using the process described in Section 3.5, all the internal and external measurements of the input power can be compared. Figure 4.1 shows this comparison for six sample test cases for one command each. Each segment of yellow data points represents one run of a load-generating `stress-ng` command. Each cluster of 10 consecutive blue data points represents the internal measurements for that particular command. This interval from the first internal measurement to the last is the one over which medians are computed for both internal and external measurements. Some internal measurements may fall within the transient period after the previous command, as there is no warm-up time. However, this is not an issue, as we use medians, and the power draw remains stable otherwise.

We now compare internal and external measurements of input power by comparing the medians of all measurements within each interval defined by the first and last internal measurements corresponding to a particular test. The absolute and relative differences between the medians of internal and external measurements can be found in Figure 4.2.

These plots show that the constant offset between internal and external power measurements is smaller than in previous work on routers [3]. External measurements are up to 6W larger than internal measurements, though a few potential outliers are also smaller. There are also variations in accuracy between servers or PSU models. For example, the delirium PSUs (orange and blue) typically draw less power than the snowball PSUs, resulting in larger relative differences. The differences in external and internal measurements could be explained by measurement noise, calibration issues, or the limited resolution of internal measurements, the latter of which is discussed in Section 4.1.3.

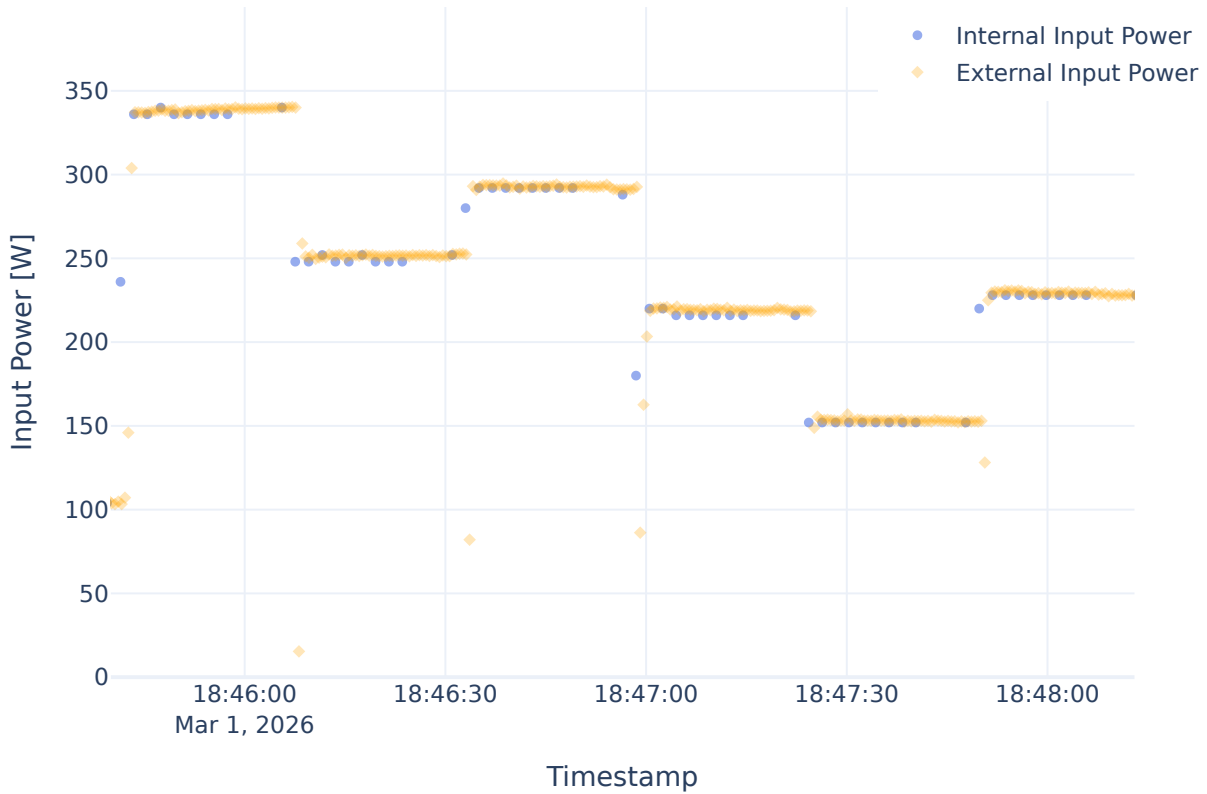
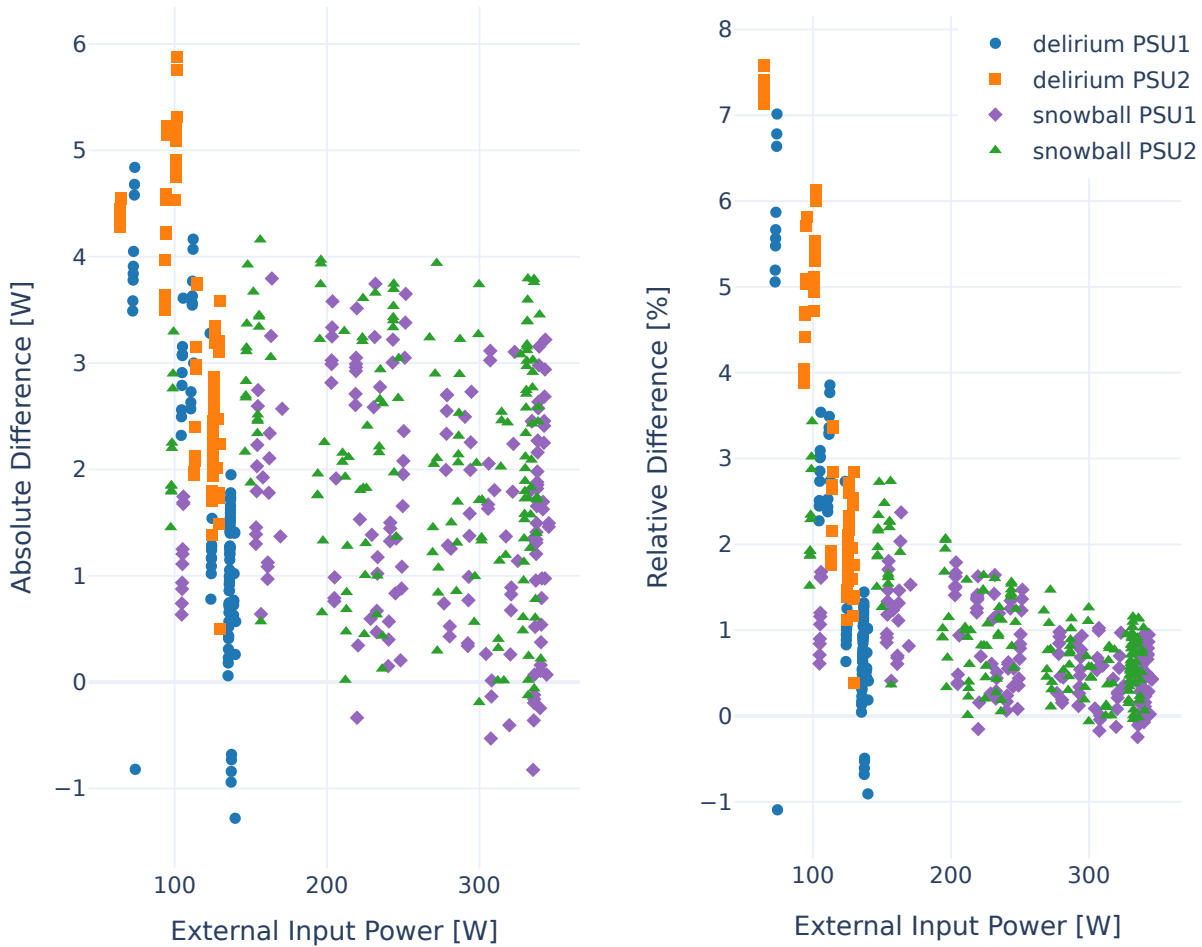


Figure 4.1: Comparison of internal and external measurements of input power for six sample tests (one load level each) on PSU1 of snowball. Each set of 10 consecutive blue data points defines one interval over which the medians of internal and external input power are computed. The stability of the measurements justifies taking the median over such intervals.



(a) Absolute differences.

(b) Relative differences.

Figure 4.2: Absolute and relative differences of medians of internal and external input power measurements per test (external - internal). delirium PSUs exhibit larger relative differences than snowball PSUs due to their lower power draw. The offset between internal and external measurements is smaller than in previous work.

### 4.1.2 Comparing Internal Input and Output Power Estimate

Now we compare the internally measured input power with the estimate of the output power based on the output current. For clarity, we only consider a small section of a graph that compares the internally measured input power and our estimate of the output power based on the output current in Figure 4.3. The entire plot is shown in Appendix A.5 as a representative example for all analyzed PSUs. We can see that the input and output powers move mostly in sync, with the input power higher than the output power. This is as expected due to losses in the PSU energy conversion process. Occasionally, we see small changes in input power that are not reflected in output power, which is likely explained by the resolution differences explored in Section 4.1.3.

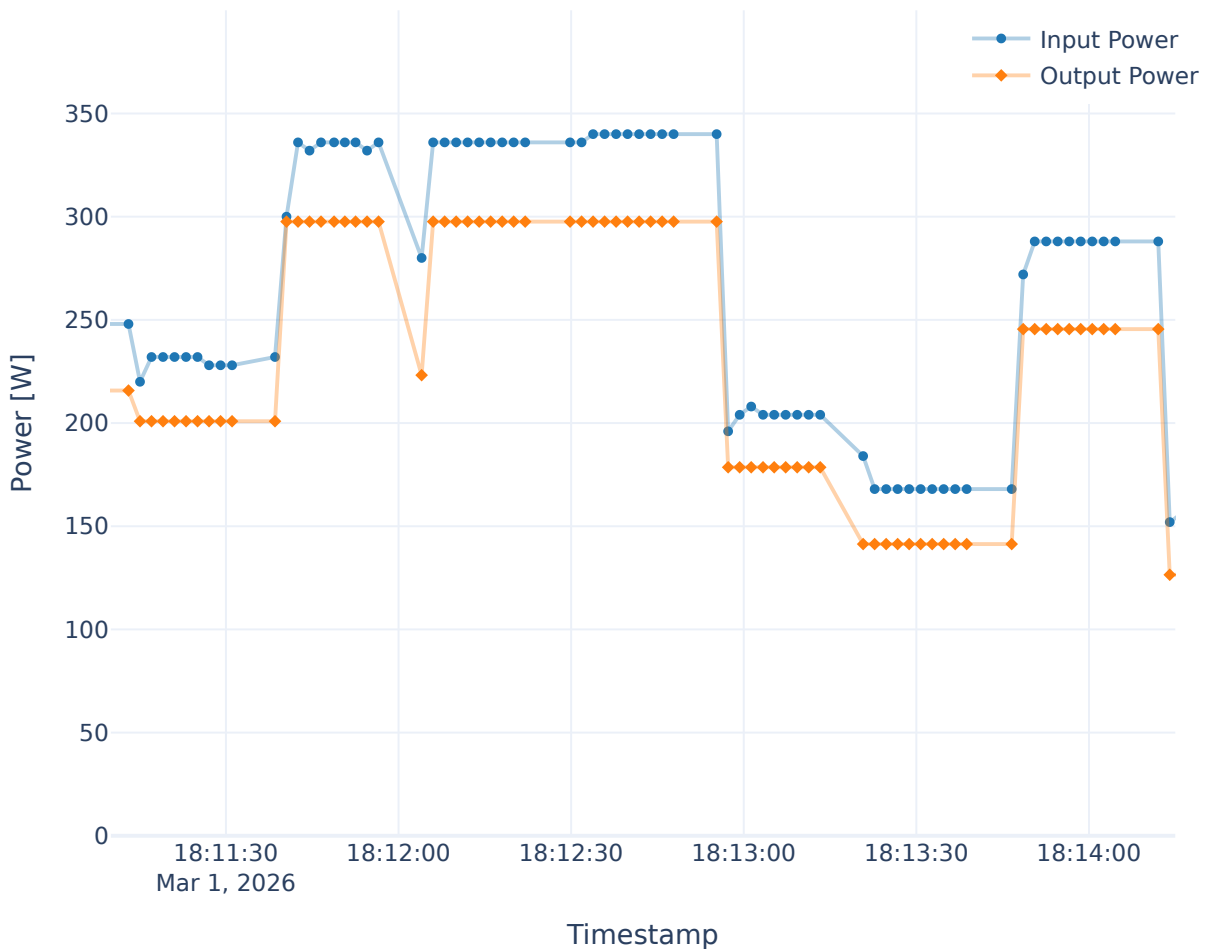


Figure 4.3: Subsection of plot comparing internal input power and estimated output power for PSU1 on snowball. Input and output power are mostly synchronized, with output higher than input, as expected. There are some small changes in the input where the output remains flat.

### 4.1.3 Critical Assessment

#### Limited Sampling Frequency

Internal measurements have a sampling frequency limited to 0.5 Hz due to computational overhead. In practice, this is not an issue, as the power level remains stable during a single-command test, and we are not interested in transient effects when using internal measurements for research questions one and two.

#### Limited Measurement Resolution

The more pressing issue is that internal measurements are limited in resolution, especially our estimate of the output power. The internal PSU sensor data has a resolution of  $\Delta P_{\text{in}} = 1W$  for the PSU input power and  $\Delta I_{\text{out}} = 1\%$  for the PSU output current as a percentage of the maximum.

Using the formula used to estimate the output power in section 3.6, we can estimate the resolution of the output power  $\Delta P_{\text{out}}$ .

$$\begin{aligned} \Delta P_{\text{out}} &= P_{\text{out}}(I_{\text{out}} + \Delta I_{\text{out}}) - P_{\text{out}}(I_{\text{out}}) \\ &= ((I_{\text{out}} + 1) * (I_{\text{max}}/100) * 12V) - (I_{\text{out}} * (I_{\text{max}}/100) * 12V) \\ &= \frac{I_{\text{max}}}{100} * 12V = \begin{cases} 7.44W & \text{if } I_{\text{max}} = 62.00A \quad (\text{snowball}) \\ 4.3512W & \text{if } I_{\text{max}} = 36.26A \quad (\text{delirium}) \end{cases} \end{aligned}$$

To summarize, the external input power measurements have a resolution of 1mW, the internal input power measurements have a resolution of 1W, and the internal output power estimate has a resolution of roughly 4.4W or 7.4W. The different resolutions likely explain at least some of the differences between internal and external input power measurements in Section 4.1.1, as well as the asynchronicity in Section 4.1.2.

## 4.2 PSU Efficiency

In this section, our goal is to obtain an idea of the efficiency curves and their potential differences among the four analyzed PSUs.

### 4.2.1 Efficiency Curves

Figure 4.4 shows the efficiency values plotted against the load for the four PSUs analyzed. The load and efficiency values were calculated according to Section 3.6 using the median values for input power and output current per test, i.e., one command run.

We can see differences between PSUs. For example, comparing the two snowball PSUs, PSU1 (purple) is consistently below PSU2 (green). Likewise, on the delirium server, PSU1 (blue) is consistently lower than PSU2 (orange). Given that PSU1 seems to be the default active PSU on both servers, i.e., the one that always provides power to the server even at low loads when only one PSU is active, this suggests that other factors, such as temperature or hardware degradation, may affect efficiency.

We can also see that there is some spread between efficiency values at the same load. As an example (indicated by a red circle), the delirium PSU2 at around 18% load has efficiency values ranging from 74% to 86%.

Finally, we notice that no PSU meets the 80 Plus Platinum efficiency target of 90% at 20% load. Based on a label on the PSUs, we know that at least the two snowball PSUs are Platinum-certified,

yet they do not reach this target. This may be due to different testing conditions, measurement inaccuracies, or aging of the PSUs since they were officially tested for the standard.

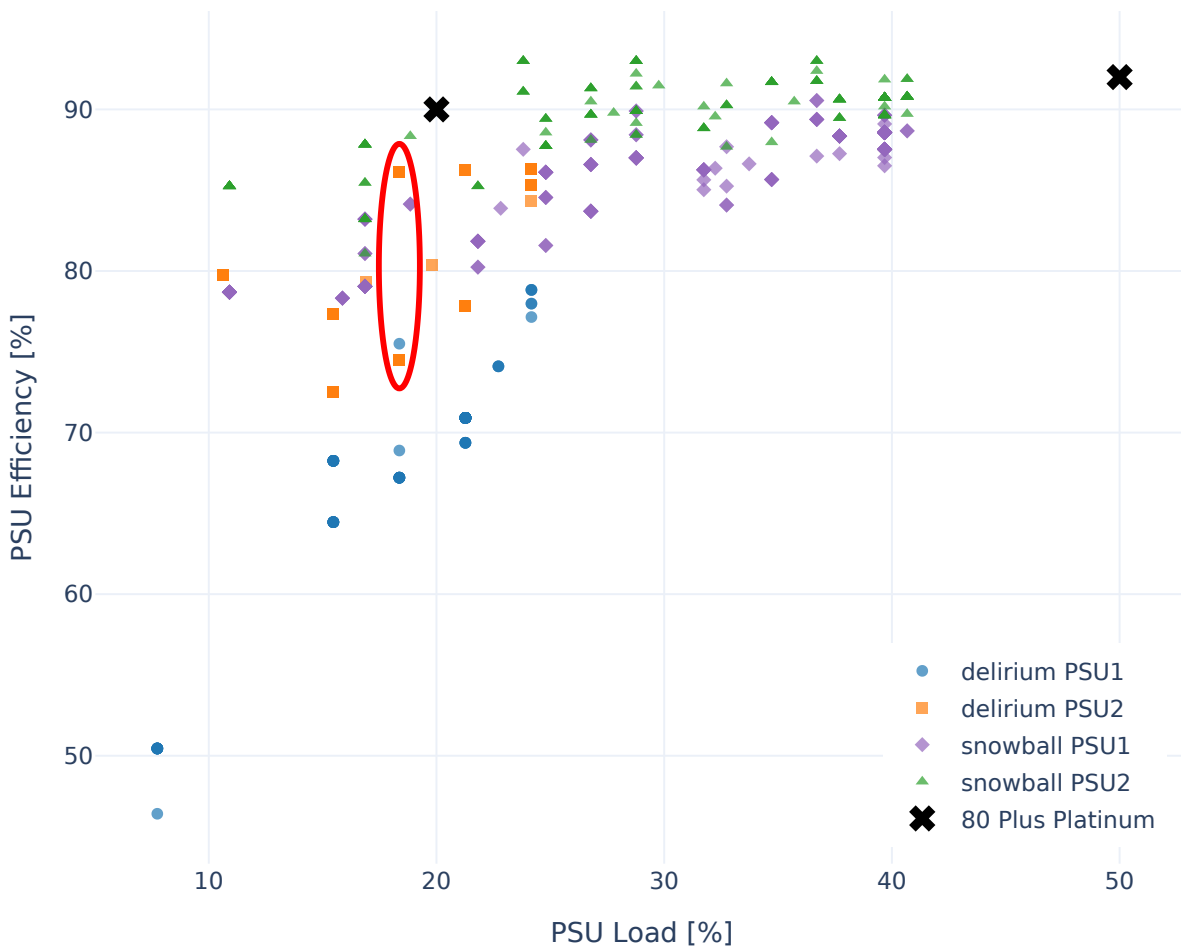


Figure 4.4: Efficiency values of the four PSUs at various loads and the two relevant targets for the 80 Plus Platinum Certification. There is a spread in efficiency values at the same loads (one example is circled in red). There are differences between the default active PSU1 and PSU2 on each server, and the platinum-certified snowball PSUs do not reach the Platinum efficiency target at 20% load.

### 4.2.2 Critical Assessment

There are a few reasons to be critical of the efficiency plots above that must be discussed.

#### Limited Measurement Range

We were mostly unable to generate loads below 10% and above 41%. This limits our plot of the efficiency curves to the range of 10-41%. However, this clearly includes the range in which these PSUs usually operate, as we were unable to achieve a higher, consistent power draw on the servers, even when using only one PSU. Therefore, this is the most relevant range.

### Assumptions

The calculation of load and efficiency values relies heavily on assumptions about the PSU's maximum output current and power. For our PSUs, these were taken from labels on the back of the PSUs, which may be inaccurate.

We also assume the output voltage to be fixed at 12V, though in reality it fluctuates slightly. According to BMC measurements on the snowball, the motherboard voltage was either 12.5V or 12.2V during the relevant experiments. On delirium, it was either 11.78V or 11.83V. However, these values do not account for the energy lost from the PSU to the motherboard, and the measurements may not be accurate either. If the voltage is below the assumed 12V, we overestimate the output power and therefore the efficiency. Similarly, if the voltage is above 12V, we underestimate the efficiency.

### Limited Measurement Resolution

As discussed when analyzing the validity of PSU-internal measurements, there is also the issue of limited measurement resolution in both input and output power used for efficiency computation as described in Section 3.6. Continuing the calculations done in Section 4.1.3, we can obtain an estimate of the efficiency resolution  $\Delta\eta$  using the formula for the propagation of errors:

$$\begin{aligned}
 \Delta\eta(P_{\text{in}}, P_{\text{out}}) &= \left| \frac{\delta}{\delta P_{\text{in}}} \eta(P_{\text{in}}, P_{\text{out}}) \right| * \Delta P_{\text{in}} + \left| \frac{\delta}{\delta P_{\text{out}}} \eta(P_{\text{in}}, P_{\text{out}}) \right| * \Delta P_{\text{out}} \\
 &= \left| \frac{\delta}{\delta P_{\text{in}}} \frac{P_{\text{out}}}{P_{\text{in}}} \right| * \Delta P_{\text{in}} + \left| \frac{\delta}{\delta P_{\text{out}}} \frac{P_{\text{out}}}{P_{\text{in}}} \right| * \Delta P_{\text{out}} \\
 &= \left| -\frac{P_{\text{out}}}{P_{\text{in}}^2} \right| * \Delta P_{\text{in}} + \left| \frac{1}{P_{\text{in}}} \right| * \Delta P_{\text{out}} \\
 &= \frac{P_{\text{out}}}{P_{\text{in}}^2} * 1W + \frac{\Delta P_{\text{out}}}{P_{\text{in}}}
 \end{aligned}$$

This formula gives us an estimate of the worst-case error due to limited measurement resolution, expressed as a fraction. Multiplying by 100, we obtain the percentage value that is then included in Figure 4.5. In the context of PSU efficiencies, single-percentage changes are very important. For example, according to the 80 Plus Standard, the difference between a Gold and Platinum classification is 2% or 3% at certain loads. However, just the errors discussed above range from 2.5% at the highest loads to 6-8% at the lowest loads.

### Other Factors affecting Efficiency

There may also be other factors influencing efficiency, such as temperature or PSU wear and tear. Using the BMC and IPMI, we were able to obtain internal temperature measurements for each PSU, but only at a resolution of 1 °C. Plotting the median temperatures per test against the efficiency values per test, as seen in Figure 4.6, shows no correlation between temperature and efficiency. However, this may also be due to the limited resolution of the temperature measurement.

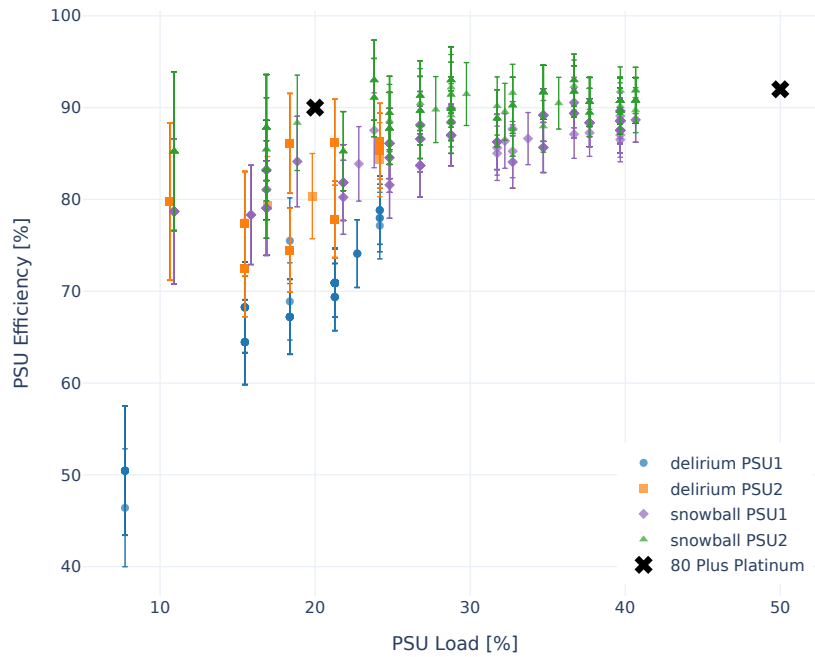


Figure 4.5: Efficiency values of the four PSUs at various loads with error bars for the errors caused by limited measurement resolution for each data point. At the lowest loads, the error is between 6% and 8%, while at the highest loads it is at around 2.5%.

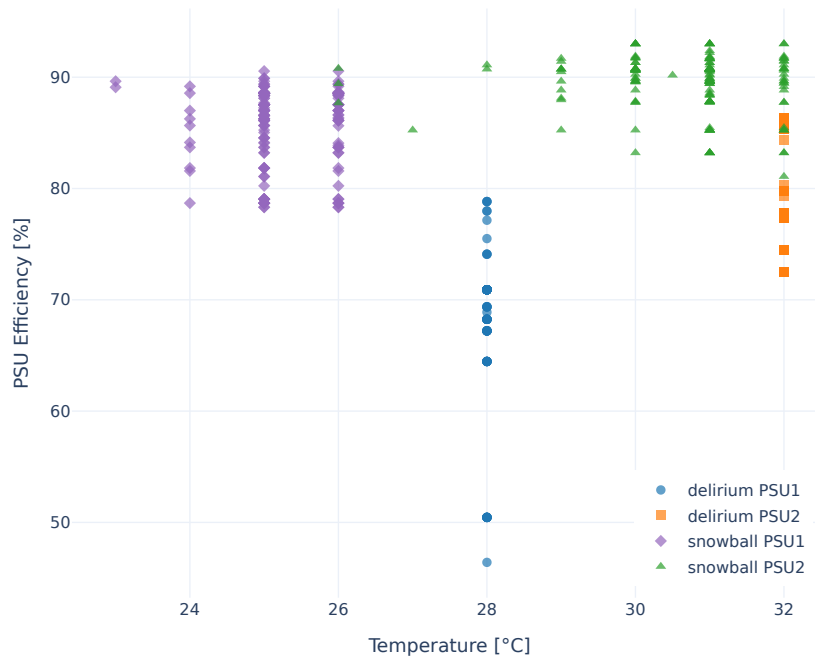


Figure 4.6: Efficiency values plotted against temperature (median per test) per PSU. This shows no correlation, but the measurement resolution is also limited.

## 4.3 Power Savings from Standby Mode

In this section, we estimate the power savings from forcing one PSU to remain on standby even at higher loads, rather than splitting the power load between both units after reaching roughly 30% of one PSU's load.

### 4.3.1 Power Savings

Using the procedure described in Section 3.7, we use external measurements to compute potential savings at a few different values of total input power. The results can be found in Figure 4.7. We can see that the savings are around 10 Watts, or roughly 2.9 percent of the total input power draw.

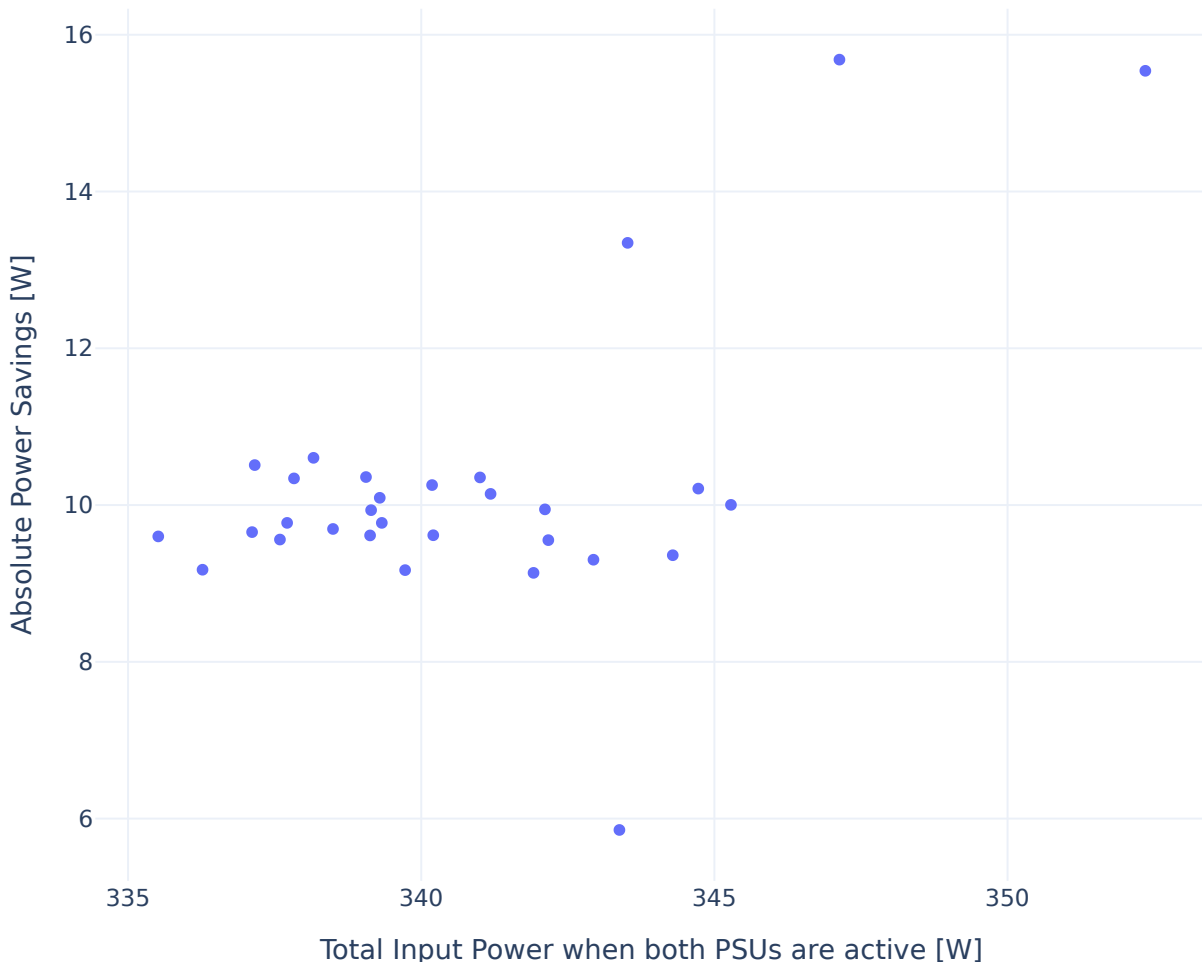


Figure 4.7: Power savings for a small range of high power draws. The savings are around 10W or roughly 2.9% of the total input power.

### 4.3.2 Critical Assessment

#### Limited Power Range

Due to our limited control over the server's power load balancing across both PSUs, we cannot estimate power savings over a wide range of input power values. However, the range we investigated is the most relevant, as it is where, before disconnecting and after reconnecting one PSU, the load is shared between both PSUs. So, this is exactly where one PSU could be kept on standby instead.

#### Assumptions

Our main assumption is that the estimated standby power draw of PSU2 at lower loads is also the standby power draw of PSU1 at higher loads. As the two PSUs are of the same model and standby power should not be proportional to load, this is a reasonable assumption. However, it may still introduce errors. This assumption may not account for PSU1 being the default active PSU and therefore potentially suffering greater degradation, or that load affects standby power, for example, due to temperature changes.

#### Power Spike after disconnecting PSU

After disconnecting PSU1 via the Smart Plug, the remaining PSU sometimes experiences a significant power spike. Examples of this can be seen in Figure 4.8, which shows all external input power measurements during the first 30 minutes of a cycled experiment. The gray dotted lines separate different tests, and we can see that sometimes, after PSU1 (blue) is forcibly disconnected, the remaining PSU's power increases by up to 50W for around 10 seconds. This may be explained by the server's emergency response to suddenly losing power to one PSU, but it is a temporary effect that is not of interest in the long run. We remove this effect from our data by waiting 30 seconds for warm-up before using the measurements for computations.

#### Other Effects of Standby Mode

We have not investigated other effects that forcing one PSU to take on all of the load longer may have. The active PSUs may experience increased wear and tear, and it is unclear whether a single PSU can handle sudden power spikes as well as two PSUs. There may also be consequences for the server's performance.

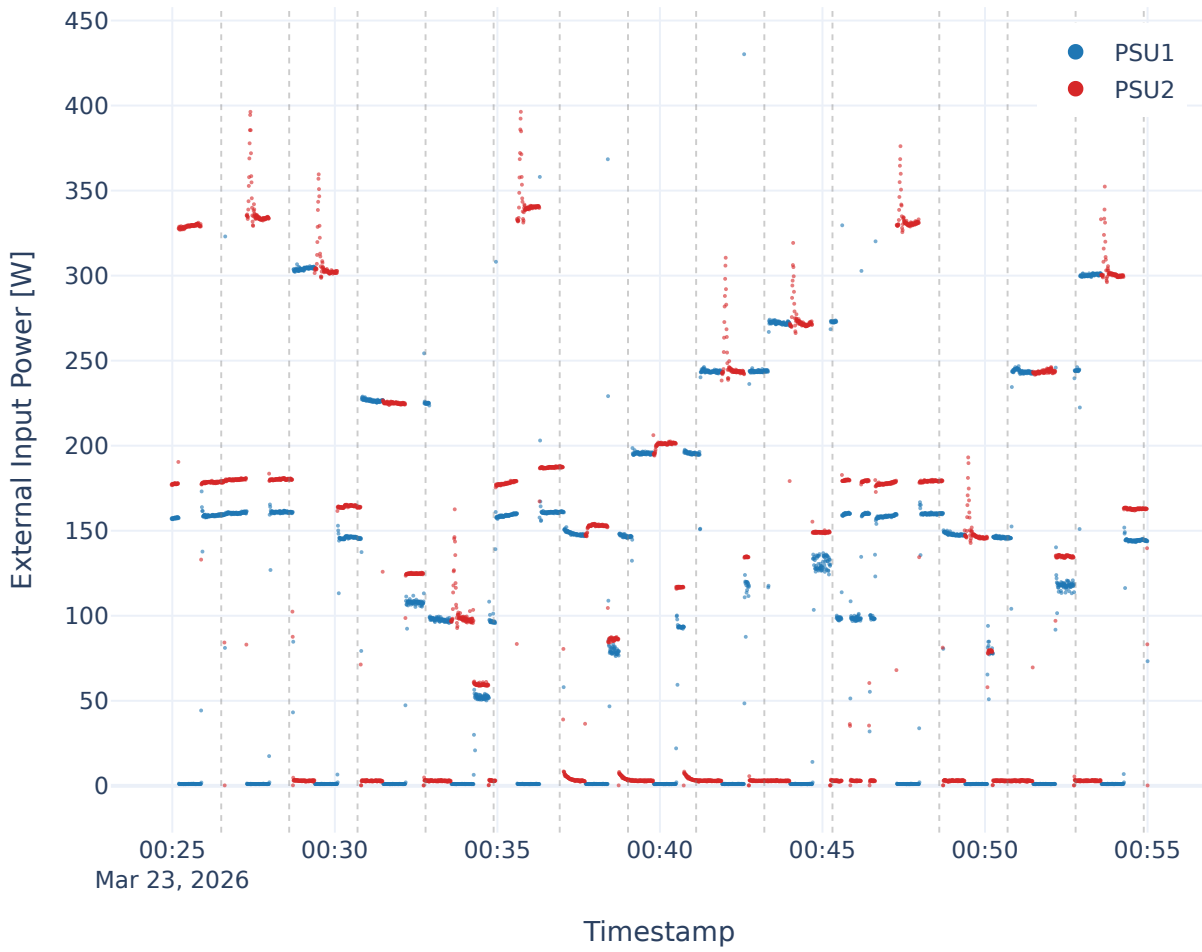


Figure 4.8: External input power measurements for the first 30 minutes of a cycled experiment. Gray dotted lines separate different tests. Significant power spikes for PSU2 sometimes occur after PSU1 is disconnected.

## Chapter 5

# Outlook and Summary

### 5.1 Outlook

Our analysis of PSU efficiency was constrained by the limitations of internal measurements. The main issue is the lack of an output power sensor in our hardware, which necessitates an output power estimate based on a low-resolution output current sensor and multiple assumptions. Potential solutions are using other PSUs with output power sensors or externally measuring PSU output power, for example, by using electronic load banks as the 80 Plus<sup>®</sup> testing does. This would also allow generating loads across the entire PSU operating range, rather than being limited by the server's power draw. Alternatively, the second issue could be mitigated by drawing more power, for instance by forcing fans to operate at maximum speed or by carefully selecting PSUs with a smaller but still sufficient capacity.

We discovered that servers already use a standby mode at low loads, and both PSUs will only start sharing the load at around 30% of a single PSU's capacity. Our attempt to estimate the power savings from delaying load sharing did not yield very promising results, as the potential savings are less than 3%. We have also not looked into other potential negative side effects, such as increased hardware degradation in one PSU and potential performance loss in the server, but these savings likely do not warrant the downsides. Given the architectural similarities, it is unlikely that the savings for routers are significantly higher, and routers are less likely to already have a standby mode for PSUs than servers are. Therefore, the additional expense of modifying the hardware to support standby is difficult to justify. There is also more work needed to assess the negative side effects of standby mode, e.g., on hardware or router performance.

### 5.2 Summary

To summarize, we collected internal and external power usage measurements from multiple PSUs. The key takeaways from these experiments were as follows:

- The difference between external and internal input power measurements is up to 6 and 8 Watts. Some of it can be attributed to differences in measurement resolution.
- There are efficiency differences between 2 PSUs of the same model on the same server. It is likely that hardware degradation and potentially other factors play a role.
- Servers already use a form of PSU standby at low loads. Forcing them to keep a PSU on standby instead of sharing the load at higher loads yields small savings of less than 3%.

# Bibliography

- [1] Energy and AI 2025. [Online]. Available: <https://www.iea.org/reports/energy-and-ai>
- [2] R. Jacob, L. Röllin, J. Lim, J. Chung, M. Béhanzin, W. Wang, A. Hunziker, T. Moroianu, S. Tabaeiaghdaei, A. Perrig, and L. Vanbever, “Fantastic Joules and Where to Find Them. Modeling and Optimizing Router Energy Demand,” in *25th ACM Internet Measurement Conference (IMC 2025)*, Madison, WI, USA, October 28-31, 2025.
- [3] M. G. Béhanzin, “Reducing power conversion losses in modern PSUs,” Bachelor’s thesis, ETH Zürich, 2024.
- [4] Corsair RMx Series 850 W (2021) Review. [Online]. Available: <https://www.techpowerup.com/review/corsair-rmx-series-850-w-2021/5.html>
- [5] CLEAResult. 80 PLUS Power Supply Certification Program. [Online]. Available: <https://www.clearesult.com/80plus/>
- [6] Intel Corporation, *Intel® Server Boards and Server Platforms Server Management Guide*, Intel Corporation, 2012.
- [7] D. Laurie, *ipmitool - utility for controlling IPMI-enabled devices*. [Online]. Available: <https://manpages.ubuntu.com/manpages/bionic/man1/ipmitool.1.html>
- [8] C. King, *stress-ng - a tool to load and stress a computer system*. [Online]. Available: <https://manpages.ubuntu.com/manpages/noble/man1/stress-ng.1.html>
- [9] J. Chung, “Automated power measurement for network devices: Collecting data reliably made simple?” Bachelor’s thesis, ETH Zürich, 2024.

# Appendix A

## My Appendix

### A.1 stress-ng commands to generate loads

Full command sets, used for RQ1 and RQ2 on snowball (48 CPU cores) and delirium (8 CPU cores) respectively

```
1 STRESS_CMDS = [  
2     [], # no load  
3     ["stress-ng", "--cpu", "1", "--cpu-method", "matrixprod"],  
4     ["stress-ng", "--cpu", "2", "--cpu-method", "matrixprod"],  
5     ["stress-ng", "--cpu", "4", "--cpu-method", "matrixprod"],  
6     ["stress-ng", "--cpu", "6", "--cpu-method", "matrixprod"],  
7     ["stress-ng", "--cpu", "8", "--cpu-method", "matrixprod"],  
8     ["stress-ng", "--cpu", "10", "--cpu-method", "matrixprod"],  
9     ["stress-ng", "--cpu", "12", "--cpu-method", "matrixprod"],  
10    ["stress-ng", "--cpu", "16", "--cpu-method", "matrixprod"],  
11    ["stress-ng", "--cpu", "18", "--cpu-method", "matrixprod"],  
12    ["stress-ng", "--cpu", "20", "--cpu-method", "matrixprod"],  
13    ["stress-ng", "--cpu", "22", "--cpu-method", "matrixprod"],  
14    ["stress-ng", "--cpu", "24", "--cpu-method", "matrixprod"],  
15    ["stress-ng", "--cpu", "28", "--cpu-method", "matrixprod"],  
16    ["stress-ng", "--cpu", "32", "--cpu-method", "matrixprod"],  
17    ["stress-ng", "--cpu", "48", "--cpu-method", "matrixprod"],  
18    ["stress-ng", "--cpu", "48", "--cpu-method", "matrixprod", "--vm", "4",  
19    "--vm-bytes", "100%", "--vm-method", "incdec"],  
]
```

Listing A.1: full set of stress-ng commands on snowball

```
1 STRESS_CMDS = [  
2     [], # no load  
3     ["stress-ng", "--cpu", "1", "--cpu-method", "matrixprod"],  
4     ["stress-ng", "--cpu", "2", "--cpu-method", "matrixprod"],  
5     ["stress-ng", "--cpu", "3", "--cpu-method", "matrixprod"],  
6     ["stress-ng", "--cpu", "4", "--cpu-method", "matrixprod"],  
7     ["stress-ng", "--cpu", "5", "--cpu-method", "matrixprod"],  
8     ["stress-ng", "--cpu", "6", "--cpu-method", "matrixprod"],  
9     ["stress-ng", "--cpu", "7", "--cpu-method", "matrixprod"],  
10    ["stress-ng", "--cpu", "8", "--cpu-method", "matrixprod"],  
]
```

```

11     ["stress-ng", "--cpu", "8", "--cpu-method", "matrixprod", "--vm", "4",
12     "--vm-bytes", "100%", "--vm-method", "incdec"],
]
```

Listing A.2: full set of stress-ng commands on delirium

Subset of commands (used for RQ3 on snowball)

```

1 STRESS_CMDS_SUBSET = [
2     [], # no load
3     ["stress-ng", "--cpu", "1", "--cpu-method", "matrixprod"],
4     ["stress-ng", "--cpu", "4", "--cpu-method", "matrixprod"],
5     ["stress-ng", "--cpu", "8", "--cpu-method", "matrixprod"],
6     ["stress-ng", "--cpu", "12", "--cpu-method", "matrixprod"],
7     ["stress-ng", "--cpu", "16", "--cpu-method", "matrixprod"],
8     ["stress-ng", "--cpu", "20", "--cpu-method", "matrixprod"],
9     ["stress-ng", "--cpu", "24", "--cpu-method", "matrixprod"],
10    ["stress-ng", "--cpu", "32", "--cpu-method", "matrixprod"],
11    ["stress-ng", "--cpu", "48", "--cpu-method", "matrixprod"],
12 ]
```

Listing A.3: subset stress-ng commands for RQ3 on snowball

To each of these commands, the needed prefix for ssh is prepended and a timeout is appended.

```

1 SSH_PREFIX = ["ssh", f"{args.ssh_user}@{args.ssh_host}"]
2
3 def build_remote_cmd(cmd: list[str], total_secs: int) -> list[str]:
4     if not cmd:
5         return []
6     return SSH_PREFIX + cmd + ["--timeout", f"{total_secs}s"]
```

Listing A.4: function that builds full command that runs on target

## A.2 code snippets: get\_measurements.py

Contains important code snippets from the script that generates loads and obtains internal measurements. Some error handling, logging, comments and other less relevant code has been removed for clarity and brevity.

```

1 def run_ipmitool(*args) -> str:
2     """ Run 'sudo -E ipmitool -I lanplus [args]' on the configured BMC
3     where args are any additional appended words.
4     Returns combined stdout/stderr output or an empty string if something
5     fails with errors logged. """
6     ipmi_pass = os.environ.get("IPMI_PASS")
7     cmd = ["sudo", "-E", "ipmitool", "-I", "lanplus",
8           "-H", BMC_HOST, "-U", IPMI_USER, "-P", ipmi_pass] + list(args)
9     try:
10        return subprocess.check_output(
11            cmd, universal_newlines=True, stderr=subprocess.STDOUT
12        )
13    except subprocess.CalledProcessError as e:
14        logger.warning("ipmitool failed (exit %s). Output:\n%s",
15                       e.returncode, e.output)
```

```
14     return e.output or ""
```

Listing A.5: function to run ipmitool over LAN

```
1 def take_ipmi_measurements(writer, cmd_string: str, duration: int,
2   num_measurements: int, id_gen) -> None:
3     """ Write 'num_measurements' rows to *writer*, spacing the
4       measurements evenly over 'duration' seconds.
5       'id_gen' is a global counter using 'itertools.count()'. """
6     interval = max(1, duration // num_measurements)
7     start_mon = time.monotonic()
8
9     # take measurements periodically
10    for i in range(num_measurements):
11        measurement_id = next(id_gen)
12        measurement_ts = datetime.now(timezone.utc).isoformat()
13
14        out = run_ipmitool("sdr", "-c")
15        # write to csv file
16        for line in out.strip().splitlines():
17            fields = [x.strip() for x in line.split(",")]
18            writer.writerow([measurement_id, measurement_ts, cmd_string, *
19                            fields])
20
21        if i != num_measurements - 1:
22            next_t = start_mon + (i + 1) * interval
23            time.sleep(max(0, next_t - time.monotonic()))
```

Listing A.6: function to periodically take measurements

```
1 def start_stress_process(remote_cmd: list[str], log_file=None):
2     """Return a subprocess.Popen running the given 'remote_cmd'. If '
3       remote_cmd' is an empty list, 'None' is returned (no process is
4       started). """
5     if not remote_cmd:
6         return None
7     try:
8         if log_file is not None:
9             proc = subprocess.Popen(remote_cmd, stdout=log_file, stderr=
10                log_file)
11        else:
12            proc = subprocess.Popen(remote_cmd)
13        return proc
14    except Exception as e:
15        logger.warning("Failed to start stress-ng '%s': %s", " ".join(
16            remote_cmd), e)
17    return None
```

Listing A.7: function to start stressor process remotely

```
1 def get_single_cmd_measurement(writer, cmd, test_duration,
2   num_measurements, warmup_time, id_gen, log_file = None):
3     """Run a single test-phase: start a stress-ng command (unless no load)
4       , optionally wait for 'warmup_time',
```

```

3     then take 'num_measurements' sensor samples evenly spaced out over '
        test_duration'."""
4     total_secs = warmup_time + test_duration + 5
5     remote_cmd = build_remote_cmd(cmd, total_secs)
6     proc = start_stress_process(remote_cmd, log_file)
7
8     # Wait for warmup time, then periodically get measurements
9     try:
10        if warmup_time > 0:
11            time.sleep(warmup_time)
12            take_ipmi_measurements(writer, cmd_string, test_duration,
13                                   num_measurements, id_gen)
14
15        # wait for stress-ng process to exit if needed
16    finally:
17        if proc:
18            proc.wait()

```

Listing A.8: function to do a single (non-cycled) test

```

1 def toggle_plug(state: bool):
2     """Sets the myStrom plug state over ssh. If state is True, will turn
3         it ON, otherwise it will turn it OFF. Returns True if and only if
4         toggling succeeded."""
5     state_value = int(state)
6     remote_cmd = SSH_PREFIX_PI + ["curl", "-s", "-S",
7                                   f"http://{PLUG_IP}/relay?state={state_value}"]
8     ]
9     try:
10        subprocess.check_call(remote_cmd, timeout=5)
11        return True
12    except Exception as e:
13        logger.warning("Failed to toggle plug to %s", "ON" if state else "
14                        OFF")
15    return False

```

Listing A.9: function to toggle smart plug

```

1 def get_single_cycled_cmd_measurement(writer, cmd, phase_duration,
2   num_measurements_per_phase, warmup_time, warmup_each_phase, id_gen,
3   log_file=None):
4     """ Run a single command test in 3 phases: First phase has both PSUs
5         on, then turn one PSU off via smart switch, then both on again.
6         Warm-up time (time between stressors starting & measurements starting)
7         is applied before the first phase by default. If '
8         warmup_each_phase' is True, warmup_time is also applied before
9         phase 2 and phase 3.
10        There are 'num_measurements_per_phase' measurements evenly spaced out
11        during each phase lasting 'phase_duration'."""
12    # total secs = 3 phases + warmup time(s) + buffer
13    total_secs = phase_duration * 3 + (warmup_time * (3 if
14        warmup_each_phase else 1)) + 5
15    remote_cmd = build_remote_cmd(cmd, total_secs)
16    proc = start_stress_process(remote_cmd, log_file)

```

```

9
10 # for each phase: wait for warmup time (always before first,
11 # optionally before 2nd & 3rd), then periodically get measurements
12 try:
13     for phase in range(3):
14         if warmup_time > 0 and (warmup_each_phase or phase == 0):
15             time.sleep(warmup_time)
16             take_ipmi_measurements(writer, cmd_string, phase_duration,
17                                   num_measurements_per_phase, id_gen)
18         if phase == 0:
19             toggle_plug(False)
20         elif phase == 1:
21             toggle_plug(True)
22
23     # wait for stress-ng process to exit if needed
24 finally:
25     if proc:
26         proc.wait()

```

Listing A.10: function to do a single cycled test

```

1 def run_synchronization_spike(writer, id_gen, log_file = None):
2     """ One-time measurement spike where each phase is 10 sec long with 5
3     measurements written into writer.
4     no load -> high CPU load -> no load """
5     cmds = [
6         [],
7         ["stress-ng", "--cpu", "48", "--cpu-method", "matrixprod"],
8         []
9     ]
10    for cmd in cmds:
11        get_single_cmd_measurement(writer, cmd, test_duration=10,
12                                  num_measurements=5, warmup_time=0, id_gen=id_gen, log_file=
13                                  log_file)

```

Listing A.11: function to force an obvious power spike (for synchronization)

```

1 def run_single_run(args, writer, run_number, id_gen, log_file = None):
2     """Execute the full set of STRESS_CMDS (or STRESS_CMDS_SUBSET if psu)
3     once (shuffled if global RANDOM_SHUFFLE_CMDS is True)."""
4     cmds = list(STRESS_CMDS_SUBSET if IS_PSU_CYCLE else STRESS_CMDS)
5     if args.random_shuffle: random.shuffle(cmds)
6     for cmd in cmds:
7         warmup_time = max(0, args.warmup_time)
8         if IS_PSU_CYCLE:
9             phase_duration = max(1, args.phase_duration)
10            num_measurements_per_phase = max(1, args.
11            num_measurements_per_phase)
12            get_single_cycled_cmd_measurement(writer, cmd, phase_duration,
13            num_measurements_per_phase, warmup_time, args.
14            warmup_each_phase, id_gen, log_file)
15        else:
16            test_duration = max(1, args.test_duration)
17            num_measurements = max(1, args.num_measurements)

```

```

14         get_single_cmd_measurement(writer, cmd, test_duration,
            num_measurements, warmup_time, id_gen, log_file)

```

Listing A.12: function to complete one run

```

1  args = parse_args() # helper function that parses passed arguments
2  global SSH_PREFIX_SERVER, SSH_PREFIX_PI, DEBUG, BMC_HOST, IPMI_USER,
   IS_PSU_CYCLE, PLUG_IP
3  DEBUG = args.debug
4  BMC_HOST = args.bmc_host
5  IPMI_USER = args.ipmi_user
6  IS_PSU_CYCLE = args.psu_cycle
7  SSH_PREFIX_SERVER = ["ssh", args.ssh_alias_server]
8  SSH_PREFIX_PI = ["ssh", args.ssh_alias_pi]
9  PLUG_IP = args.plugin_ip
10
11 # not included: creating output folder, configuring logfile, saving stress
   -ng commands & FRU information in txt files and config.json file
12
13 for run in range(1, args.runs + 1):
14     id_gen = itertools.count(0)
15     csv_name = f"run{run:03d}.csv"
16     csv_path = os.path.join(experiment_folder, csv_name)
17     with open(csv_path, "w", newline="") as csvfile:
18         writer = csv.writer(csvfile)
19         headers = [
20             "id",
21             "timestamp",
22             "command",
23             "data"
24         ]
25         writer.writerow(headers)
26         run_synchronization_spike(writer, id_gen, log_fh)
27         run_single_run(args, writer, run, id_gen, log_fh)
28         run_synchronization_spike(writer, id_gen, log_fh)

```

Listing A.13: code that drives experiment

### A.3 Code computing Load and Efficiency

MAX\_CURRENT, MAX\_POWER, OUTPUT\_VOLTAGE are globals denoting the assumed maximum output current, maximum output power and output voltage (on the main rail) respectively.

```

1  output_power = output_current * (MAX_CURRENT / 100.0) * OUTPUT_VOLTAGE
2  load        = (output_power / MAX_POWER * 100)
3  efficiency   = (output_power / input_power * 100).where(input_power != 0)

```

Listing A.14: code computing load and efficiency

### A.4 Code computing Power Savings

This section includes a code that estimates the power drawn by PSU2 and a code that calculates the power savings.

```

1 OFF_THRESH = 10.0
2 def estimate_standby_power(df, target_psu):
3     target_num = str(target_psu)[-1]
4     other_num = "1" if target_num == "2" else "2"
5
6     phase1_target_col = f"phase1_psu{target_num}_input_w"
7     phase3_target_col = f"phase3_psu{target_num}_input_w"
8     phase1_other_col = f"phase1_psu{other_num}_input_w"
9     phase3_other_col = f"phase3_psu{other_num}_input_w"
10
11     # target on standby = target is passive, while other is active
12     phase1_mask = (df[phase1_target_col] <= OFF_THRESH) & (df[
13         phase1_other_col] > OFF_THRESH)
14     phase3_mask = (df[phase3_target_col] <= OFF_THRESH) & (df[
15         phase3_other_col] > OFF_THRESH)
16
17     samples = pd.concat(
18         [
19             df.loc[phase1_mask, phase1_target_col],
20             df.loc[phase3_mask, phase3_target_col],
21         ],
22         ignore_index=True,
23     ).dropna()
24
25     standby_w = samples.median()
26     return standby_w, samples
27
28 # grouped_combined_df is a dataframe with one row per cycled test where
29 # phase1 = both PSUs on at first, phase2 = PSU1 off, phase3 = both PSUs
30 # on again
31 standby_power_w, psu2_standby_samples = estimate_standby_power(
32     grouped_combined_df, "PSU2")
33 print(f"PSU2 standby power estimate: {standby_power_w:.3f} W (n={len(
34     psu2_standby_samples)})")

```

Listing A.15: code estimating the power drawn by PSU2 on standby

```

1 grouped_combined_df["avg_phase13_w"] = (
2     grouped_combined_df["phase1_total_input_w"] + grouped_combined_df["
3     phase3_total_input_w"]
4 ) / 2.0
5 grouped_combined_df["power_difference_w"] = (
6     grouped_combined_df["avg_phase13_w"] - grouped_combined_df["
7     phase2_psu2_input_w"] + standby_power_w
8 )

```

Listing A.16: code computing power savings

## A.5 Full Plot comparing Internal and Output Power

This is the full plot comparing internally measured input power and the output power estimate based on output current measurements for PSU1 on snowball.

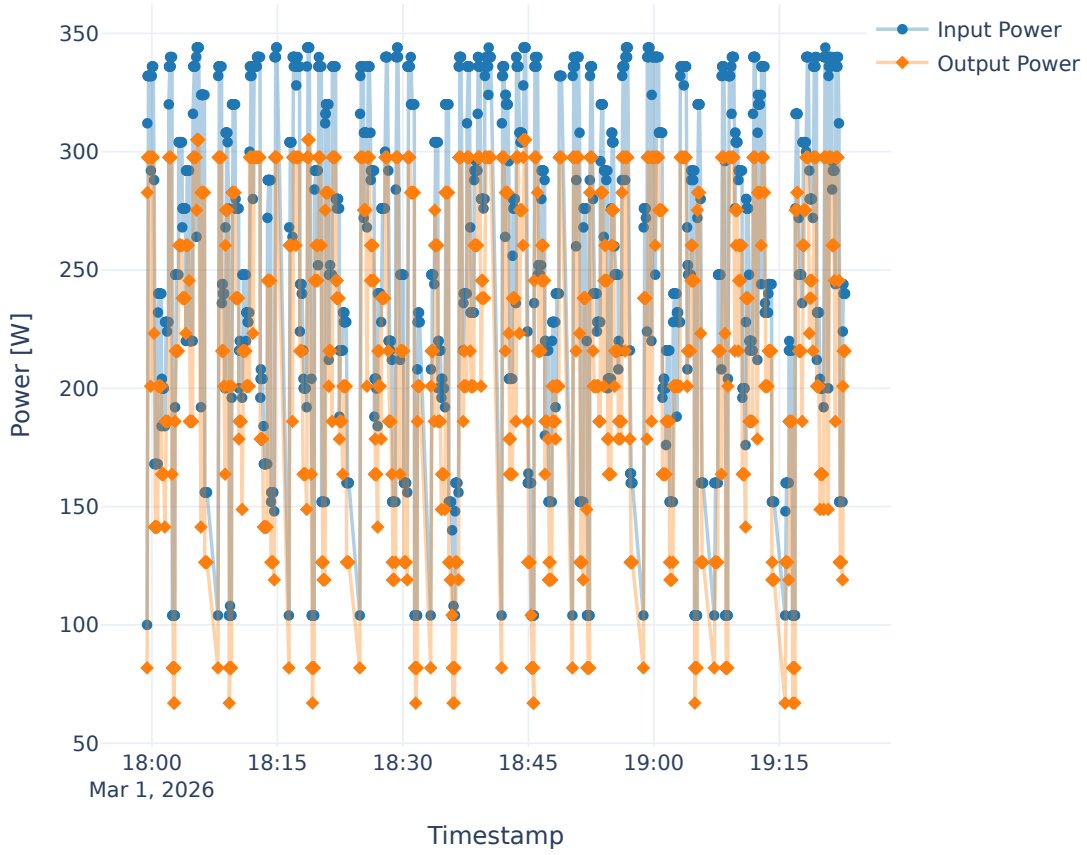


Figure A.1: Entire plot comparing input and output power for snowball PSU1.