Using GUI position to improve scheduling decisions in a desktop environment

Vladimir Hanin

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2023

Abstract

The process scheduler is an essential part of operating systems. A lot of research has been done to find the best scheduling decisions and how to take those as fast as possible. The result is versatile and robust schedulers like the one found in the Linux kernel. One of the goals of schedulers is to improve the CPU utilisation, by avoiding running processes that don't need to run, for instance because those processes are waiting for an IO request to complete. In this report, a new scenario will be investigated where the scheduler is currently unaware of running processes that don't need to run. The scenario is the one of schedulers running processes that are in a desktop environment, and hence where GUIs are present. Two ideas will be brought forward, namely that certain GUIs don't need to run when they are not in the user focus, and others don't need to run when not visible to the user. The results of a simulated model showed that schedulers currently spend a lot of time on processes that don't need to run based on those two ideas. Three different implementations for a solution were then compared, and the best one was determined to be editing the display server, which is the piece of software found in all operating systems with a desktop environment as it is responsible for managing the windows on the screen. Using that solution, a battery test was made which managed to double the battery life thanks to the CPU not doing wasted work anymore. This report shows that taking the position of GUIs into account to help the process scheduler is an issue worth solving, and that there exists solutions that can efficiently solve those scheduling mistakes.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Vladimir Hanin*)

Table of Contents

Chapter 1

Introduction: scheduling and GUIs

1.1 Setting the scene

1.1.1 Wasted work given to CPU

The task of a process scheduler in a general purpose OS is to decide when processes get to run on the CPU and for how long. It tries to maximise the utilisation of the CPU, minimise turnaround time, minimise response time, be fair to all the processes to prevent starvation, and to allow for as much progress as possible by decreasing kernel interrupts and their processing times as much as possible.

The processes that the scheduler manages must sometimes get access to external resources in order to make progress. For instance, a process might need to retrieve data from a file on disk in order to know what to do next. In those cases, one way to implement this would be to have the process request the resource to the OS, and then busy wait until the resource becomes available. During that time, if the scheduler decides to execute this process, it would lead to wasted CPU work since the process can't make any progress. This is why blocking waits were introduced, where processes send the IO request to the OS and then they are given the state of "sleeping" so that the scheduler knows not to schedule that process until the resource becomes available. In the example given above, this would mean that the process that sent the request to get the page from disk will not be scheduled until the page is found in memory. In general, whenever a process is waiting for an IO request to complete, and hence whenever it doesn't have to run, the scheduler will not execute that process thanks to the implementation of these blocking reads.

In this report, a new scenario will be brought forward where currently the scheduler erroneously decides to execute processes that don't have to run, which leads to wasted CPU time. The reason why those processes don't have to run isn't related to them busy waiting for a resource, but rather because of a higher level requirement based on the nature of the work done by the process and the environment it is running in. The analysis of this report pertains to general purpose OS's with a desktop environment, where processes can create graphical user interfaces (GUIs) that are visible on a screen for a user.

1.1.2 Types of processes

Processes running in an OS with a desktop environment can be categorised in different ways. The first distinction that can be made is to differentiate between the ones that have a GUI and the ones that do not. An example of a process of the latter type is the file compressor “tar”, which is accessed through the command line. Those processes will be henceforth called “no-GUI”.

Of the processes that do have a GUI, they can be further categorised into two groups. The distinction between the two groups relies on the nature of the work done by the processes themselves. The first group of processes called “input-driven”, are processes that *only* need to run when the user triggers an interaction with its GUI(s). For instance, a Minesweeper game only needs to execute some code when the user decides to place a flag or expose a tile. When the user isn’t interacting with the GUI of the game, the process has nothing to execute since the only work that has to be done is when the user changes the state of a tile on the board.

The second group of processes with a GUI do not only rely on the interaction of the user with their GUI(s) in order to execute some task. They must also for instance wait for an external packet to arrive, or they must continuously update their GUI(s) in a loop. Those will be called “input-output”.

This second group can be further split into two subgroups. The first subgroup are processes that *only* display real time data in a loop. For instance, a camera application that displays a preview of the webcam is displaying real time data. What is important to note is that it doesn’t accumulate information over time, it simply fetches information somewhere, maybe transforms it, then displays it, and starts again in a loop. If this process were to be stalled and then resumed some time later, no artefacts will be visible since it can simply continue fetching the real time data in a loop. Processes in this subgroup will be referred to as “RTD” (for real time data).

The second subgroup within “input-output” are application that do not only display real time data. The data they display is something that they compute over time, or they must wait for a certain condition such as a network packet to execute a task and display the result on the GUI. These processes have a visual component, but they also have a continuous piece of work that must be executed continuously. For instance, the “gnome-system-monitor” app for GNOME desktop environments, continuously displays the resource consumption over time. This process cannot be stalled and resumed, since the stall will introduce missing data in the graph during the stall. Processes within this subgroup will experience artefacts if they are interrupted, whether it is because during their pause they won’t be able to detect an external event, or because the pause will mean that they couldn’t continue executing some piece of work. Those processes will be called “CW” (for continuous work).

It is worth noting that every process corresponds to exactly one category above, though it isn’t static. For instance, a process can first be of type no-GUI and then become of process with a GUI. The point of those categories is to see what kind of optimisation can be applied to processes that are currently of that type. All the categories are visualised in figure 1.1.

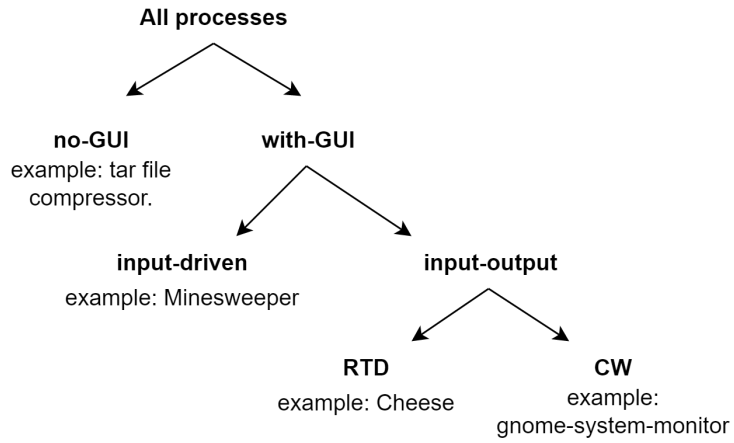

Figure 1.1: The categories that all processes in a desktop environment belong to.

1.1.3 GUI position brings value for scheduling

Based on the categories introduced above, their relationship with process scheduling will be explained below.

1.1.3.1 input-driven processes

As explained above, processes that are of type input-driven only need to execute pieces of work when the user interacts with their GUI(s). This can be either because of a mouse click or a keyboard press.

In the context of a desktop environment, there is this concept of a single GUI being "under focus". In general purpose OS's like Windows, macOS or Ubuntu, the focus is indicated with a visual aid, usually with a slightly different colour for the banner at the top of the GUI under focus. This focus was created to help the user to for instance only write in the text document that they are looking at, and not also in the browser search bar that happens to also be open. This focus means that the main mouse clicks and keyboard presses will be sent to the window under focus, and not the others.

Using the two pieces of information above, we can conclude that when an input-driven process isn't under focus, then it doesn't have to run, since the mouse or keyboard events won't be sent to that GUI. If the scheduler decides to execute that process, it would lead to wasted CPU time. This means that processes of that type could be manually paused when they are not in focus, and resumed when they are.

Though, care must be taken when deciding when to pause those processes. For instance, if a user is playing Minesweeper, clicks a tile to reveal what is underneath, and then changes the focus to another GUI before Minesweeper had the time to determine whether the tile had a mine beneath, then the game would not be able to display the result of the click of the user. Processes of this type can be described as always waiting for a mouse or keyboard event, execute some work based on the event, and then wait again for the next event. But if the work that has to be done for an event is very long, it must still be able to finish that task before it is killed by the fact that it isn't under the

focus anymore. The bottom line of this optimisation is that since input-driven processes need a mouse or keyboard event to continue executing, then without the focus that requirement can't be met, and hence those processes don't need to run anymore until that requirement is able to be satisfied again.

1.1.3.2 RTD processes

A new concept will be introduced here, namely that the value of real time information is related to whether it is viewed by an observer. If we consider the situation where the camera app is displaying a preview of the webcam, but the user isn't looking at the preview, then the information that the preview is displaying is lost. It is lost because as soon as the user looks back at the preview of the webcam, they will see the current correct preview, and hence calculating and displaying the previous previews when the user wasn't looking was wasted. Real time data implies that it is only valid in the present, and hence without an observer to act on this data or store it, then the real time data has no purpose of being displayed.

In the context of a desktop environment, the visibility can be determined in three ways. Firstly, the GUI can be minimised, in which case it isn't visible anymore. This is found in most general purpose OS's, where the window can be folded away in a taskbar for instance. Secondly, the GUI can be in another virtual desktop or workspace, in which case it isn't visible. Lastly, the GUI can be hidden because it is obstructed, meaning that there are other GUIs that completely cover it.

Using the information above, this means that processes of type RTD only need to run when their GUI is visible to the user. If not visible, those processes can be paused until they become visible again. One subtlety is that the user isn't the only potential observer. For instance, a screen recorder could be recording the GUI of the preview of the webcam, in which case the GUI can't be paused if minimised for instance, since the screen recorded would not get the data it needs. This means that the default visibility checking must sometimes be able to be overridden by another application, which will be taken into account.

1.1.3.3 CW processes

As explained above, CW processes must always run, regardless of the interaction with the user, or the visibility of their GUI. An example of a CW process is a visual simulation of planets in our solar system. The process has some visual component to display the position of every planet, but must also compute the new position of planets based on their current position using complicated equations. It must continuously work regardless of the interaction of the user, or of the visibility of its GUI.

What is interesting is that the type of work the process is doing can be separated into two purposes. The first purpose is to continuously calculate the new position of the planets based on the current ones, and the second is to create the visualisation of the planets moving around. What can be done here is to refactor the code so that instead of having one process, there are two processes, where the first continuously calculates the new planet's positions, and the second will continuously visualise the planet's positions

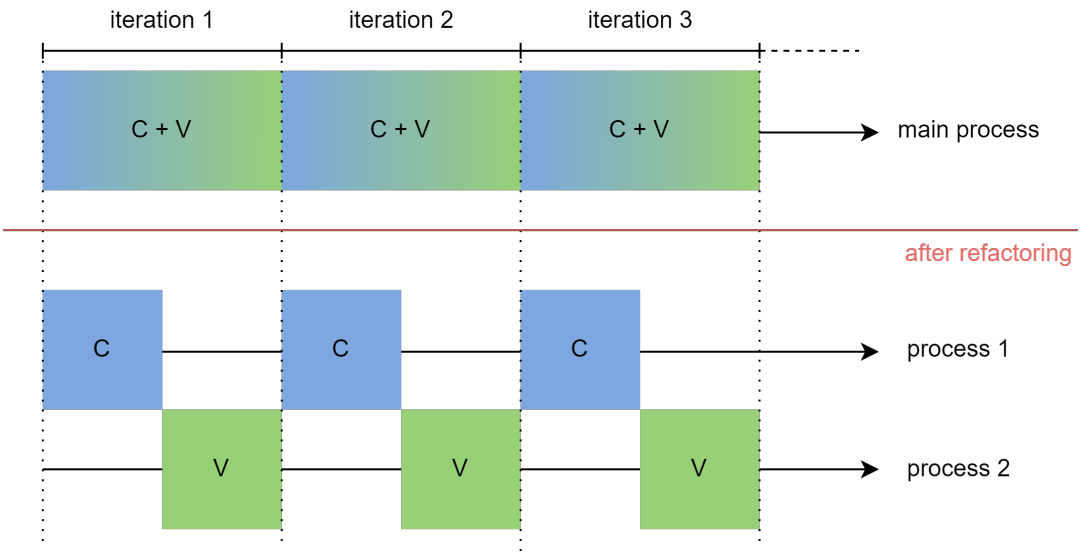

Figure 1.2: Example of a CW process which does calculation (C) and the visualisation (V) in a single process in a loop, that can be refactored into two processes which do a single purpose each. The new process only doing the visualisation can now be pause when the GUI isn't visible.

that the first process created, as shown in figure 1.2.

What is interesting is that now the second process that does the visualisation has become a RTD process, since it is continuously displaying this "real time data" that the first process provides it with. If the user isn't looking at the simulation of the planets, then the second process has no value in displaying that data, and the first process can simply continue calculating the new position of planets. Also, the first process has become a no-GUI process.

This method of refactoring CW processes into a no-GUI and RTD process can always be applied, since those processes always have a component that does some kind of continuous work, and a visualisation part. Some might have a very small visualisation part compared to the continuous work or the opposite, it depends on the process itself. The visualisation part is displaying the real time data that the continuous work is exposing, just like an model-view pattern in software development. Hence, also for CW processes the position of their GUIs can be taken into account to improve the CPU utilisation, though it does require the process to be refactored first to extract the visualisation work from the continuous one.

1.1.3.4 Bigger picture

What this analysis shows is that all categories of processes mentioned above can be optimised using the the position of their GUI. Since all processes that exist with a GUI fit into the three categories discussed above, they all have the potential to be optimised. This makes the overall idea of using GUI position for scheduling quite appealing.

The higher level idea that is being conveyed here is that all the tasks executed by a

computer with a desktop environment belong to one of the three following types: either visualising real time data in a GUI (such as a RTD process like a camera app, or within a CW process like visualising the position of the planets), input-driven with a GUI (for instance Minesweeper), or continuous work with no GUI (such as the tar file compressor, or within a CW process like calculating the position of the planets). The first type of task can be paused when its work can't be visualised by the user, the second type can be paused when the user can't interact with it, and the last type must always run.

1.1.3.5 Distinguishing between the categories

One important thing to determine is how to categorise a new unknown process. This is important since the category it belongs to will determine what optimisation could be applied to it. The best option would be that the OS can determine automatically what type the process is, based on the assembly code or the behaviour of the process while it runs.

A distinction that must be made is for instance between RTD and CW processes. It is worth noting that real time data doesn't really exist as computers take some time to first retrieve the data, perhaps transform it, and then display it. For instance, the preview of the webcam isn't perfect real time information, though for us humans it is unnoticeable. What is interesting is that increasing the delay between each update of the preview makes it become less and less real time data. In the extreme case where the preview is only updated every 10 seconds, then most users would agree that it isn't real time data anymore. Hence, the process can't be paused if not visible, since some users will want to see what was happening 10 seconds before the GUI was visible again (and not 10 seconds because the GUI was paused as it wasn't visible anymore). The issue is deciding how long a delay can be to still be considered real time data. This is purely subjective, as each user might have a different threshold. This implies that there can't be a general rule to determine whether a process is RTD or CW. A similar argument can be made about the distinction between input-driven and input-output processes.

The subjectivity in the distinction between the processes implies that the OS can't always make a correct decision. It could be told to take a conservative choice, though there might always be users that won't agree with it. Hence, the distinction between the categories must be made by the developers themselves to decide for their app, or the user to override that choice if necessary. The first option will mean that each process should be able to dynamically report what category it belongs to during its life time. This is the approach that will be taken in the implementations discussed below.

1.2 Problem description

1.2.1 Problem definition

As explained above, processes of type input-driven don't need to run if they are not in focus. What should be noted is that those processes should already exhibit this behaviour due to the way they ask for mouse or keyboard events. They can use blocking reads when waiting for the mouse or keyboard presses, which will not make them run

when they are not in the user focus. For instance, gnome-mines (Minesweeper for GNOME) manages not to run whenever not in the user focus. The issue is that there are a lot of other commonly found apps that are of type input-driven that still run when they are not in focus, such as GIMP for instance. The reason why those processes could be due to multiple reasons. The most obvious is that those processes don't use a blocking read on the mouse and keyboard events, but rather busy wait for them. Another reason could be because of the development stack that is too big. When creating big programs like GIMP or Blender, developers usually use high level libraries to draw on the screen. Those libraries do not always expose these blocking reads for the next keyboard or mouse events. The fact that the processes of type input-driven are still running when not in focus is more a bad coding practice since as shown by gnome-mines it is possible to get that optimal behaviour.

The second issue is related to the RTD processes. As explained above, those processes don't need to run when they are not visible. However, processes like Cheese (camera app) keep running even when minimised for instance. In this case it isn't bad coding practice, rather the fact that this higher level requirement of the visibility of real time data isn't taken into account by the scheduler, and hence those processes keep running even though the nature of their work and the environment they are in implies that their work is wasted.

The last issue is the fact that processes of type CW keep doing visual work when their GUI is not visible. For instance, a visual simulation of the solar system will do work related to updating the GUI even though it isn't visible. Once again, this isn't bad coding practice, rather the fact that those applications are not refactored properly and that they don't take their position into account to determine whether their work is useful.

1.2.2 Advantages of solving the problem

There are multiple advantages in solving the issue. The first major one is that the CPU can remain idle instead of doing useless work. As a direct consequence of this the battery life should improve. This advantage would be very valuable for portable devices like phones and laptops. Also, this should reduce the power generated by the CPU, which might help with certain demanding workflows where high temperatures were reached and the CPU had to throttle.

An indirect advantage is that as the CPU is doing less work, that new idle time could be given to the other processes that do useful work. This means that for instance their responsiveness can be improved. Also, if the OS has too many processes to handle that are asking for too much memory, it could easily push those paused processes to disk. Then, as soon as those become visible again, the OS can bring the process back to memory and resume it. Thanks to those optimisations the OS can now better manage its resources, which means that for instance it can run more processes concurrently.

Chapter 2

Current work and delta

The current academic work done in this area is very limited. Regarding scientific papers, none could be found that introduced this issue of processes running when they don't need to due to the position of their GUI. This paper will be the first one. As a consequence, there is also no scientific reports that tried to measure how big this issue was, or how it could be efficiently solved. However, there are some concrete solutions found in current systems that try to directly or indirectly mitigate this issue, as discussed below.

2.1 UWP

UWP is one of many ways possible to create an application with a GUI for the Windows operating systems, which is developped and maintained by Microsoft. It provides a unified way to create GUI applications for all of the versions of Windows but also the XBox and other environments, without having to create a specialised version for each.

In the earlier versions of UWP, applications were either running or not running. Due to the increasing presence of portable devices and hence of power management, Microsoft updated UWP to include two more states: "running in background" and "suspended", where an application transitions through them as shown in figure 2.1. These new states were added to reflect the position of the window. By default, if the window is not visible, then it will transition to "running in background" and finally to "suspended", in which case it will be paused until it is visible again to the user. The precise data that they gathered about the need for better power management which motivated their decision to add those states could not be found.

The approach taken by Microsoft to solve the issue was to create this heavy development environment. To create a GUI using UWP, programmers have to strictly abide to creating specific methods which become callbacks that will be called when certain events occur, for instance when the window isn't visible anymore. If they want to keep their app running while in the background they have to use a specific class and implement it using specific guidelines.

An advantage of this approach is that by default apps will be suspended when not

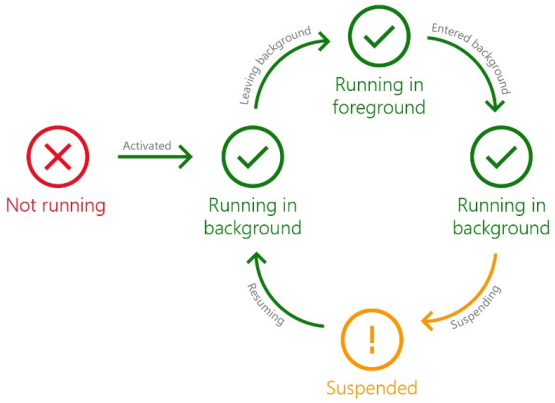

Figure 2.1: Different application states that a UWP app transitions through based on its position. The position of the GUI influences the state, and the state determines whether the process is running.[3]

visible. When a programmer wants to create an app, their application will be killed in the background by default. This means that even if a developer doesn't know about this feature of UWP, the advantages of their app not running in the background will be noticed regardless.

One thing to note is that UWP doesn't take the focus of the window into account. It doesn't differentiate when the window is running in the foreground because the user has given it the focus, or because it is simply visible on the screen but the user is busy doing something else. Hence, windows like GIMP won't be suspended if they are still visible to the user but not in focus.

Another downside with this approach is that the task of the developers is a bit more complex, as it isn't as easy as switching a variable and then the app is allowed to run in the background. The developers have to actively think about each of the jobs their application has to complete in advance and know whether it must run while the GUI not being visible. This makes developing applications in such an environment a bit more convoluted.

It is worth noting that UWP isn't the only way to create applications for Windows. There is also Windows Forms for instance. UWP was the only one found that had this feature of not running when not visible. Apps that aren't developed under UWP can't easily be optimised in terms of their visibility or focus. Also, the user can't easily override the decision made by the developers. For instance, it isn't easy for a user to override the decision of the developers to pause the app when it isn't visible.

2.2 Android

App development in android is another area where the issue was attempted to be solved. In that development environment, developers have to abide to strict rules such as implementing callbacks for certain events that their apps might be subjected to, as shown in figure 2.2.

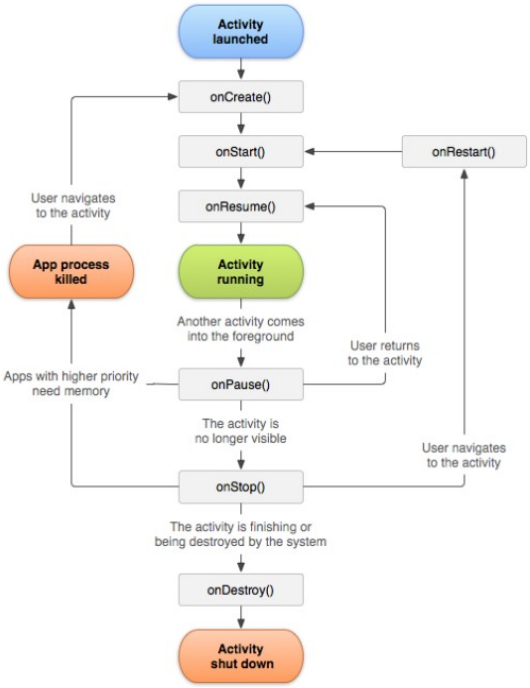

Figure 2.2: Android activity life cycle, which includes callbacks related to the position of the GUI on the screen.[1]

As we can see, developers can easily know when their application is in the foreground, or hidden. They can also know when their application is in focus. This means that developers can already adapt the work that their application is doing corresponding to the position of their GUI. For instance, a camera app can release its connection to the webcam and hence stop displaying its preview as soon as it isn't visible by editing the "onStop()" callback.

Though, this isn't enforced by anything. Applications are still allowed to run in the background by ignoring the calls to the respective callbacks. Though, this issue is mitigated by certain android versions that let the owner of the phone to decide when to restrict background usage to certain apps.

Overall, this shows that in the Android phone environment, due to the heavy model in which the developers work in, it is easy for them to create an app that behaves differently based on the visibility that the app has, and it is easy for users to correct apps that don't behave properly in the background.

2.3 Delta of this report

These results show that Android has a full solution implemented already. Windows does have UWP, but it doesn't have the detection for focus, and no solution on macOS or Linux was found. Based on this analysis, the first goal of this report will be to implement solutions in the Linux environment, since no solution was found for it and because of its open source nature. The scripts discussed in this report for the different

implementations are made available. Regarding the last implementation, the changes proposed to the display server were submitted as a pull request to the X11 display server to contribute to the open source community.

Another goal of this report is to find the best solution to solve the problem, by comparing different implementations. The first one is a naive implementation where each process that knows it can be paused will run a script to pause itself when they don't have the focus or they are not visible. The second implementation is to group all those individual scripts into one centralised one. The last implementation is to change the display server so that it can itself pause or resume windows based on their properties.

The last goal of this report is to analyse and measure how big the problem is, since no academic work was found about this topic. It will be attempted to measure as accurately as possible how much time the CPU spends on windows that shouldn't run when not in focus, or shouldn't run when not visible. This will indicate whether the issue is worth analysing further and whether the solutions proposed are worth spending more time optimising.

2.3.1 Goals of the implementations

The goals of the three implementations to solve the issue will be three fold:

1. First, the smallest overhead is sought. The amount of time the CPU that is spared thanks to the implementation should be the smallest compared to the extra work done by the CPU due to the implementation itself.
2. Second, the implementations should be as unnoticeable as possible to the user. The user shouldn't notice that this optimisation is being implemented. One thing in particular that might be introduced is that for RTD processes (not for input-driven) when it is not visible and hence paused, and when the GUI suddenly becomes visible again, the process should be revived as soon as possible. The problem that could occur here is that if the process is not revived fast enough, the user will see the GUI in the state that it was before it was paused, and hence this would introduce an error for the user since they could interpret that information to be valid. For instance gnome-system-monitor is frozen when minimised, and it reports that a certain process q is running, then right after making gnome-system-monitor visible again the process q might still appear on the GUI. In that case, the user might not be sure whether the process q was running now or if it is an artifact of when the GUI was paused. In the implementation discussed below, those artifacts will be minimised as much as possible.
3. Lastly, developers should have the easiest app development possible. The implementations proposed shouldn't make the life of developers much more complicated, otherwise developers might not decide to take the route of optimising their app.

The implementations proposed in this report will be compared based on those criteria.

Chapter 3

Problem analysis

In this chapter, the magnitude of the problem will be measured. This means measuring how much the CPU is executing processes that don't need to run, hence how much wasted work it is doing. More precisely, how often and how long input-driven processes are scheduled even though their GUI is not under focus, and RTD processes are scheduled even though their GUI isn't visible to the user. Also, the amount of visualisation done by CW processes will be estimated.

The goal will be to have the most precise and accurate value as possible. Though, it is worth noting that this task is quite complex, as shown below. Averages will have to be made in certain cases. Hence, the primary goal of this analysis is rather to determine whether the problem is a significant one, by obtaining an order of magnitude, which will indicate whether investigating it further is worthwhile.

3.1 Challenges

The final value sought is what percentage of the work that a CPU is doing is wasted. The problem is that this value depends on the following factors :

1. The position of the GUIs. If for instance only the camera app is open and visible, then nothing that the CPU is doing is wasted. However, if that GUI is minimised then the CPU is doing wasted work.
2. The types of processes and the types of their GUI. If all the processes are no-GUI, then all the work the CPU does is useful, and hence there is no waste. On the contrary, if all processes are of type input-driven and none are in focus, then there is wasted CPU time. Related to that is the process themselves. If we take gnomes-system-monitor and compare it to Cheese, both show real time data, though Cheese is taking more CPU time than gnome-system monitor. This is related to the type of work that the process behind the GUI is doing.
3. The number of open GUIs. If there is no open GUIs, there is no wasted CPU time, whereas in the situation where there are 15 GUIs open there can be wasted time spent on the CPU.

These three factors mean that there exists cases when everything the CPU is doing is wasted and everything it is doing is useful. There exists many different configurations of processes and positions, where each has a different resulting wasted percentage. Hence, there is no single correct final value that exists, since all scenarios have a different one. This implies that the average amount of wasted work that the CPU does must be found instead.

3.2 Empirical data

Perhaps the most straightforward way to measure how much time is wasted on average is to simply monitor a computer used by an average user, by measuring the percentage of wasted work done by the CPU while the user is busy working on it. The issue here is to find an average user. Each person has a different way of using applications on their computer. For instance, some keep lots of open browser tabs open, while others keep very few. Hence, finding one average user is very hard. A way to combat that would be to take lots of users, and then take the average of all of them. One way this could be done is through a university, where the computers made available to students would be monitored. The issue again is that the students might not be representative of every user. Students might use some piece of software that most users don't use, like a specialised calendar apps for instance. Due to the complexity of finding a representative sample of users in the population, this approach was set aside.

While the approach discussed here should be very accurate for those specific cases, it would be hard to know whether the results can be generalised. This is due to the different pieces of software that people use, or the different ways they interact with applications and their GUIs. For this reason, another approach was sought.

3.3 Model

Instead of using empirical data, a simulated model will be used. Each simulation will depict a specific scenario, where the percentage of wasted work done by the CPU can be calculated. The advantage with this approach is that now lots of different scenarios can be simulated, where the average can then be taken, but also specific interesting scenarios can be analysed in isolation, and general trends across all scenarios can be noticed.

3.3.1 General idea behind model

The model relies on the key observation that when there are a few processes running on a system, for long enough time intervals, the CPU will be idle most of the time. This is because the running processes are for instance executing long IO requests, or because they are deliberately pausing themselves to wait for a certain event. For instance, we can take the fake scenario where there are two processes running P1 and P2 which each take about one sixth of the total time interval, and where the remaining time the CPU is idle. This situation is depicted in figure 3.1, where the proportion of the time that the

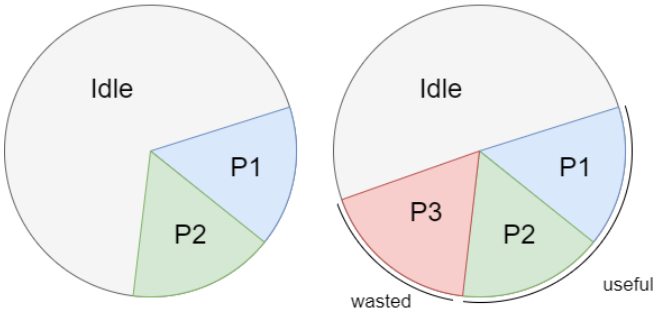

Figure 3.1: Diagram showing the individual proportion of time taken by process P1 and P2 to run on the CPU during a 10 second time interval. Right is when a third process P3 is added. In this case, the model would calculate that one third of the work is wasted.

CPU spent on each process or idle is shown within an arbitrary 10 second interval. The interesting behaviour is that if a new process P3 is launched, it will simply replace a proportion of the idle time of the CPU. The CPU utilisation of the individual processes P1 and P2 will be almost identical to the original scenario, as shown on the figure.

The model then takes this approach to simulate different processes running on a system. It will measure the proportion that each process takes individually on the CPU, and then create scenarios with different combination of processes and different positions of their GUIs, by summing their individual CPU consumption, and then calculate the resulting ratio of wasted work done by the CPU. The only limitation that the model must take into account is that there must always be some idle time left to be given to the new process, otherwise the new process will steal some of the CPU time of the other processes. In reality, the scheduler will have to make the decision also based on the priority of each process. This will be taken care of in the analysis to make sure such scenarios are detected.

In reality, the CPU consumption of a process running individually on the system will not be exactly the same as its consumption when there are a few others running. Namely, it will be slightly smaller. Though, this reduction was observed to be similar for all processes. Hence, while their total consumption will be overestimated, taking their ratio means that this decrease isn't affecting the result.

3.3.2 Equations

The equation that the model uses to simulate each scenario as described above starts from the following:

$$z = \frac{TTw}{TT} = \frac{TTw}{NG + (TTw + TTu)} \tag{3.1}$$

where z stands for the final value the model tries to calculate, which is the proportion of work that the CPU is doing that is wasted. TTw stands for the total time spent on processes with GUIs that is wasted due to their position, and TT stands for the total time that the CPU worked for (hence ignoring idle time). Based on the categories of

processes introduced in this report, TT can be decomposed into the work done for processes with a GUI and the ones without. The total time spent on processes of the latter type, which also includes kernel threads, is denoted with NG. The processes with a GUI can then be decomposed between the ones that were wasted work on the CPU TTw, and the work that was useful TTu.

The total time the CPU spent that is wasted TTw can be decomposed further based on where the wasted work originates from:

$$TTw = TTwfocus + TTwf + TTwv = TTwf + TTwv \tag{3.2}$$

where $TTwfocus$ is the total time wasted by the process under focus, $TTwf$ is the total time wasted for processes that are visible but not in focus, and $TTwv$ is the total time wasted for processes that aren't visible. This separation means that if the CPU does wasted work, it will belong to exactly one of those values. In this case, there is no time wasted when the process is under the user focus, hence $TTwfocus$ is zero.

In order to construct different scenarios, the model will need two parameters. The first one x will denote the number of open apps on the system. It will range between 0 and 20. The second parameter that the model uses is y, which represents the number of GUIs that are visible to the user. This parameter will range between 0 and x. Based on these parameters, the $TTwf$ and $TTwv$ can be rewritten as follows:

$$TTw = 0 + TTwf + TTwv = (y-1)TTwfi + (x-y)TTwvi \tag{3.3}$$

where $TTwfi$ and $TTwvi$ represent the time one process wastes individually on average while visible but not in focus and while not visible respectively. The same reasoning can be done with TTu, giving :

$$TTu = TTufocusi + (y-1)TTufi + (x-y)TTuvi \tag{3.4}$$

where $TTufocusi$ is the average amount of useful time given for one process running individually while under the focus of the user, $TTufi$ is the average amount of useful time spent for one process that is visible but not in focus, and $TTuvi$ is the average amount of useful time spent on a process that is not visible. The resulting full equation of the model for a given x and y is then:

$$z = \frac{(y-1)TTwfi + (x-y)TTwvi}{NG + ((y-1)TTwfi + (x-y)TTwvi) + (TTufocusi + (y-1)TTufi + (x-y)TTuvi)} \tag{3.5}$$

The model will then range over different values of x and y, and then use the equation above to calculate the percentage of wasted CPU time. The task is now to find the six following unknowns: $TTwfi$, $TTwvi$, $TTufocusi$, $TTufi$, $TTuvi$ and NG.

3.3.2.1 What processes

In order to find the values of the first five unknowns above, an average will be taken from a list of different processes. Since the goal is to represent the most common scenarios, the programs that are the most often used in Linux distros will be used as reference. This way, the results of the model will try to represent the most common workflow that a user might have on their computer. The programs decided to be part of the analysis were chosen because they were very common for their respective task. For instance, the most common free 3D rendering software available on Linux is Blender.

The resulting list of programs used for the model is the following: Firefox, Chrome, Thunderbird, LibreOffice Draw, LibreOffice Impress, LibreOffice Write, LibreOffice Calc, VLC Media Player, Shotcut, GIMP, Visual Studio Code, Ardour, Discord, Intellij IDEA, Pycharm, gnome-system-monitor, Cheese, Blender and Spotify. For more information on each program such as their version, see appendix A.

3.3.2.2 Method to obtain first five unknown values

For calculating the first five unknown values above, every program was measured in that scenario to see how much they contributed to that unknown, and then an average was calculated from all of the programs. For instance, to measure the average amount of CPU time that one process wastes when not visible $TTwvi$, the GUI of the program was minimised so that it wasn't visible anymore, and then the amount of CPU time they wasted was recorded. Some programs didn't waste time when not visible, like Discord, so 0ms was recorded for those. The programs were recorded when running individually on a Ubuntu 20.04.5 LTS. The average was then calculated for all of the programs mentioned above.

In order to measure how much CPU time they received, it was decided to record during a window of 10 seconds. During those 10 seconds, the number of milliseconds that the program was running on the CPU was recorded. This was done by recording the number of clock ticks the CPU spent on the process in user mode and kernel mode (by accessing the `/proc/<pid>/stat` file). If a GUI had multiple processes behind it, all of them were recorded at the same time.

Since programs can sometimes have quite erratic behaviours, 12 samples of 10 seconds were recorded, which in total means 120 seconds of wall-clock time was recorded. Then, those 12 samples could be visualised to see whether the program was indeed using a constant amount of time on the CPU over time, and wasn't behaving erratically. Sometimes, it was found that a program would increase or decrease its CPU usage over time, which meant that the average behaviour was hard to determine. In those cases, 12 new samples were measured and the analysis was restarted until a normal behaviour was observed. If the program showed a relatively constant CPU time, the average of those 12 samples was recorded as final value.

The script used to record the usage of a program can be found in appendix B, along with an example of how Cheese was analysed.

3.3.2.3 How NG was measured

The amount of time spent by processes without a GUI NG also had to be measured. The first thing to note is that the amount of work that those processes produce is heavily influenced by the number of processes that are running on the system. A lot of those processes are utilities that other processes can use, and hence the more processes are running on the system, the more those utilities will be used. Hence, taking one single value and applying it to all values of x isn't accurate.

Instead, the value will be approximated for the different x. This was done using linear interpolation between the value found for $x = 0$ and $x = 20$, which were 297.5ms and 1255.8ms respectively. Note that this method will slightly overestimate the work that must necessarily be done. This work is considered to be "useful", but since there are applications that are running that don't need to run, their access to those utilities isn't necessary and hence is wasted. This means that the final percentage of work wasted will be slightly underestimated.

3.4 Results

The values found for the first five unknown values are: $TTwfi = 238.5ms$, $TTwvi = 341.1ms$, $TTufocusi = 3199.8ms$, $TTufi = 361.2ms$ and $TTuvi = 282.4ms$. Using those values, for all x and y, the result of the model is shown in figure 3.2.

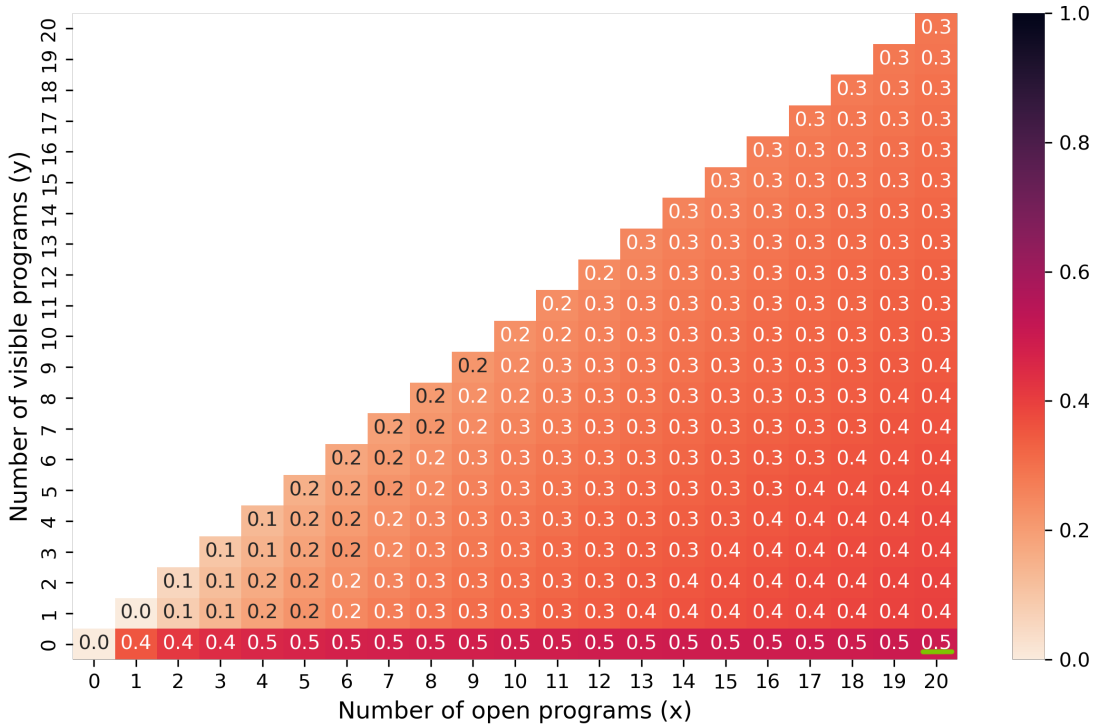

Figure 3.2: Proportion of time that the CPU spends on running input-driven processes that are not in the user focus, or RTD processes that are not visible.

Overall, what can be noted is that for almost all combinations of x and y, the proportion

of work done that is wasted is very high. The only exception is when there is very vew programs open, and the user can see most of them. In all other scenarios, the proportion of wasted work is greater than about 20%, which is quite substantial. What should be noted though is that on average, a user is only seeing a few GUIs at a time from all the ones that are open, since on average users only have one screen and only work on a couple of programs at a given point in time, and hence the last few bottom rows are the most important ones.

The conclusion from this graph is that in general, the proportion of wasted work is quite big. In the most extreme scenario (where 20 programs are open but none are visible), about half of what the CPU is doing is wasted. This would imply that in that case, the battery consumption could be improved by as much as 50%. A battery test will be done later to verify this result.

3.5 Refactoring

Here, the model was modified so that it also took into account the wasted work done by processes that have a GUI of type CW. For instance, when gnome-system-monitor is displaying the resources over time, when it is not visible it does some useful work of gathering the CPU consumption, but also the wasted work of visualising that result at each iteration. The details of how this was estimated can be found in appendix C. The model was modified accordingly and the results are shown in figure 3.3.

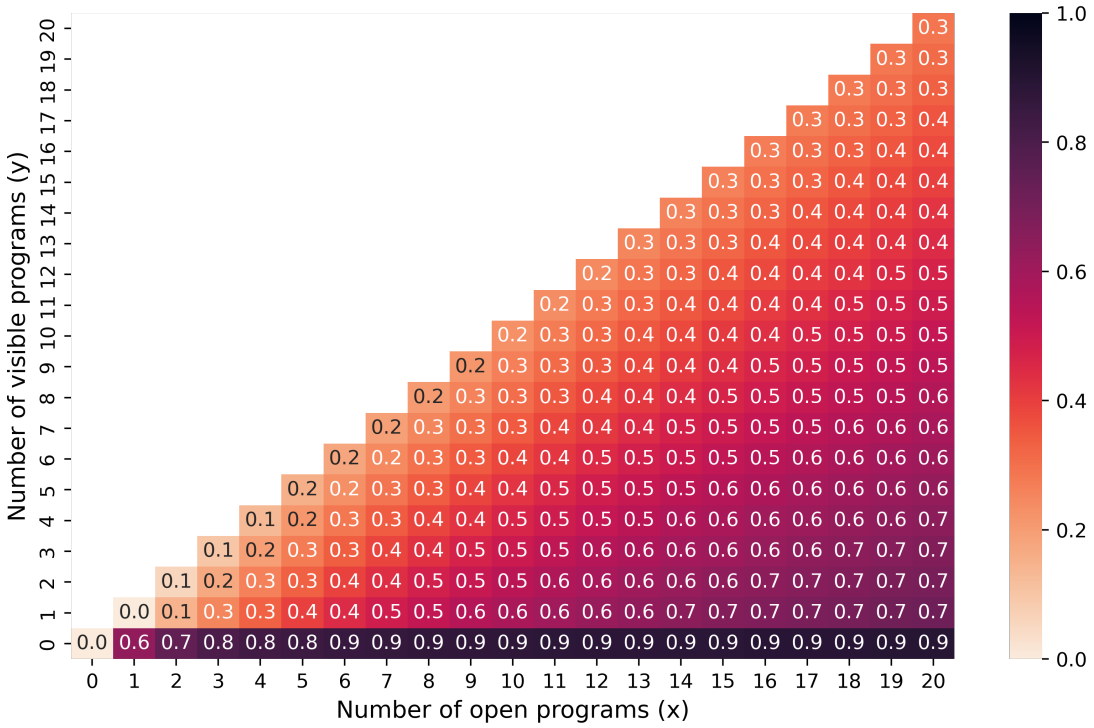

Figure 3.3: Proportion of time that the CPU spends on running input-driven processes that are not in the user focus, or RTD processes that are not visible, or on the visual component of CW processes when those are not visible.

As we can see, the proportion of wasted work is even more substantial for all x and y. The only exception is when there is only a few programs open and all of them are visible. The most extreme value is when there are 20 programs open and none are visible, where the proportion of wasted work is as high as 90%. This graph shows that the true amount of wasted work that is happening on average on our computers is very significant. This implies that refactoring CW programs so that they are split between visual work and continuous work would be very advantageous.

3.6 Discussion

Overall, throughout the creation of this model, a lot of assumptions and averages were made due to missing data on certain aspects of the problem. Whenever such decisions had to be made, they were always based on partial evidence or educated guess from experience. The final decision was made to give more weight to the cases where the problem would be slightly underestimated, compared to overestimated. As the results of this analysis show that the problem is already quite significant while being underestimated, then in practice it is expected that it should be even greater, confirming the significance of the problem.

A first factor that could have influenced the validity of the model is the choice of programs to find those first 5 unknown values. While the most common and famous apps were chosen, they might not properly represent the true average scenario found in practice. Also, their version might heavily influence their CPU consumption, if for instance one has a bug where it was stuck in an infinite loop. The measurement of the unknown NG is also very much influenced by the Linux distro, since each distro chooses which utilities are used for their environment. Additionally, the values recorded for each program could heavily depend on external factors. For instance, if the laptop the programs were running on lost internet connection, then the behaviour of the programs could have been influenced by that. Though, in order to reduce the change in external factors between each recording session, all the values were recorded in one sitting. All these factors mean that the results shown for this model should only be considered to be valid for those versions of those programs and under that version of Ubuntu. Though, since the scenario depicted by the model is quite common, a similar situation should be found with other processes and other Linux distros. More testing must be done to verify that.

It is also worth noting that the experiments were done on a 8 core computer. This means that the measurements of each program were influenced by this. However, as explained above, since the ratio of their CPU consumption is calculated, the final value shouldn't be influenced by the number of cores on the computer. However, doing the same experiment on one CPU and adapting the model for it would be interesting to verify the results found here.

Chapter 4

Implementation 1: user space individualism

In this first implementation to solve the issue, each application that knows it can be paused when not in focus or when not visible will each implement code to make themselves sleep. Hence, the developers of each application will be responsible for their own program. For instance, the developers of Cheese know that their program doesn't have to run when their GUI is not visible, hence they will program themselves the code required for their processes to sleep when it is not visible on the screen. In this implementation, a new process is created alongside the main ones which will be called the "agent", that will monitor the position of the GUI(s) and pause or resume the processes of the program when necessary, as shown in figure 4.1.

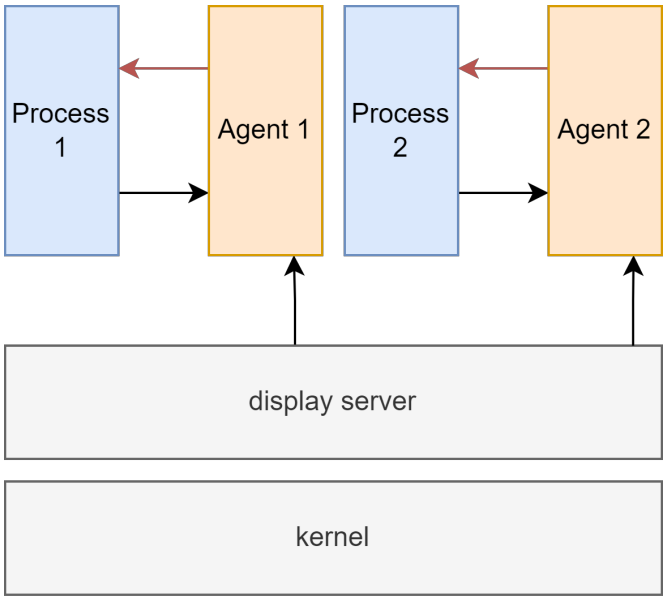

Figure 4.1: Conceptual design of this implementation where each process receives its own agent. The black arrows are processes reporting their category to the agent and the agent asking for window positions for instance, and the red are the signals.

4.1 Design

4.1.1 Input-driven processes

The solution for input-driven processes is to take advantage of events that are sent by the display server. One of those events is a *FocusChange* event, which tell the window whether it has received or was taken out of focus. These events are in the lowest level of the software stack, hence they are the most efficient way of getting that piece of information. This event means that it is quite easy for a input-driven process to know when it can be paused, since the only thing it has to do is to subscribe to the *FocusChange* event when it creates its GUI at launch. When the agent receives the event, it then simply has to send a SIGTSTP or SIGCONT signal to the main processes. Note here that the reason why a SIGTSTP signal is sent and not a SIGSTOP is to allow for the main threads to finish the work they were potentially finishing due to the previous keyboard or mouse event (solving the issue raised in the introduction).

The script created to test this implementation can be found in appendix D. It was written in C to have the highest speed, and to use Xlib, which is the most efficient way of communicating with the display server X11 that is used in Ubuntu. The script also currently creates a window and then waits for the events from the display server, though in practice the creation of the window would be done in a separate process, so that the agent is isolated in another process and hence can send signals to the main processes without being paused itself. The only thing not implemented in the script yet is the communication between the main processes and the agent, since the type of GUI can be dynamic. Though, this can easily be implemented with a shared variable between the processes, by having them share a page in memory. The overhead of this extra feature should be minimal as it would only require to read or write a single byte from memory.

4.1.2 RTD processes

The solution for RTD processes is a bit more substantial, as the information they have to find is more complex. Those processes need to determine whether they are visible on the screen. The issue is that there is no display server event that can notify them whether they are visible to the screen or not. They must verify every possible scenario manually.

The first reason why the GUI isn't visible could be because it is in another workspace. The issue is that there is no display server event that can notify when the window is in another workspace. In order to get that information, a constant polling mechanism must be used to check the current active workspace. This will significantly increase the overhead of this solution, since the agent must busy wait in the background.

The second scenario is when the GUI is minimised. This can easily be determined thanks to a window property change event. Though, since a busy wait approach must be used to get the current active workspace, the state of the window will be fetched alongside the workspace at each iteration.

The last scenario is when the window is obstructed as it is hidden behind other GUIs. Again, there is no display server event to know this piece of information directly. Instead,

the first step is to know what other GUIs are open on the system. Then, only the ones that are on the current active workspace and that are not minimised should be kept. Lastly, the position of each GUI should be checked to see whether they cover the initial GUI of interest. It is only after doing this whole analysis that the agent knows whether the GUI in question is visible to the user, and hence whether it can be paused or not.

The script created to test this implementation can be found in appendix E. Again, the code was written in C in order to benefit from the highest speed, and for the use of the Xlib library. Also, the communication wasn't yet implemented between the main process and the agent. For the same reasons as above, this shouldn't influence the results presented below.

A limitation of the current approach is that the calculations done to detect when the GUI is obstructed by others is a bit simplified. The current approach is to check whether there is one other GUI that completely covers the GUI the agent has to work on. However, there are more complex scenarios where perhaps 2 GUIs together manage to completely cover the GUI of interest, but that individually they only cover some part of it. In general, the current approach means that in rare cases, the RTD processes will not be paused event though they should. Another approach was implemented to take those partially overlapping GUIs into account, though that analysis introduced a lot of edge cases where the GUI of interest was paused even though it shouldn't. This is not acceptable as it contradicts one of the goals of the implementations mentioned in the second chapter, which is that the user should not notice the optimisation that is implemented. Also, the analysis became much more complex and hence took much more time to finish, which heavily increased the overhead and worsened the responsiveness of the implementation. This is why the simple solution of checking whether another GUI fully covers the GUI in question was kept.

Another limitation of this approach is that determining whether a GUI is visible to the user was slightly simplified. For instance, in Ubuntu, a window preview can be found when clicking on the program that sits on the tray. This preview means that the GUI is visible to the user, and hence a RTD process must be resumed when its preview is visible. The approach taken in this script is quite generalised to any Linux distribution, since they nearly all have this concept of workspaces and of minimising a window. The script would have to be tailored to each Linux distro if deployed in a real program.

It is worth noting that the script is $O(n)$ where n is the number of GUIs. This asymptotic analysis gives an indication of how much time each iteration takes, which will be used for the responsiveness analysis mentioned below.

4.2 Overhead

4.2.1 input-driven processes

The major advantage with this implementation is that the agent is not busy waiting for the event of the display server. It has the ability to do a blocking read to wait for the next event, and hence when the GUI is not changing focus this agent isn't being executed at

all. In the model, the average CPU consumption of this script would be zero since the model doesn't take into account how often the user changes windows.

Though, the script does need to do a bit of work each time it receives a *FocusChange* event. If those events where sent frequently enough, this could accumulate and make the agent spend a considerable amount of time on the CPU. The time that the agent takes might then become greater than what the agent manages to save for the input-driven process when it is not in the user focus. From the model, it was found that on average an input-driven process wastes 238.5ms on the CPU every 10 seconds. After analysing the script, it was found that on average it took about 1.2µs to receive the event and send the corresponding signal. This implies that for the agent to consume as much CPU as an input-driven process, about $238500/1.2 = 198750$ focus changes must be performed on that GUI during a 10 second interval. Since that value can't be obtained in practice, this implies that on average the agent will always use considerably less CPU than the input-driven process regardless of the behaviour of the user.

4.2.2 RTD processes

As explained above, the script for RTD processes must busy wait in the background to constantly check whether the GUI in question is visible to the user. In order to measure this overhead, the same technique was used as for recording the processes for the model, namely the script was measured alone on the system for 10 seconds, where 12 samples were measured, and then the average was taken. The result showed that on average, the agent takes 5416.6ms of CPU time every 10 seconds. This is considerably more than what a RTD process consumes on average, which is about 889.6ms. In this case, this implementation of the agent would be disadvantageous since the gain in killing the RTD processes is smaller than the overhead of the implementation.

4.2.3 Overhead in model

The model will now be used to see whether this implementation manages to reduce the amount of work that the CPU is doing. In this scenario, all the processes of type input-driven and RTD will have an accompanying agent that will pause them if they are not in the focus or not visible. The final value that the model computes is the ratio between overhead of the agents with the overhead of the original wasted work done by the CPU. If the ratio is greater than one, it means that the overhead of the agents is greater than the original amount of wasted work done by the CPU. The result is shown in figure 4.2.

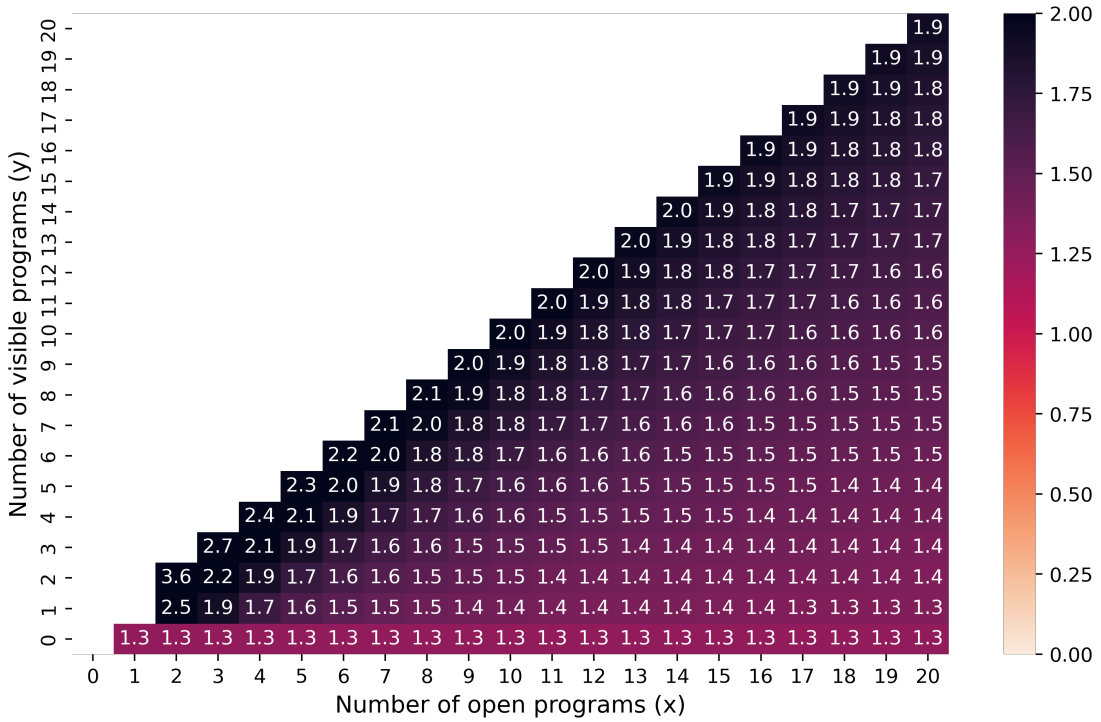

Figure 4.2: Ratio of overhead of all the agents compared to the original work done by the CPU.

As we can see from the graph, the ratio is always bigger than 1, meaning that the work done by the agents is always greater than the original wasted work done by the CPU. Hence, it can be concluded that if all the programs were to implement this solution where they individually run an agent in the background to check their visibility or focus, the CPU would actually be doing more work than originally due to the overhead of the agents being too high.

4.3 Responsiveness

In the implementation of the agent for RTD processes, it was decided to have the best responsiveness. Responsiveness here means that the the agent would try to detect that the process should be paused or resumed as quickly as possible. A bad responsiveness means that the agent takes a lot of time to resume a process that should be revived, and hence the user might notice an artefact of the previous time the GUI was paused.

One thing that could improve the overhead would be to have a dynamic time between each iteration of the loop. When the GUI of interest is not visible a very short time between iterations is used, and when the GUI is visible a much longer time between each iteration is used. This means that when the process is paused and should be resumed, the agent will detect this as fast as it can, though when the process is running and it should be paused then the agent might take a bit more time to detect that. This would simply mean that sometimes RTD processes won't be killed when they could, which is acceptable. This will improve the overhead, while keeping the same desired

responsiveness.

A new agent was made where the time between each iteration of the loop was 10 seconds when the GUI was visible, and 0 seconds when the GUI is not visible. Its overhead was again measured using the technique mentioned above. The result of the model with this new version is shown in figure 4.3

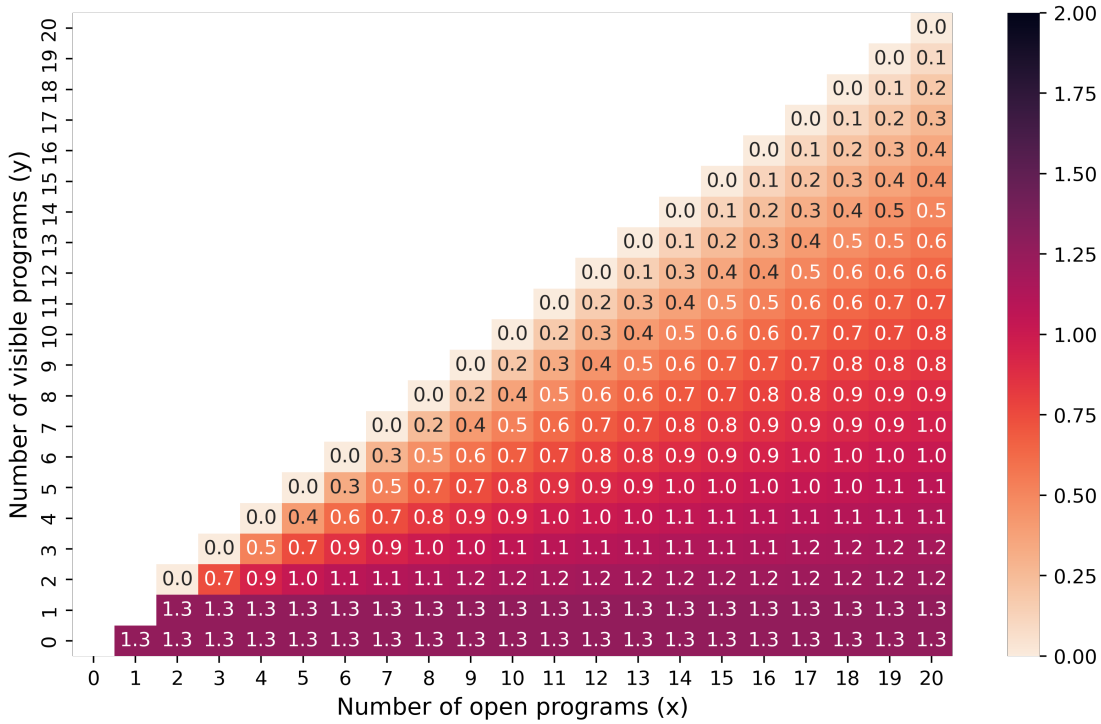

Figure 4.3: Ratio of overhead of all the the agents compared to the original work done by the CPU. Here, the agents have a dynamic time between iterations.

As we can see, there are a lot of situations where the ratio is lower than 1, meaning that the overhead is smaller than the original wasted work. Though, it is worth noting that the more common cases is when y is small. When y is small, the ratio is almost always more than one. The only small exception is when there are few programs open and more than three GUIs are visible.

Another solution to decrease the overhead even further would be to increase the time between each iterations even if the GUI is hidden. While this will hurt the responsiveness, it is worth noting that perhaps not all the apps need to have such a high responsiveness. Some developers like the ones of Cheese, might decide that for their application the fact that their GUI isn't responsive for a few extra milliseconds is acceptable for their product. This means that only their agent has to be modified so it doesn't need to be so active in the background. The exact time between each iteration can be decided by the developers of each program, or perhaps also by the users.

A new version of the agent was made, and this time the time between each iteration was 0.1 seconds when the GUI wasn't visible, and 10 seconds when the GUI was visible. 0.1 seconds should still be very close to the side of not being noticeable to the user,

which should show how much the overhead could already be reduced even for a very small decrease in responsiveness. The result of the model with this new agent is shown in figure 4.4.

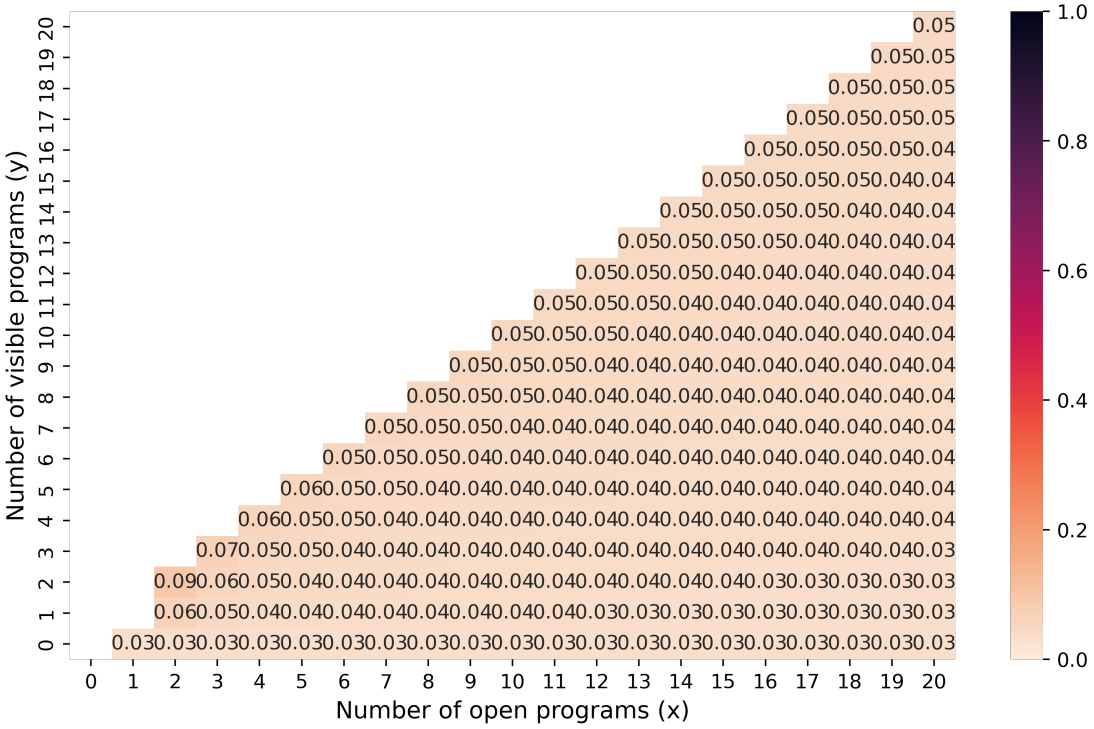

Figure 4.4: Ratio of overhead of all the the agents compared to the original work done by the CPU. Here, the agents have a dynamic time between iterations and the responsiveness is slightly worse.

As we can see, most ratios are very close to zero, meaning that the overhead of the agents is much smaller than the original wasted work. It is worth noting that perhaps not all the apps can accept this worse responsiveness, and hence here these results might overestimate the gains of this version of the implementation. Though, this graph in general shows that this implementation is viable to tackle the problem.

4.4 Discussion of this implementation

As discussed above, it was shown that for agents of input-driven processes, their overhead is insignificant and hence optimal, and for agents of RTD processes their overhead for optimal responsiveness is too high. Though, it was shown that a slightly worse responsiveness can quickly lead to a reduced overhead.

The conceptual disadvantage with this implementation is the fact that the calculation of the visibility of the GUIs is done in multiple agents. If there are multiple agents for RTD processes, they will compute almost the identical information, without sharing any results. This will be addressed in the second implementation. The other side of the coin is that each program can have its personalised agent. For instance, an app that

needs a very high responsiveness can have a small time between each iteration, while an app that doesn't need such a high responsiveness can have a very long time between each iteration. This can lead to an overall personalised situation where each app gets the behaviour that it requires. Though, it does come with a slight overhead for agents that need a high responsiveness, which might reduce the potential gains of the solution.

While this implementation might seem like a big burden on the developers, it can heavily be alleviated by turning those scripts into libraries, that they can simply call with one method. They would only have to decide which method they want to call based on the category of their process. However, this means that the developers of the libraries now have to do all the difficult work of verifying that the visibility checking they do is correct for each desktop environment that exists. For instance, Ubuntu provides the preview of windows in the tray, which should be taken into account by the agent. This visibility checking should really be done by the developers working on the desktop environment, since they know best where a GUI is visible. This will be addressed in the third implementation.

Lastly, this implementation has the advantage of being the fastest to deploy. If developers of an app see that their app is taking too much CPU time when the user isn't using it, then they can easily fix that issue by adding an agent to their app. This implementation would serve as a good short-term solution, especially for input-driven processes since their solution has a very low overhead.

Chapter 5

Implementation 2: user space centralised

The second implementation will try to address the conceptual drawback of the first implementation, which was that the computation of the visibility for applications having real time data was spread across different agents. In this implementation, the calculation will be done in one single agent, which will do all the work for all the GUIs. Each app will then have to individually tell the agent whether they should be paused when not in focus or when not visible. Figure 5.1 below shows the overall structure of this implementation.

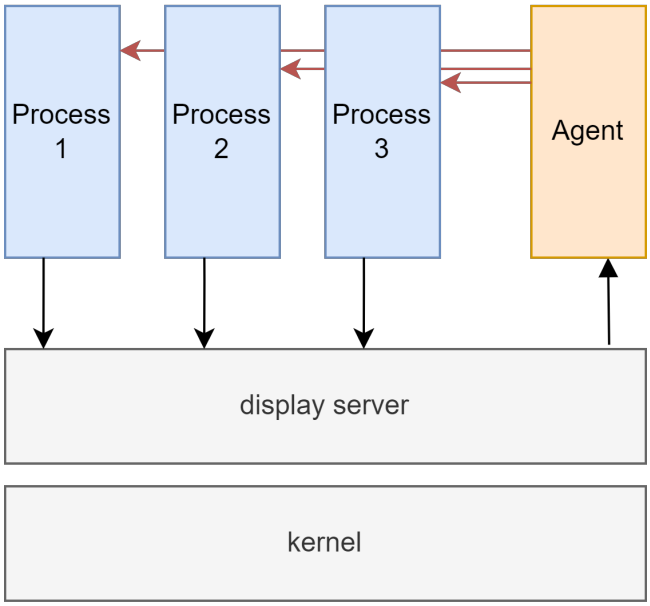

Figure 5.1: Conceptual design of this implementation where there is one central agent. The black arrows are processes reporting their category and the agent asking for window positions for instance, and the red are the signals.

5.1 Design

In order to pause input-driven processes, the agent must know when those are in focus or not. The issue is that the display server by default doesn't allow to receive events for a change in focus in other windows. Hence, the agent cannot get that information using a blocking wait. The agent must keep polling the state of every GUI to verify which one is in focus. A similar situation occurs for the visibility of a different window. The agent must also keep polling the display server to get their position and then calculate whether some are overlapping others. Hence, the agent will have a central loop were it checks for both focus of input-driven processes and visibility of RTD processes.

Compared to the first implementation where the communication between the process and its own agent was done internally through a shared memory page, here the communication is a bit more complex since all the processes must communicate individually with this central agent, which are developed by different developers at different times. The central agent must be able to know from each program whether it could be paused and for what GUI position. The first naive way to solve that problem is to enable this communication through files. The applications will write to a given file, and the agent will read that file on each iteration. The issue here is that the developers of the apps and of the agent must agree in advance what file that is, and where it is located. This can't change after the apps are built and deployed. Also, the user would be able to see those files which isn't a clean solution. Hence, it was decided to communicate via the window properties that the display server stores. Three new properties were added, one for whether the window can be paused if not in focus, one for whether it can be paused if not visible, one for remembering whether the app was paused (to reduce the overhead of sending too many signals), and finally the PIDs of all the processes that should be paused or resumed. Then, each application sets its own properties via the display server, and the agent simply has to read those properties. This is a much cleaner approach, not only because the properties of a window can be added to any Linux distro (this means the programmers don't have to hard code a communication for each distro), but also because that information is stored where it makes the most amount of sense, and also the user can't accidentally see them.

The script created to test this implementation can be found in appendix F. Again, the code was written in C in order to benefit from the highest speed, and from the Xlib library. The calculation for the visibility of the GUIs is the same as for the first implementation.

In terms of the run time analysis, it is $O(n^2)$ where n is the number of open GUIs. This indicates that as the number of open GUIs increases, the responsiveness of the windows will be more significantly impacted compared to the first implementation.

5.2 Overhead

In order to analyse the overhead of this central agent, it was first recorded while running individually for 10 seconds, using the same method as above. The result showed that on average, the agent takes 5413.3ms of CPU time every 10 seconds. This is almost the

same as the agent for the visibility of the first implementation. While this is again quite substantial, since there is only one of this central agent perhaps the overhead might be smaller overall.

The model will be used again to compare the overhead of this central agent with the original wasted work by the CPU, by looking at their ratio. The result is shown in figure 5.2.

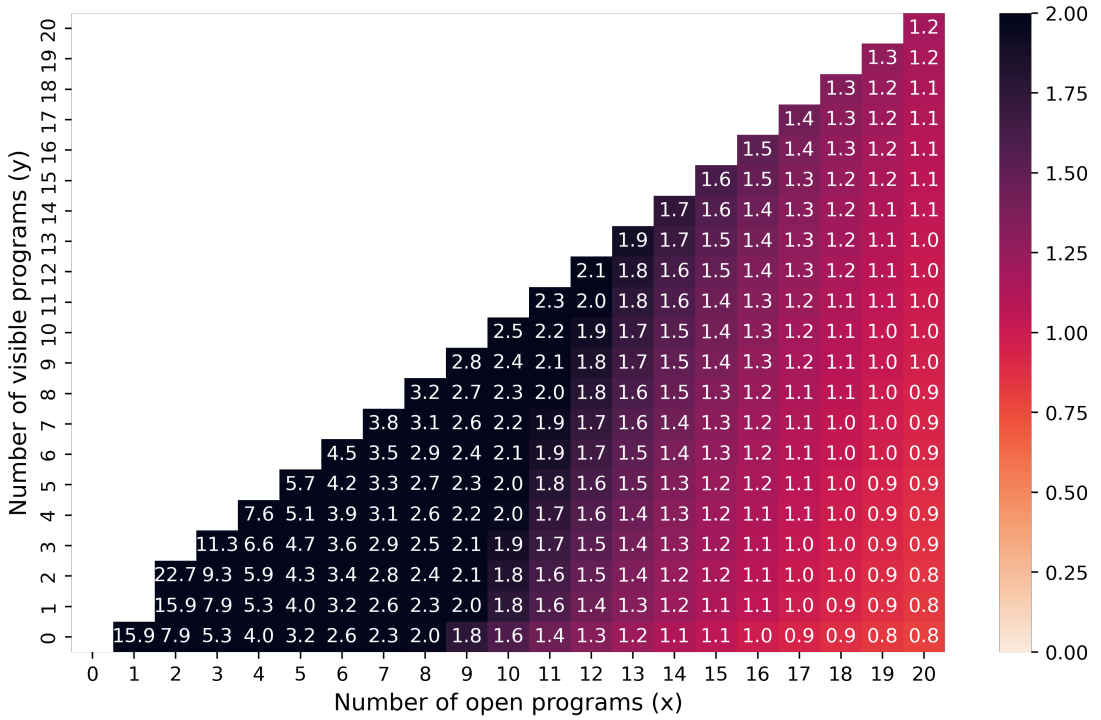

Figure 5.2: Ratio of overhead of the central compared to the original work done by the CPU.

As we can see, for most scenarios the ratio is greater than 1, hence the overhead of this central agent is greater than the original wasted work done by the CPU. Though, when the number of open apps is greater than about 17, the ratio starts to be lower than one, meaning that this implementation becomes advantageous in those cases. Though the ratio does remain quite close to one, so the agent isn't helping that much.

5.3 Responsiveness

Once again, the version of the agent analysed above is with the best responsiveness possible. The first thing to note is that a dynamic time between each iteration isn't that easy anymore. This is due to the presence of input-driven processes. There will always be one that isn't in focus (unless no programs are open) and hence the script must be kept in high speed to make sure it can revive it as soon as possible. The time between each iteration will have consequences on all the processes, which can't be adapted for each one of them.

Though, as noted above, perhaps the responsiveness can be very slightly worsened in order to heavily improve the overhead. Again, a new version of the agent was made with a time of 0.1 seconds between the iterations. The overhead of this new agent as analysed using the same technique as above. The model with this new version is shown in figure 5.3.

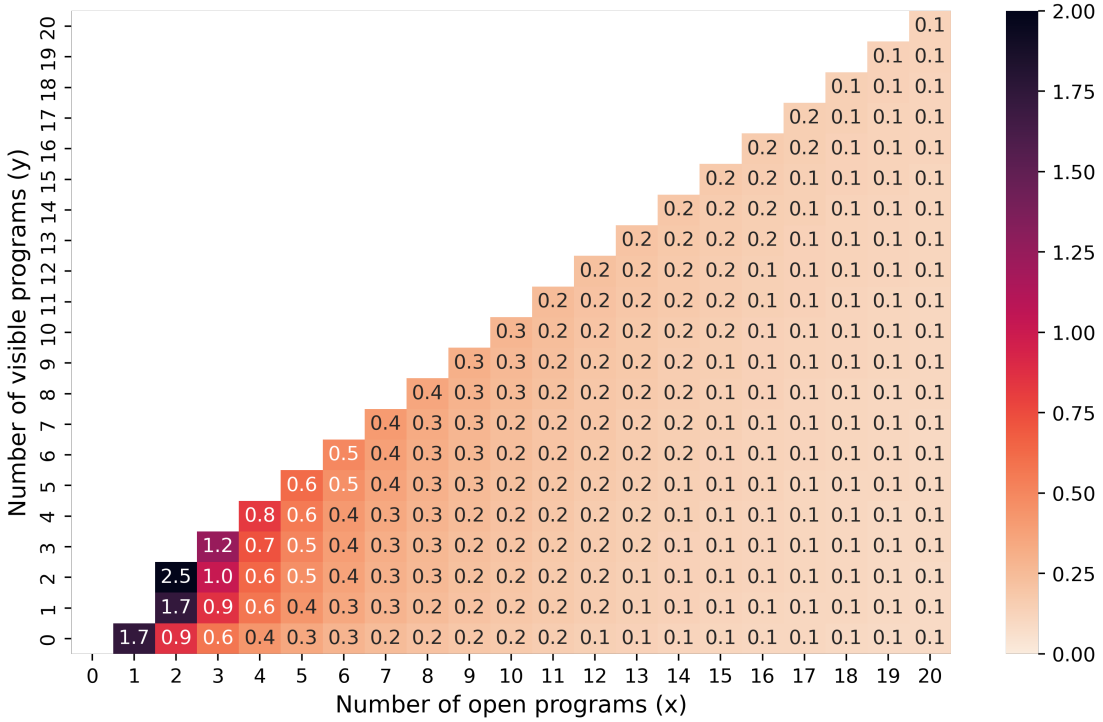

Figure 5.3: ratio of overhead to wasted time when the time between iterations is 0.1s

As we can see, the ratio for almost all the scenario is smaller than one. The only cases where the ratio is greater than one is is when the number of open programs is very small (less than 3). The ratio is also quite small on average, meaning that the agent manages to be significantly better than the original wasted time.

5.4 Discussion of implementation

As discussed above, it was shown that the overhead of this implementation is quite significant, and is only slightly advantageous when a lot of processes are running. Though, it was shown that slightly worsening the responsiveness will decrease the overhead which can make this implementation viable.

The key design decision of having only one agent brings the advantage that if there are a lot of RTD processes, their visibility can be calculated in one place and hence more efficiently. In such scenarios, then this implementation is more advantageous compared to the first implementation. However, a drawback of this design is that the responsiveness is uniform for all the processes. If there is one that needs a very good responsiveness, then the central agent must increase its overhead, which was shown that

it will quickly counterbalance the wasted work that it manages to save. Also, this central agent must keep busy waiting even if there is only input-driven processes that are open. In such a scenario, it was shown that the most optimal solution was to have a dedicated agent for each of those processes. In general, this shows that this implementation is only more adequate than the first implementation in the cases where there are a lot of RTD processes running on the system.

The conceptual advantage with this approach is that this agent can easily be deployed. Linux distributions can create a custom agent for their desktop environment (making a personalised calculation of the visibility of a GUI), and then make it be installed on their distro by default and then run on startup. This means the developers of apps only need to change the property of their windows by communicating with the display server, they don't need to do more. Developers of applications now don't have to think about all the different kinds of desktop environments their app could run in. Also, changing the property of their process from input-driven to RTD is as easy as changing one variable. Overall, the workflow of developers in this implementation is much easier compared to UWP for instance, and compared to the first implementation where a different agent must be used for the different types of process.

Another advantage over the first implementation is that the user can easily make an app pause when not in focus or not visible, since the communication and decision making is done via the display server properties. This means that even if the developers of an app forgot to implement this optimisation, it is still possible to apply it after their app is deployed.

Chapter 6

Implementation 3: display server

The third implementation will try to take advantage of the fact that the display server is already doing work related to the management of windows. For instance, the display server sends events to windows when certain properties or features change about them, such as when the window receives focus for instance. In this implementation, the display server X11 used in Ubuntu will be modified so that it does the job of the agent to pause the input-driven processes that are not in focus, and the RTD processes that are not visible. The version of X11 which was modified is 1.20.13, which was developed in a virtual machine running Ubuntu 20.04.5 LTS. Figure 6.1 below shows the overall structure of this implementation.

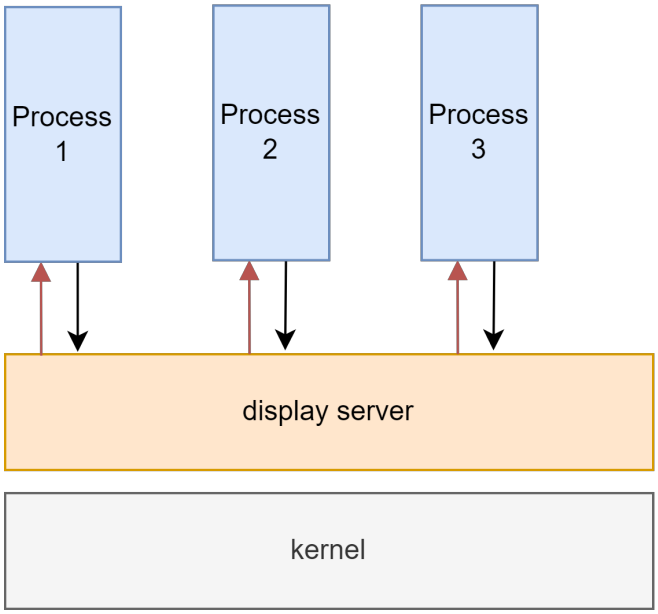

Figure 6.1: Conceptual design of this implementation where the display server becomes the agent. The black arrows are processes reporting their category, and the red are the signals.

6.1 Design

6.1.1 Input-driven processes

For input-driven processes, the solution is similar to the first implementation. The display server knows when a window receives or leaves focus, since it must send an event to that window if it asked for it. Hence, the display server was modified so that when it wants to send a *FocusChange* event (before it has checked whether the window subscribed to that event) it will first check whether the process under the GUI is of type input-driven, in which case it will pause or revive that process.

The modifications made to X11 can be found in appendix G. The code that must run is quite simple, it only involves a few lines of code. Once again, the communication between the display server and the processes is enabled using the windows properties that the process can change at any time.

6.1.2 RTD processes

For RTD processes, the approach taken is similar to the one above. The display server generates a $ConfigureNotify$ event whenever a window changes state, such as size, position, border, or stacking order. This means that whenever such an event is generated, a GUI has moved on the screen. Hence, before those events are generated, the visibility of the GUIs is calculated again, and the processes are paused or resumed as necessary. The way it is done is the same as the previous two implementations.

There are two small subtleties though. Sometimes, GUIs don't change position but still have their visibility changed. That is the case for instance when the active workspace changes. No $ConfigureNotify$ event is generated in that situation. Though, it is easy to check for that case. There is a $PropertyNotify$ event that is generated whenever a property of a window changes. The root window of the display server stores as a property the active workspace. Hence, whenever a $PropertyNotify$ event is generated that relates to the property of the active workspace of the root window, the visibility calculation is done again.

The second subtlety is when a GUI is minimised. In that case too, the $ConfigureNotify$ event isn't sent. Hence, the same way as for the active workspace, when the $PropertyNotify$ event related to whether a window is minimised is changed, the visibility calculation is done too.

The modifications made to X11 can be found in appendix H. The work that has to be done is a bit more complex compared to the first two implementations, since now the information must be first found and formatted before being used. For instance, finding all the GUIs must be done manually by recursively visiting all the children of the root window. Once again, the communication between the apps and the display server is implemented using the properties of a window.

6.2 overhead

6.2.1 Input-driven processes

The overhead of the code that checks whether the process should be paused based on its focus is bound to the *FocusChange* events. Those events are only generated when a window changes focus, hence the script isn't busy waiting. It can be seen as doing a blocking read on the focus of any window. Hence, the script will only be run when it is necessary. This implementation would manage to completely remove the wasted work done by the CPU for input-driven processes, while having no overhead in the background. In the model, the average CPU consumption would then be zero since the amount of time a window changes focus isn't taken into account.

Though, there is still some extra work that has to be done when the *FocusChange* events are generated. If those events are generated frequently enough, the overhead of this extra code could be greater than the original wasted work done by the CPU. From the model, it was found that on average one input-driven process wastes $238.5ms$ on the CPU every 10 seconds. After analysing the extra work done for every *FocusChange* event, it was found that it took on average $223.6s$ to do the extra work (which includes sending the signal). Since such an event is generated for both the GUI that leaves focus and the one that enters focus, in total the extra work done by the CPU takes $223.6s * 2 = 447.2s$. For the new version of X11 to consume as much CPU as what on average consumes an input-driven process, about $238500/447.2 = 533$ changes in window focus must be made during a 10 second interval. As this value is not obtainable in practice, and because this would only compensate for one input-driven process, this shows that on average the new implementation of X11 is always manage to reduce the work done by the CPU, regardless of the behaviour of the user.

What is interesting to note though here is that the extra work done is greater than the one for the first implementation ($223.6s$ compared to only $1.2s$ for the first implementation). This is primarily due to the fact that in this implementation, the display server much check whenever a *FocusChange* event is generated, what type of process is running behind it by accessing the properties the reported. In the first implementation though, that didn't have to be done since the presence of the agent was what determined whether the process would be paused or not. The agent in the first implementation didn't have to do any checks. This shows that the overhead of this implementation is greater than the one for the first implementation.

6.2.2 RTD processes

The overhead of the code that checks whether a process should be paused because it isn't visible is bound to two types of events, $ConfigureNotify$ and $PropertyNotify$. Those events are also only generated when those properties or configuration settings change value. Hence, the calculation for the visibility will not be run when it doesn't need to. Again, it could be seen as the script executing a blocking read until one of a necessary change requires a visibility calculated to fire again. This implementation would manage to completely remove the wasted work done by the CPU for RTD processes, while having no overhead in the background. In the model, the movements of GUIs isn't

taken in to account, which means that on average this implementation has an overhead of zero.

Though once again, we do have to do extra work when those events are generated. From the model, it was found that a single RTD process on average wasted $890.8ms$ during a 10 second interval. The extra work done by the script took about $105.1s$ to calculate the visibility of the windows each time the events were generated (when there were 20 programs launched, hence this is an overestimate of the average case). This means that in order for the new version of X11 to take more extra CPU time compared to the original wasted work, about $890800/105.1 = 9332$ of those events must be generated within 10 seconds. As this value is not obtainable in practice, and because more than one RTD processes can be running, this shows that the overhead of this implementation is much smaller than the original wasted work done by the CPU.

6.3 Responsiveness

The responsiveness of this implementation should be the best compared to the other two, since the display server is the first to know when windows change position or focus. As soon as the focus or visibility changes, an event is generated inside the display server, and even before it is sent the process can be paused or resumed.

What is interesting is that the artefact for RTD processes was noticeable using this implementation, though it was much smaller compared to the first two implementations. One solution to reduce this delay even further would be to optimise the current changes made to X11 so that the computation takes even less time. One way would be to store the intermediate results of the visibility from the previous generated event, which could dramatically reduce the overhead since the computation for the next event would be very similar since only one window can change position between two events. Another solution that can be looked into is to make the display server stop the rendering engine from showing the new window for as long as the computation of the visibility hasn't finished. While this will make the window take a bit more time to become visible, it will completely remove the artefact. The reason why this solution is possible is because the display server is the one responsible for the rendering of the windows, and hence implementing that change would be the easier for the display server to do. Though this is outside the scope of this report.

6.4 Discussion of implementation

As discussed above, the biggest advantage with this implementation is that when the windows are not moving around or not changing focus, the implementation will not cause the CPU to do unnecessary work. In those cases, the CPU will be completely spared of the original wasted work it had to do, while having very little extra work to do only when a GUI changes focus or position. This shows that this implementation has the lowest overhead compared to the first two. However, for input-driven processes, it was observed that this implementation was giving slightly more CPU time compared to the first implementation. The responsiveness was also shows to be the lowest for this

implementation, and that it could potentially completely remove the artefact of RTD processes.

A conceptual advantage with this implementation is that the visibility calculation is done by the display server. This not only means that it is only computed once compared to the first implementation, but also that module is the one that is the most apt to know that information, since it is the one that manages all the windows. Hence, now the work of the visibility is given to the developers that are the ones that know best when a certain window is visible or not. Now, each display server can adapt to its desktop environment and directly know whether a window is visible or not. The work necessary to solve the problem is done by the developers that are the most related to the issue.

Just like the second implementation, the life of app developers is very easy. They only need to change the property of their windows by communicating with the display server, they don't need to do more. Again, changing the property of their process from input-driven to RTD is as easy as changing one variable. Also, in this implementation the user can easily change the properties of certain processes to make them pause or resume if the developers didn't do their job properly.

Another advantage compared to the first two implementations is that no extra processes must be created to solve the problem. If a lot of input-driven processes are running using the first implementation, a lot of extra dummy processes must be made for each one of them. This does lead to an inefficient use of the resources available, and also gives more work to the scheduler. Hence, this implementation is the most efficient in terms of the use of resources.

Chapter 7

Conclusions

7.1 Magnitude of problem

In the third chapter, the magnitude of the problem was measured. Using a model to simulate different scenarios, it was shown that in most cases, the proportion of work that the CPU is doing that is wasted is quite significant. It was then shown that if the processes of type CW were refactored to extract their visual component, then the amount of wasted work that can be optimised is even higher. The general conclusion from this analysis is that first of all the problem of the position of the GUIs not being taken into account is quite significant, and hence this issue is worth optimising and investigating. Also, the results show that it is definitely worth refactoring all CW processes so that their visual component can easily be paused when it is not viewed by the user.

7.2 Implementations

Based on the results shown above, the first implementation seems to have the lowest overhead for the input-driven processes. For the RTD processes, it was shown that the lowest overhead was achieved in the third implementation, since in that case the display server manages only to compute the visibility when a window changes position, and hence it doesn't busy wait in the background. Regarding the responsiveness for RTD processes, it was shown that the best implementation is the third one, since in that case the artefact could also be completely removed.

This means that the optimal solution to solve the problem might be a mix of the first and third implementation. The input-driven processes are paused by an individual agent that executes busy reads to wait for the *FocusChange* events, while RTD processes are paused by the display server. Though, the gain in overhead for input-driven processes of the first implementation is quite small compared to the third implementation. Due to the fact that chaining from input-driven to RTD isn't easy in the first implementation, a cleaner more centralised approach of giving all the responsibility to the display server might be more judicious.

Though, the first and second implementations do have an advantage over the third one,

which is that they are much easier and faster to deploy, and hence they might be very advantageous in the short term for developers that want to improve the CPU utilisation of their app, or users that have a workflow where their CPU is doing a lot of wasted work because developers haven't yet fix the issue on their end.

A major difference between the third implementation proposed here and the ones that already exist like UWP or Android, is the work that the developers must do to enable that optimisation. In UWP and Android, the developers have to by default abide to implementing the callbacks, and find workarounds when their application must run when not visible. However, in this implementation the developers only have to set a certain property and that is all. It is much easier to change the status of a thread so it can continue running in the background compared to doing it in UWP or Android. It is also much easier for users to override the decision taken by the developers.

7.3 Verification

In order to verify that the solution works, and that the advantages predicted by the model are correct, a real life test was made, where the battery usage was measured to see whether and how much it was improved.

The battery consumption was measured when there is no optimisation in place, with the battery consumption when the third implementation is used. To have a proper accurate comparison where the only thing that changes between the two situations is the optimisation introduced, it was decided to have as setup 20 applications running, where none of them were visible (all in another workspace), and the computer was left untouched. If a scenario where the user is busy using the application under focus is used, it will be too hard to replicate that exact usage for both cases. Hence, leaving the computer unused while all the applications are running in the background for both scenarios should mean that the only difference is the introduction of the improved version of the display server. 20 applications are launched to make sure that a difference could be noticed between the normal and optimised scenario, which will enable the verification of the bottom right value in the result of the model, as shown in green in figure 3.2.

The modified display server was a running in a virtual machine, running on Ubuntu (also 20.04.5 TLS). In order to reduce the overhead of the host machine, every non essential processes were killed. This way, the actual work done by the CPU given to the host machine would be much smaller compared to what was running inside the virtual machine. Also, 7 virtual CPUs were given to the virtual machine out of the 8 for the host machine.

The normal scenario without optimisation lost about 55% of battery charge in one hour. The scenario with the optimised display server lost about 20% of battery charge during one hour. The graphs of the battery consumption can be found in appendix I. These results show that indeed the battery consumption does manage to be heavily improved thanks to pausing processes that don't need to run, and that the value reported by the model is quite accurate since it predicted a two-fold improvement. This result might

not generalise to all the cases covered by the model though, more experiments should be done to verify that.

7.4 Future work

One point that should be emphasised in this report is that the results described here are all based on a model. While it was tested thoroughly and verified with the battery test, it still remains slightly uncertain how precisely accurate it is. The next step of research in this area would be to gather empirical data, where the implementation proposed here is deployed on computers and where the overhead compared to the actual wasted work can be accurately compared. This would give interesting data where potentially some workflows can be discovered to have much higher wasted CPU time than others.

Another potential direction is to try to adopt the same implementation as UWP or Android. In the third implementation, the processes could easily report whether they needed to be paused, but perhaps it might be interesting to also make a coding environment where developers implement callbacks and different types of threads where some are sensible to them being visible and others not. This would make the development a bit more complicated for expert developers working on complicated pieces of software, but it might make the process for small apps and beginner developers easier to understand.

A last direction of research could be to create a new implementation that involves editing the scheduler itself. Perhaps a specialised scheduler can be made where it knows about GUIs and their position, which could lead to a more optimal decision making process, and hence a lower overhead. One way could be to use an ePBF program to extend the current scheduler. However, due to the infancy of this area, this report didn't explore that route yet.

7.5 Summary

In conclusion, this report first introduced the novel idea of using GUI position and the nature of the processes to show that the CPU is doing a lot of wasted work. Then, a model was created to measure the current amount of wasted work, which was determined to be quite high. Three different implementations were then proposed and analysed to measure their overhead, responsiveness and how easy the developer's work is to use it. Lastly, a battery test showed that the third implementation successfully managed to reach the value predicted by the model, showing that the implementation worked and that the model was accurate.

This report shows that while schedulers have heavily been optimised, they still remain quite general. The same Linux kernel is used to manage no-GUI processes in servers, and applications with real time data in Linux distros for personal computers. As this report shows, analysing the whole context from kernel to desktop environment introduces new requirements on the data, and hence new ways of optimising the scheduling decisions. This process should be applied to other contexts which might real other kinds of requirements and hence other optimisation opportunities.

Bibliography

[1] Android Studio by Google. The activity lifecycle. https://developer.android.com/guide/components/activities/activity-lifecycle, Accessed 2023-04-13.

[2] Investisdigital. The most visited sites of 2022. https://www.investisdigital.com/blog/technology/most-visited-sites-2022, Accessed 2023-04-13.

[3] Microsoft. Windows 10 universal windows platform (uwp) app lifecycle. https://learn.microsoft.com/en-us/windows/uwp/launch-resume/app-lifecycle, Accessed 2023-04-13.

Appendix A

Programs used for the model

The list below explains every program that was used for the model with a few details about how it was measured.

A.0.0.1 Firefox

The version used was 111.0.1. Firefox was determined to be input-driven, since most of the time the web pages that the user visits are simple static pages. From personal experience of the author, Firefox can be used very differently by different users. Some can have a few tabs open, while others will have a lot of them. Hence, it was decided to analyse Firefox in three different scenarios: once with one tab open, once with 10 tabs, and once with 50 tabs open. The final utilisation was then the average of those three scenarios. Also, the tabs were set to websites that were the most common for the year 2002[2].

A.0.0.2 Chrome

The version used was 111.0.5563.64. Chrome was determined to be input-driven for the same reason as Firefox. Chrome was analysed the same way as for Firefox.

A.0.0.3 Thunderbird

The version used was 102.9.0. Thunderbird was determined to be CW as it must run even when not visible.

A.0.0.4 LibreOffice Draw

The version used was 6.4.7.2 40(Build:2). It is a input-driven process.

A.0.0.5 LibreOffice Impress

The version used was 6.4.7.2 40(Build:2). It is a input-driven process, assuming that there are no animations one the slides for instance.

A.0.0.6 LibreOffice Write

The version used was 6.4.7.2 40(Build:2). It is a input-driven process, assuming that there are no animations one the document for instance.

A.0.0.7 LibreOffice Calc

The version used was 6.4.7.2 40(Build:2). It is a input-driven process.

A.0.0.8 VLC

The version used was 3.0.9.2. It is a input-driven process, since it is assumed that its main purpose is to watch movies. Hence, when it isn't in the user focus, it is likely that the user isn't looking at the movie anymore, and hence that it is paused. This means VLC is only waiting for input of the user to continue executing.

A.0.0.9 Shotcut

The version used was 20.02.21. It is a input-driven process, assuming that when it is in the background, the preview of the currently edited video is not playing.

A.0.0.10 GIMP

The version used was 2.10.30. It is a input-driven process, assuming no animations are playing on the image.

A.0.0.11 Visual Studio Code

The version used was 1.77.3. It is a input-driven process.

A.0.0.12 Ardour

The version used was 7.0.0. It is a input-driven process, assuming that when it is in the background no music is playing.

A.0.0.13 Discord

The version used was 0.0.26. Discord was determined to be CW as it must run even when not visible to detect when a new message arrives for instance.

A.0.0.14 Intellij IDEA

The version used was 222.4345.14. It is an input-driven process.

A.0.0.15 Pycharm

The version used was 2022.2.3. It is an input-driven process.

A.0.0.16 gnome-system-monitor

The version used was 42.0. This process has the property of having different tabs. The most commonly used ones is the “processes” tab and the “resources” tab. The first is a RTD process, while the second is a CW process. Both were analysed separately as if they were two separate apps.

A.0.0.17 Cheese

The version used was 3.34.0. It is a RTD process.

A.0.0.18 Blender

The version used was 3.3.1. It is an input-driven process.

A.0.0.19 Spotify

The version used was 1.2.8.923.g4f94bf0d. Spotify can either be found while playing music, or not. In the first case, it is a CW process. In the latter case, it is a input-driven process. The two separate cases were analysed separately.

Appendix B

Method of measurement of CPU utilisation

The following is the bash script that was used to measure the CPU utilisation of a process:

```
function record() {

    for i in {1..12}
    do

        # get cuttent cpu time received up to now
        local stats=$(cat "/proc/$1/stat")
        local statsarr=($stats)
        local utime_old=${statsarr[13]}
        local stime_old=${statsarr[14]}

        # let program run for 10 seconds
        sleep 10

        # if program doesn't exist, then exit the script
        if ! test -f "/proc/$1/stat"; then
                exit
        fi

        # get new cpu time
        local stats=$(cat "/proc/$1/stat")
        local statsarr=($stats)
        local utime_new=${statsarr[13]}
        local stime_new=${statsarr[14]}

        # see how much utime and stime it received during 10 seconds
        local utime_total=$(echo $utime_new - $utime_old | bc -l)
        local stime_total=$(echo $stime_new - $stime_old | bc -l)

        # calculate the number of seconds the process received
        local CLK_TCK=$(getconf CLK_TCK)
        local utime_sec=$(echo $utime_total / $CLK_TCK | bc -l)
        local stime_sec=$(echo $stime_total / $CLK_TCK | bc -l)
```

```
        local total_sec=$(echo $utime_sec + $stime_sec | bc -l)

        if (( $(echo "$total_sec > 0" |bc -l) )); then
            echo $total_sec
            fi

    done
}

if test -f "/proc/$1/stat"; then
    record $1
fi
```

B.0.0.1 Example: Cheese

As example, the results of the recording of the app Cheese are shown below.

Cheese was first recorded when minimised to see how much the CPU spent working on it. Its 12 samples are shown in figure B.1. As we can see, its CPU usage is quite consistent throughout the samples, and hence the average of those samples was taken to represent the average amount of time that the CPU wastes on Cheese while it is hidden. Also, it was recorded that this app does no useful work in the background.

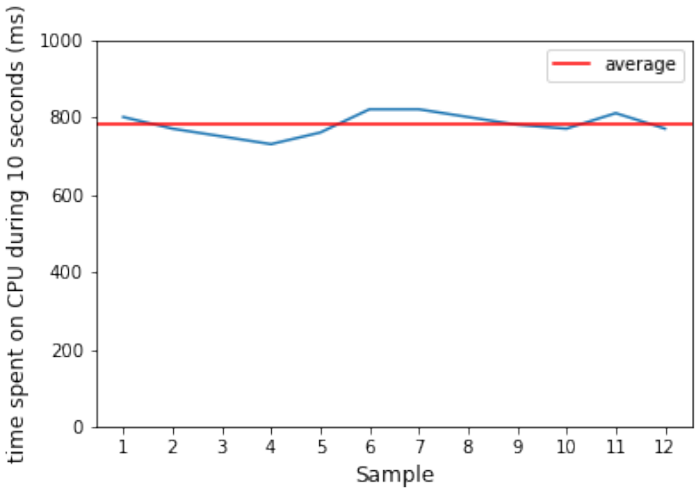

Figure B.1: amount of ms given to cheese during 12 samples of 10 seconds, while cheese is in the background

Cheese was then recorded when in the foreground. Its 12 samples are shown in figure B.2. As we can see, its CPU usage is again quite consistent throughout the samples, and also very similar to what it consumed when not visible (as expected since linux doesn't do anything special in terms of visiblity). The average of those samples was taken to represent the average amount of ms that the CPU spends on useful work in the foreground, and hence no wasted word was recorded in the foreground.

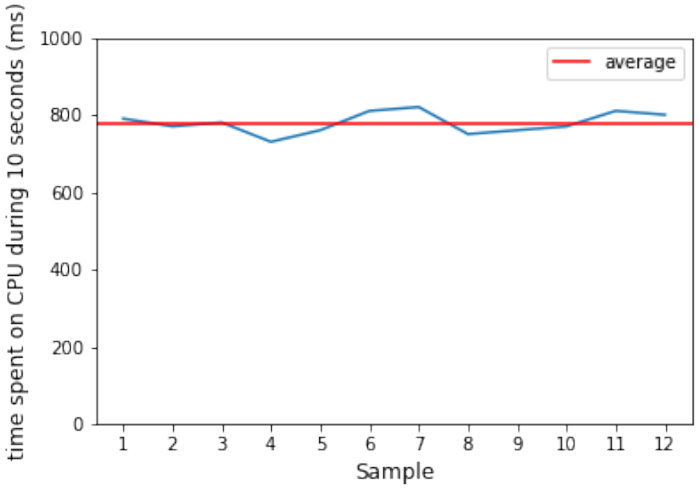

Figure B.2: amount of ms given to cheese during 12 samples of 10 seconds, while cheese is in the foreground

Cheese was then recorded when in the focus of the user, while the user is busy using the app. Its 12 samples are shown in figure B.3. As we can see, the CPU usage isn't that constant. Though, it doesn't decrease or increase over the 2 minute interval. Hence, the average of those samples was used as the amount of time that was spent on Cheese.

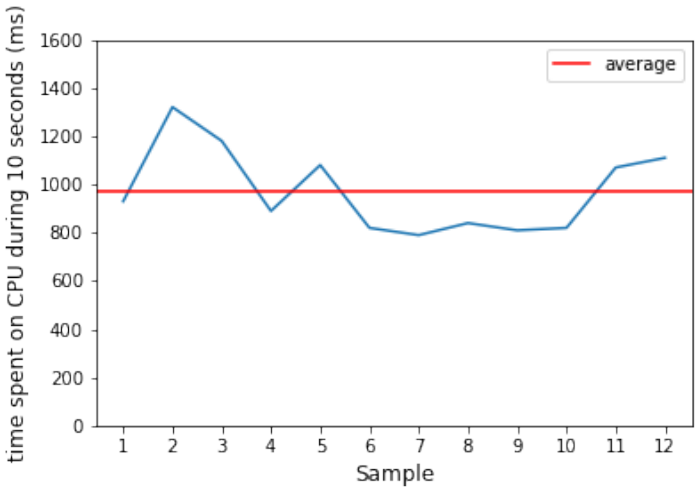

Figure B.3: amount of ms given to cheese during 12 samples of 10 seconds, while cheese is in the use focus while being used

Appendix C

Estimation of visual component for CW

The estimation of the visual component of processes of type CW was done based on the four programs of that type, which where: Thunderbird, Spotify when music is playing, gnome-system-monitor on the “resources” tab, and Discord.

The source code of gnome-system-monitor was used to measure how much time was spent on the visualisation and on the gathering of the data. It was found that when it had spent 1119.525ms on the CPU, only 32.194ms were spent on the gathering of the data (useful work). Hence, that means about 97% of what the CPU did was spent on the visualisation.

For Spotify, it was not possible to find the source code. Though when it was launched, a lot of processes were running. It was found that the total work that those processes did took about 1525.833ms during a 10 second interval. Though, by sending signals, it was possible to reduce the number of running processes, while Spotify could still play the music in the background. Hence, what was killed was only the work done related to the GUI. The least amount of work that was still enabling to listen to music was about 0.566ms every 10 seconds. This implies that Spotifies spends on average 99% of the time on visual computation.

For Discord, a similar approach as Spotify was used, and it was bout that about 94.5% is spent on visual work.

No data was found about Thunderbird.

Using the information gathered above, it was decided to take the value that on average 95% of what the CW processes do on the CPU is work related to visualising of data.

Appendix D

Script for input-driven processes of implementation 1

The following is the script for pausing input-driven processes discussed in implementation 1.

```
#include <X11/X.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/XKBlib.h>
#include <X11/Xatom.h>
#include <stdio.h>
#include <err.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdint.h>
#include <signal.h>
#include <sys/time.h>

#define POSX 500
#define POSY 500
#define WIDTH 500
#define HEIGHT 500
#define BORDER 15

static Display* dpy;
static int scr;
static Window root;

Window newWindow;
int pidToKill = 11231;

static Window createWindow(int x, int y, int w, int h, int b, Window parentWin) {
    Window win;
    XSetWindowAttributes xwa;

    xwa.background_pixel = BlackPixel(dpy, scr);
    xwa.border_pixel = BlackPixel(dpy, scr);
    xwa.event_mask = FocusChangeMask;
```

```
    win = XCreateWindow(dpy, parentWin, x, y, w, h, b, DefaultDepth(dpy, scr), InputOutput,
        ↪ DefaultVisual(dpy, scr),
        CWBackPixel | CWBorderPixel | CWEventMask, &xwa);

    return win;
}

static void run() {

    XEvent ev;
    while (XNextEvent(dpy, &ev) == 0) {
        switch(ev.type)
        {
            case FocusIn:
                kill(pidToKill, SIGCONT);
                break;

            case FocusOut:
                kill(pidToKill, SIGSTOP);
                break;
        }
    }
}

int main () {

    dpy = XOpenDisplay(NULL);
    if (dpy == NULL) {
        errx(1, "Can't open dislay");
    }

    scr = DefaultScreen(dpy);
    root = RootWindow(dpy, scr);
    newWindow = createWindow(POSX, POSY, WIDTH, HEIGHT, BORDER, root);
    XMapWindow(dpy, newWindow);

    run();

    XUnmapWindow(dpy, newWindow);
    XDestroyWindow(dpy, newWindow);
    XCloseDisplay(dpy);
    return 0;
}
```

Appendix E

Script for RTD processes of implementation 1

The following is the script for pausing RTD processes discussed in implementation 1.

```
#include <X11/X.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/XKBlib.h>
#include <X11/Xatom.h>
#include <stdio.h>
#include <err.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdint.h>
#include <signal.h>
#include <time.h>

#define POSX 500
#define POSY 500
#define WIDTH 500
#define HEIGHT 500
#define BORDER 15

static Display* dpy;
static int scr;
static Window root;
Window newWindow;

int isRunning = 1;
int pidToKill = 4810;

typedef struct range {
   int start;
   int end;
   int shouldBeRemoved;
} range_t;

int catcher( Display *disp, XErrorEvent *xe )
{
```

```
    printf( "An error occured with error code: %d\n", xe->error_code);
    return 0;
}

static Window createWindow(int x, int y, int w, int h, int b, Window parentWin) {
    Window win;
    XSetWindowAttributes xwa;

    xwa.background_pixel = BlackPixel(dpy, scr);
    xwa.border_pixel = BlackPixel(dpy, scr);

    win = XCreateWindow(dpy, parentWin, x, y, w, h, b, DefaultDepth(dpy, scr), InputOutput,
        ↪ DefaultVisual(dpy, scr),
        CWBackPixel | CWBorderPixel, &xwa);

    return win;
}

static int getActiveWorkspace() {
    Atom atom = XInternAtom(dpy, "_NET_CURRENT_DESKTOP", False);
    if (atom == None) {
        fprintf(stderr, "Failed to get atom\n");
        exit(1);
    }

    Atom actual_type;
    int actual_format;
    unsigned long nitems, bytes_after;
    unsigned char *prop;

    if (XGetWindowProperty(dpy, root, atom, 0, 1, False, XA_CARDINAL,
                           &actual_type, &actual_format, &nitems, &bytes_after,
                           &prop) != Success) {
        fprintf(stderr, "Failed to get property\n");
        exit(1);
    }

    if (actual_format != 32 || nitems != 1) {
        fprintf(stderr, "Invalid format or number of items\n");
        exit(1);
    }

    int workspace = *(int *)prop;

    return workspace;
}

static int getWorkspace(Window window) {

    Atom actual_type;
    int actual_format;
    unsigned long nitems, bytes_after;
    unsigned char *prop;
    int workspace;
    int status = XGetWindowProperty(dpy, window, XInternAtom(dpy, "_NET_WM_DESKTOP",
        ↪ True),
```

```
                                        0L, 1L, False, AnyPropertyType, &actual_type, &
                                            ↪ actual_format, &nitems, &bytes_after, &prop);
    if (status == Success && nitems > 0) {
        workspace = *(long*) prop;
        XFree(prop);
        return workspace;
    }
    else {
        return -1;
    }
}

static int isMinimisedNew(Window window) {
    Atom wm_state;
    Atom minimized;
    Atom type;
    int format;
    unsigned long nitems, bytes_after;
    unsigned char *prop;

    wm_state = XInternAtom(dpy, "_NET_WM_STATE", True);
    minimized = XInternAtom(dpy, "_NET_WM_STATE_HIDDEN", True);

    if (wm_state == None || minimized == None) {
        return 0;
    }

    // Get the value of the window state property
    if (XGetWindowProperty(dpy, window, wm_state, 0, 1024, False,
                           XA_ATOM, &type, &format, &nitems, &bytes_after,
                           &prop) != Success) {
        printf("Error: Could not get window state property\n");
        exit(1);
    }

    // Check if the minimized property is present in the window state property
    for (int i = 0; i < nitems; i++) {
        if (((Atom *)prop)[i] == minimized) {
        XFree(prop);
        return 1;
        }
    }

    XFree(prop);
    return 0;
}

static void findAllWindowsStackOrder(Window window, Window** allWindows, int*
    ↪ numAllWindows) {

    Window rootWindow;
    Window parent;
    Window *children = NULL;
    unsigned int num_children;

    if (!XQueryTree(dpy, window, &rootWindow, &parent, &children, &num_children)) {
```

```
        printf("can't query tree \n");
        return;
    }

    // add youself to the list only if you are not the root
    if (window != root) {
        Window* newAllWindows = malloc(sizeof(Window) * (*numAllWindows + 1));
        for (int i = 0; i < *numAllWindows; i++) {
            newAllWindows[i] = (*allWindows)[i];
        }
        newAllWindows[*numAllWindows] = window;
        (*numAllWindows)++;
        free(*allWindows);
        *allWindows = newAllWindows;
    }

    for (int i = num_children - 1; i >= 0; i--) {
        findAllWindowsStackOrder(children[i], allWindows, numAllWindows);
    }

    if (children != NULL) {
        XFree(children);
    }
}

static void getMaxXandY(int* maxX, int* maxY) {
    XWindowAttributes xwa;
    XGetWindowAttributes( dpy, root, &xwa );
    *maxX = xwa.width;
    *maxY = xwa.height;
}


void stopProgram (char* message) {
  if (isRunning == 1) {
    if (pidToKill > 0) {
      kill(pidToKill, SIGSTOP);
    }
    printf("stopped : %s \n", message);
    isRunning = 0;
  }
}

void resumeProgram (char* message) {
  if (isRunning == 0) {
    if (pidToKill > 0) {
      kill(pidToKill, SIGCONT);
    }
    printf("resumed : %s \n", message);
    isRunning = 1;
  }
}

int main( int argc, char *argv[]) {
    dpy = XOpenDisplay(NULL);
    if (dpy == NULL) {
```

```
        errx(1, "Can't open dislay");
    }
    scr = DefaultScreen(dpy);
    root = DefaultRootWindow(dpy);
    newWindow = createWindow(POSX, POSY, WIDTH, HEIGHT, BORDER, root);
    XMapWindow(dpy, newWindow);
    XSetErrorHandler( catcher );
    while (1) {

        usleep(0 * 1000 * 1000); // the first value in this multiplication is the number of seconds

        // first check if this window is in the current active workspace
        int activeWorkspace = getActiveWorkspace();
        int ourWorkspace = getWorkspace(newWindow);
        if (ourWorkspace == -1) {
            printf("couldn't get the current workspace for some reason \n");
            continue;
        }
        if (activeWorkspace != ourWorkspace) {
            stopProgram("window in another workspace");
            continue;
        }
        // now check if the window is minmised
        if (isMinimisedNew(newWindow) == 1) {
            stopProgram("window in current workspace but minimised");
            continue;
        }
        // now you know you are in the active workspace and visible, so check all other open windows
            ↪ to see if you are covered

        // get your current coordinates
        int x_i, y_i, width_i, height_i;
        Window child;
        XTranslateCoordinates( dpy, newWindow, root, 0, 0, &x_i, &y_i, &child );
        XWindowAttributes xwa;
        XGetWindowAttributes( dpy, newWindow, &xwa );
        width_i = xwa.width;
        height_i = xwa.height;

        int maxX, maxY;
        getMaxXandY(&maxX, &maxY);

        // check if the window is completely too far left or right, up or down
        if (x_i + width_i <= 0 || x_i >= maxX || y_i + height_i <= 0 || y_i >= maxY) {
            stopProgram("window completely gone from screen");
            continue;
        }

        int haveFoundSomeoneOnTop = 0;

        // check fror each window from the topmost to the lowest
        Window* allWindows = NULL;
        int numAllWindows = 0;
        findAllWindowsStackOrder(root, &allWindows, &numAllWindows);

        for (int i = 0; i < numAllWindows; i++) {
```

```
            if (allWindows[i] == newWindow) {
                break; // we have to stop here, as the further windows are below ours
            }

            // check that this window is in the correct workspace
            int otherWorkspace = getWorkspace(allWindows[i]);
            if (otherWorkspace != ourWorkspace) {
                continue;
            }

            // check that the window is not minimised
            if (isMinimisedNew(allWindows[i]) == 1) {
                continue;
            }

            // now check whether this window is covering
            int x_j, y_j, width_j, height_j;
            Window child;
            XTranslateCoordinates( dpy, allWindows[i], root, 0, 0, &x_j, &y_j, &child );
            XWindowAttributes xwa;
            XGetWindowAttributes( dpy, allWindows[i], &xwa );
            width_j = xwa.width;
            height_j = xwa.height;

            if (x_j <= x_i && x_j + width_j >= x_i + width_i &&
                y_j <= y_i && y_j + height_j >= y_i + height_i) {
                  stopProgram("covered_by_others");
                  haveFoundSomeoneOnTop = 1;
                  break;
            }
        }

        // if you haven't been killed, meaning you are visible, you must be revived
        if (haveFoundSomeoneOnTop == 0) {
          resumeProgram("visible");
        }

        free(allWindows);
    }

    XUnmapWindow(dpy, newWindow);
    XDestroyWindow(dpy, newWindow);
    XCloseDisplay(dpy);

    return 0;
}
```

Appendix F

Script for agent of implementation 2

The following is the script for the agent used in implementation 2.

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>
#include <X11/X.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/XKBlib.h>
#include <X11/Xatom.h>
#include <time.h>

typedef struct program_t {
   char* name;
   int* pids;
   int numPids;
   int winId;
   int isRunning;
   int shouldKillIfNoFocus;
} program_t;

typedef struct range {
   int start;
   int end;
   int shouldBeRemoved;
} range_t;

static Display* dpy;
static int scr;
static Window root;

typedef struct window_coord {
   int x;
   int y;
   int width;
   int height;
```

```
} window_coord_t;

void getMaxXandY(int* maxX, int* maxY) {
    XWindowAttributes xwa;
    XGetWindowAttributes( dpy, root, &xwa );
    *maxX = xwa.width;
    *maxY = xwa.height;
}

int isMinimisedNew(Window window) {
    Atom wm_state;
    Atom minimized;
    Atom type;
    int format;
    unsigned long nitems, bytes_after;
    unsigned char *prop;
    wm_state = XInternAtom(dpy, "_NET_WM_STATE", True);
    minimized = XInternAtom(dpy, "_NET_WM_STATE_HIDDEN", True);

    if (wm_state == None || minimized == None) {
        return 0;
    }
    // Get the value of the window state property
    if (XGetWindowProperty(dpy, window, wm_state, 0, 1024, False,
                                XA_ATOM, &type, &format, &nitems, &bytes_after,
                                &prop) != Success) {
        printf("Error: Could not get window state property\n");
        return 0;
    }
    // Check if the minimized property is present in the window state property
    for (int i = 0; i < nitems; i++) {
        if (((Atom *)prop)[i] == minimized) {
        XFree(prop);
        return 1;
        }
    }
    XFree(prop);
    return 0;
}

void getPids(Window window, unsigned char**propReturn, unsigned long* nitemsReturn) {
    Atom type;
    int format;
    unsigned long nitems, bytes_after;
    unsigned char *prop;
    Atom pids = XInternAtom(dpy, "pids", False);
    if (pids == None) {
      return;
    }
    // Get the value of the window state property
    if (XGetWindowProperty(dpy, window, pids, 0, 1024, False,
                                AnyPropertyType, &type, &format, &nitems, &bytes_after,
                                &prop) != Success) {
        printf("Error: Could not get pids property\n");
        return;
    }
```

```
    *nitemsReturn = nitems;
    *propReturn = prop;
}

int getActiveWorkspace() {
    Atom atom = XInternAtom(dpy, "_NET_CURRENT_DESKTOP", False);
    if (atom == None) {
        fprintf(stderr, "Failed_to_get_atom\n");
        exit(1);
    }
    Atom actual_type;
    int actual_format;
    unsigned long nitems, bytes_after;
    unsigned char *prop;

    if (XGetWindowProperty(dpy, root, atom, 0, 1, False, XA_CARDINAL,
                           &actual_type, &actual_format, &nitems, &bytes_after,
                           &prop) != Success) {
        fprintf(stderr, "Failed_to_get_property\n");
        return -1;
    }
    if (actual_format != 32 || nitems != 1) {
        fprintf(stderr, "Invalid_format_or_number_of_items\n");
        return -1;
    }
    int workspace = *(int *)prop;
    return workspace;
}

int getWorkspace(Window window) {
  Atom actual_type;
  int actual_format;
  unsigned long nitems, bytes_after;
  unsigned char *prop;
  int workspace;
  int status = XGetWindowProperty(dpy, window, XInternAtom(dpy, "_NET_WM_DESKTOP", True)
      ↪ ,
                                  0L, 1L, False, AnyPropertyType, &actual_type, &
                                      ↪ actual_format, &nitems, &bytes_after, &prop);
  if (status == Success && nitems > 0) {
      workspace = *(long*) prop;
      XFree(prop);
      return workspace;
  }
  else {
      return -1;
  }
}

int checkIfFocused (Window window) {
    Atom actual_type;
    int actual_format;
    unsigned long nitems, bytes_after;
    Atom *atoms;
    XGetWindowProperty(dpy, window,
                       XInternAtom(dpy, "_NET_WM_STATE", True),
```

```
                                0, 1024, False, XA_ATOM, &actual_type,
                                &actual_format, &nitems, &bytes_after,
                                (unsigned char **)&atoms);
    Atom focusedAtom = XInternAtom(dpy, "_NET_WM_STATE_FOCUSED", True);
    int found = 0;
    for (int i = 0; i < nitems; i++) {
        if (atoms[i] == focusedAtom) {
            found = 1;
            break;
        }
    }
    return found;
}

void setIsRunning(Window window, unsigned char value) {
  Atom atomIsRunning = XInternAtom(dpy, "isRunning", False);
  Atom atomINTEGER = XInternAtom(dpy, "INTEGER", True);
  unsigned char* dataRunning = (unsigned char*) malloc(sizeof(unsigned char));
  *dataRunning = value;
  XChangeProperty(dpy, window, atomIsRunning, atomINTEGER, 8, PropModeReplace,
      ↪ dataRunning, 1);
}

unsigned char getIsRunning(Window window) {
  Atom actual_type;
  int actual_format;
  unsigned long nitems, bytes_after;
  unsigned char *prop;
  int isRunning;
  Atom atomIsRunning = XInternAtom(dpy, "isRunning", False);
  if (atomIsRunning == None) {
    printf("couldn't get isRunning atom\n");
    return -1;
  }
  int status = XGetWindowProperty(dpy, window, atomIsRunning,
                                  0, 1, False, AnyPropertyType, &actual_type, &actual_format,
                                      ↪ &nitems, &bytes_after, &prop);
  if (status == Success && nitems > 0) {
      isRunning = *prop;
      XFree(prop);
      return isRunning;
  }
  else {
      return -1;
  }
}

void findAllWindowsStackOrder(Window window, Window** allWindows, int* numAllWindows) {
    Window rootWindow;
    Window parent;
    Window *children = NULL;
    unsigned int num_children;
    if (!XQueryTree(dpy, window, &rootWindow, &parent, &children, &num_children)) {
        printf("can't query tree \n");
        return;
    }
```

```
    // add youself to the list only if you are not the root
    if (window != root) {
        Window* newAllWindows = malloc(sizeof(Window) * (*numAllWindows + 1));
        for (int i = 0; i < *numAllWindows; i++) {
            newAllWindows[i] = (*allWindows)[i];
        }
        newAllWindows[*numAllWindows] = window;
        (*numAllWindows)++;
        free(*allWindows);
        *allWindows = newAllWindows;
    }
    for (int i = num_children - 1; i >= 0; i--) {
        findAllWindowsStackOrder(children[i], allWindows, numAllWindows);
    }
    if (children != NULL) {
        XFree(children);
    }
}

unsigned char getShouldKillIfNoFocus(Window window) {
  Atom actual_type;
  int actual_format;
  unsigned long nitems, bytes_after;
  unsigned char *prop;
  unsigned char shouldKill;
  Atom atomKill = XInternAtom(dpy, "shouldKillNoFocus", False);
  if (atomKill == None) {
    printf("couldn't get shouldkillnofocus atom\n");
    return 0;
  }
  int status = XGetWindowProperty(dpy, window, atomKill,
                                  0, 1, False, AnyPropertyType, &actual_type, &actual_format,
                                      ↪ &nitems, &bytes_after, &prop);
  if (status == Success && nitems > 0) {
      shouldKill = *prop;
      XFree(prop);
      return shouldKill;
  }
  else {
      return 0;
  }
}

unsigned char getShouldKillNotVisible(Window window) {
  Atom actual_type;
  int actual_format;
  unsigned long nitems, bytes_after;
  unsigned char *prop;
  unsigned char shouldKill;
  Atom atomKill = XInternAtom(dpy, "shouldKillNotVisible", False);
  if (atomKill == None) {
    printf("couldn't get shouldkillnotVisible atom\n");
    return 0;
  }
  int status = XGetWindowProperty(dpy, window, atomKill,
                                  0, 1, False, AnyPropertyType, &actual_type, &actual_format,
```

```
                                        ↪ &nitems, &bytes_after, &prop);
  if (status == Success && nitems > 0) {
      shouldKill = *prop;
      XFree(prop);
      return shouldKill;
  }
  else {
      return 0;
  }
}


void stopProgram (Window window, char* message) {
  unsigned char isRunning = getIsRunning(window);
  if (isRunning == 1) {
    unsigned char* prop = NULL;
    unsigned long nitems = 0;
    getPids(window, &prop, &nitems);
    for (int indexPid = 0; indexPid < nitems; indexPid++) {
      kill(((u_int32_t *) prop)[indexPid], SIGSTOP);
    }
    XFree(prop);
    printf("stopped window id %lx %s \n", window, message);
    // record that this program is not running
    setIsRunning(window, 0);
  }
}

void resumeProgram (Window window, char* message) {
  unsigned char isRunning = getIsRunning(window);
  if (isRunning == 0) {
    unsigned char* prop = NULL;
    unsigned long nitems = 0;
    getPids(window, &prop, &nitems);
    for (int indexPid = 0; indexPid < nitems; indexPid++) {
      kill(((u_int32_t *) prop)[indexPid], SIGCONT);
    }
    XFree(prop);
    printf("resumed window id %lx %s \n", window, message);
    // record that this program is running
    setIsRunning(window, 1);
  }
}

int catcher( Display *disp, XErrorEvent *xe )
{
  printf( "An error occured with error code: %d\n", xe->error_code);
  return 0;
}

int main( int argc, char *argv[] ) {

  // open connection with xserver
  dpy = XOpenDisplay(NULL);
  if (dpy == NULL) {
      printf("Can't open dislay \n");
```

```
    exit(1);
}
scr = DefaultScreen(dpy);
root = DefaultRootWindow(dpy);
// get dimension of the screen
int maxX, maxY;
getMaxXandY(&maxX, &maxY);
XSetErrorHandler( catcher );
while (1) {

  //usleep(0.1 * 1000 * 1000); // the first value in this multiplication is the number of seconds

  int activeWorkspace = getActiveWorkspace();
  window_coord_t* allWindowsVisible = NULL;
  int numWindowsVisible = 0;
  // get all the open windows in stack order from top to bottom
  Window* allWindows = NULL;
  int numAllWindows = 0;
  findAllWindowsStackOrder(root, &allWindows, &numAllWindows);
  // * iterate through the open windows from top to bottom
  for (int i = 0; i < numAllWindows; i++) {
    // find its position
    int workspaceProgram = getWorkspace(allWindows[i]);
    int differentWorkspace = workspaceProgram != -1 && workspaceProgram != activeWorkspace
        ↪ ;
    int isMinimised = isMinimisedNew(allWindows[i]);

    // get its coodinates on the screen
    int x_i, y_i, width_i, height_i;
    Window child;
    XTranslateCoordinates(dpy, allWindows[i], root, 0, 0, &x_i, &y_i, &child);
    XWindowAttributes xwa;
    XGetWindowAttributes( dpy, allWindows[i], &xwa );
    width_i = xwa.width;
    height_i = xwa.height;
    // clip them
    x_i = x_i > 0 ? x_i : 0;
    width_i = x_i + width_i > maxX ? maxX - x_i : width_i;
    y_i = y_i > 0 ? y_i : 0;
    height_i = y_i + height_i > maxY ? maxY - y_i : height_i;

    if (getShouldKillIfNoFocus(allWindows[i]) == 1) {
      if (checkIfFocused(allWindows[i])) {
        resumeProgram(allWindows[i], "focused");
      } else {
        stopProgram(allWindows[i], "not_in_focus");
      }
    }
    // you are a visibility window, check whether you are visible hidden
    else if (getShouldKillNotVisible(allWindows[i]) == 1) {
      if (differentWorkspace) {
        stopProgram(allWindows[i], "different_workspace");
      }
      else if (isMinimised) {
        stopProgram(allWindows[i], "minimised");
      }
```

```
        // you are in the current workspace so you must check other windows
        else {
          // check if you are covered by a visible window
          int foundSomeOneOnTop = 0;

          for (int h = 0; h < numWindowsVisible; h++) {
            if (allWindowsVisible[h].x <= x_i && allWindowsVisible[h].x + allWindowsVisible[h].
                ↪ width >= x_i + width_i &&
                allWindowsVisible[h].y <= y_i && allWindowsVisible[h].y + allWindowsVisible[h].
                    ↪ height >= y_i + height_i) {
              stopProgram(allWindows[i], "covered_by_others");
              foundSomeOneOnTop = 1;
              break;
            }
          }
          if (!foundSomeOneOnTop) {
            resumeProgram(allWindows[i], "visible");
          }
        }
      }
      else {
        // just make sure that when a property changes we can be revived
        resumeProgram(allWindows[i], "changed_property_so_makeing_sure_you_are_revived");
      }
      // * now check whether you add this window to the visible windows
      if (!differentWorkspace && !isMinimised) {
        window_coord_t* newAllWindowsVisible = malloc(sizeof(window_coord_t) * (
            ↪ numWindowsVisible + 1));
        // copy old ones
        for (int h = 0; h < numWindowsVisible; h++) {
          newAllWindowsVisible[h] = allWindowsVisible[h];
        }
        // copy new
        window_coord_t newWindow;
        newWindow.x = x_i;
        newWindow.y = y_i;
        newWindow.width = width_i;
        newWindow.height = height_i;
        newAllWindowsVisible[numWindowsVisible] = newWindow;
        free(allWindowsVisible);
        allWindowsVisible = newAllWindowsVisible;
        numWindowsVisible++;
      }
    }
    free(allWindows);
    free(allWindowsVisible);
  }
  XCloseDisplay(dpy);
  return 0;
}
```

Appendix G

Changes for input-driven processes of implementation 3

Here are the changes made to X11 for the input-driven processes. Those lines were added in the file “events.c”, starting on line 4706.

```
ClientPtr client = wClient(pWin);

if (client != NULL && detail != NotifyInferior && detail != NotifyAncestor) {
    unsigned char shouldBeKilled = 0;
    PropertyPtr pPropShouldBeKilled;
    int rc = BadMatch;

    // verify that you have all the parameters to execute request
    rc = dixLookupProperty(&pPropShouldBeKilled, pWin, XA_KILL_NO_FOCUS, client,
        ↪ DixGetPropAccess);
    if (rc == Success) {
        shouldBeKilled = (*((unsigned char*) pPropShouldBeKilled->data));
    }

    if (shouldBeKilled == 1) {

        //system("touch ~/found_a_program_that_wants_to_be_killed_when_focus.txt");
        if (type == FocusIn) {
            reviveIfNotRunning(pWin, client);
        } else if (type == FocusOut) {
            killIfRunning(pWin, client);
        }

    }
}
```

Appendix H

Changes for RTD processes of implementation 3

Here are the changes made to X11 for the RTD processes. Those lines were added in the file “window.c”, starting on line 2197.

```
void reviveIfNotRunning(WindowPtr curr, ClientPtr client) {
    unsigned char isRunning = −1;
    PropertyPtr pPropIsRunning;
    int rc1 = dixLookupProperty(&pPropIsRunning, curr, XA_IS_RUNNING, client,
        ↪ DixGetPropAccess);
    if (rc1 == Success) {
        isRunning = (∗((unsigned char∗) pPropIsRunning−>data));
    }
    if (isRunning == 1 || isRunning == −1) {
        return;
    }
    PropertyPtr pPropPids;
    int rc2 = dixLookupProperty(&pPropPids, curr, XA_PIDS, client, DixGetPropAccess);
    if (rc2 == Success) {
        // send a signal too all children
        uint32_t∗ pids = ((uint32_t∗) pPropPids−>data);
        for (int i = 0; i < pPropPids−>size; i++) {
            char∗ command;
            asprintf(&command, ”touch ~/sendingCONTtopid%d”, pids[i]);
            system(command);
            kill(pids[i], SIGCONT);
        }
        // record the fact that the process is now running
        ∗((unsigned char∗) pPropIsRunning−>data) = 1;
    }
}

void killIfRunning(WindowPtr curr, ClientPtr client) {
    unsigned char isRunning = −1;
    PropertyPtr pPropIsRunning;
    int rc1 = dixLookupProperty(&pPropIsRunning, curr, XA_IS_RUNNING, client,
        ↪ DixGetPropAccess);
    if (rc1 == Success) {
```

```
            isRunning = (*((unsigned char*) pPropIsRunning->data));
        }
        if (isRunning == 0 || isRunning == -1) {
            return;
        }
        PropertyPtr pPropPids;
        int rc2 = dixLookupProperty(&pPropPids, curr, XA_PIDS, client, DixGetPropAccess);
        if (rc2 == Success) {
            // send a signal too all children
            uint32_t* pids = ((uint32_t*) pPropPids->data);
            for (int i = 0; i < pPropPids->size; i++) {
                char* command;
                asprintf(&command, "touch ~/sendingSTOPtopid%d", pids[i]);
                system(command);
                kill(pids[i], SIGSTOP);
            }
            // record the fact that it isn't running anymore
            *((unsigned char*) pPropIsRunning->data) = 0;
        }
}

void killIfAskedVisibility(WindowPtr curr, ClientPtr client) {
    // you only kill this window if it asks to be killed
    unsigned char shouldBeKilled = 0;
    PropertyPtr pPropShouldBeKilled;
    int rc = dixLookupProperty(&pPropShouldBeKilled, curr, XA_KILL_NOT_VISIBLE, client,
        ↪ DixGetPropAccess);
    if (rc == Success) {
        shouldBeKilled = (*((unsigned char*) pPropShouldBeKilled->data));
    }
    if (shouldBeKilled == 1) {
        killIfRunning(curr, client);
    }
}

void recurseSiblings(WindowPtr curr, WindowPtr last, WindowPtr** allVisibleWindowsFound, int*
    ↪ numVisibleWindowsFound, int currDesktop, ClientPtr client) {
    while (curr != last) {
        if (curr->lastChild != NULL) {
            recurseSiblings(curr->lastChild, RealChildHead(curr), allVisibleWindowsFound,
                ↪ numVisibleWindowsFound, currDesktop, client);
        }
        // only include in the array windosws that are mapped, and not minimised, and in current
            ↪ workspace
        if (curr->mapped == 1) {
            // check if the window is in this workspace
            int sameWorkSpace = -1;
            Atom propertyDesktop = MakeAtom("_NET_WM_DESKTOP", sizeof("
                ↪ _NET_WM_DESKTOP") - 1, 0);
            PropertyPtr pPropDesktop;
            int rc = dixLookupProperty(&pPropDesktop, curr, propertyDesktop, client,
                ↪ DixGetPropAccess);
            if (rc == Success) {
                sameWorkSpace = *((uint8_t*) pPropDesktop->data) == currDesktop;
            }
            if (sameWorkSpace == 0) {
```

```
                killIfAskedVisibility(curr, client);
            }
            // the windows is in the current active desktop
            else if (sameWorkSpace == 1) {
                // check if it is minimised
                int isMinimised = 0;
                Atom propertyState = MakeAtom("_NET_WM_STATE", sizeof("_NET_WM_STATE
                    ↪ ") - 1, 0);
                Atom propertyMinimised = MakeAtom("_NET_WM_STATE_HIDDEN", sizeof("
                    ↪ _NET_WM_STATE_HIDDEN") - 1, 0);
                PropertyPtr pPropWinState;
                int rc = dixLookupProperty(&pPropWinState, curr, propertyState, client,
                    ↪ DixGetPropAccess);
                if (rc == Success) {
                    // here I hard coded the fact that the atoms in the list of atoms are all 16 bits
                        ↪ long
                    uint16_t* currAtom = (uint16_t*) pPropWinState->data;
                    while (currAtom < ((uint16_t*) pPropWinState->data) + pPropWinState->
                        ↪ size) {
                        if (*currAtom == propertyMinimised) {
                            isMinimised = 1;
                            break;
                        }
                        currAtom++;
                    }
                }
                if (isMinimised == 1) {
                    killIfAskedVisibility(curr, client);
                } else {
                    // you end up here if the window is mapped, in the current workspace, and not
                        ↪ minimised
                    WindowPtr* newallVisibleWindowsFound = malloc(sizeof(WindowPtr) * (*
                        ↪ numVisibleWindowsFound + 1));
                    for (int i = 0; i < *numVisibleWindowsFound; i++) {
                        newallVisibleWindowsFound[i] = (*allVisibleWindowsFound)[i];
                    }
                    free(*allVisibleWindowsFound);
                    *allVisibleWindowsFound = newallVisibleWindowsFound;
                    (*numVisibleWindowsFound)++;
                }
            }
        }
        curr = curr->prevSib;
    }
}

void killHiddenAndReviveVisible (WindowPtr* allVisibleWindowsFound, int
    ↪ numVisibleWindowsFound, ClientPtr client, WindowPtr root) {
    // find the coordinates of the root window (which is the reference frame for all further comparisons
        ↪ so we gain time here)
    int x_root = root->drawable.x;
    int y_root = root->drawable.y;
    int width_root = root->drawable.width;
    int height_root = root->drawable.height;
    // the first iterms in the list are the lowest windows, the last are the topmost ones
    for (int i = 0; i < numVisibleWindowsFound; i++) {
```

```
        WindowPtr curr = allVisibleWindowsFound[i];
        // find whether the windows wants to be killed if not visible
        unsigned char shouldBeKilled = 0;
        PropertyPtr pPropShouldBeKilled;
        int rc = dixLookupProperty(&pPropShouldBeKilled, curr, XA_KILL_NOT_VISIBLE, client,
            ↪ DixGetPropAccess);
        if (rc == Success) {
            shouldBeKilled = (*((unsigned char*) pPropShouldBeKilled->data));
        }
        if (shouldBeKilled == 1) {
            // get the coordinates that are visible to the user on the given screen
            int x_start_i = curr->drawable.x < x_root ? x_root : ( curr->drawable.x > x_root +
                ↪ width_root ? x_root + width_root : curr->drawable.x);
            int x_end_i = curr->drawable.x + curr->drawable.width > x_root + width_root ? x_root
                ↪ + width_root : (curr->drawable.x + curr->drawable.width < x_root ? x_root :
                ↪ curr->drawable.x + curr->drawable.width);
            int y_start_i = curr->drawable.y < y_root ? y_root : ( curr->drawable.y > y_root +
                ↪ height_root ? y_root + height_root : curr->drawable.y);
            int y_end_i = curr->drawable.y + curr->drawable.height > y_root + height_root ?
                ↪ y_root + height_root : (curr->drawable.y + curr->drawable.height < y_root ?
                ↪ y_root : curr->drawable.y + curr->drawable.height);
            if (x_start_i == x_end_i || y_start_i == y_end_i) {
                // this is the case if the window is too far up or down, hence not visible, or too thin
                killIfRunning(curr, client);
                continue;
            }
            int foundSomeOneOnTop = 0;
            // now check for each window that are on top of you, whether they can decrease the
                ↪ ranges you have
            for (int j=i+1; j < numVisibleWindowsFound; j++) {
                // there is a weird edge case where some window id must refer to a subwindow, and
                    ↪ its immediate parent is hence not to be taken into account
                if (curr->parent == allVisibleWindowsFound[j]) {
                    continue;
                }
                // no need to do the complex clipping here
                int x_start_j = allVisibleWindowsFound[j]->drawable.x;
                int x_end_j = x_start_j + allVisibleWindowsFound[j]->drawable.width;
                int y_start_j = allVisibleWindowsFound[j]->drawable.y;
                int y_end_j = y_start_j + allVisibleWindowsFound[j]->drawable.height;
                if (x_start_i >= x_start_j && x_end_i <= x_end_j && y_start_i >= y_start_j &&
                    ↪ y_end_i <= y_end_j) {
                    foundSomeOneOnTop = 1;
                    killIfRunning(curr, client);
                    break;
                }
            }
            // you didn't find someone on top
            if (foundSomeOneOnTop == 0) {
                // char* command2;
                // asprintf(&command2, "touch ~/revivingWindow%d", numRangesY);
                // system(command2);
                reviveIfNotRunning(curr, client);
            }
        }
    }
```

```
}

void getAllWindowsAndKillTheOnesNotVisible(ClientPtr client) {
    WindowPtr* allVisibleWindowsFound = NULL;
    int numVisibleWindowsFound = 0;
    WindowPtr root = GetCurrentRootWindow(inputInfo.devices);
    if (root == NULL) {
        return;
    }
    // find the current active desktop
    int currDesktop = -1;
    Atom propertyDesktop = MakeAtom("_NET_CURRENT_DESKTOP", sizeof("
        ↪ _NET_CURRENT_DESKTOP") - 1, 0);
    PropertyPtr pPropCurrDesktop;
    int rc = dixLookupProperty(&pPropCurrDesktop, root, propertyDesktop, client,
        ↪ DixGetPropAccess);
    if (rc == Success) {
        currDesktop = *((uint8_t*) pPropCurrDesktop->data);
    }

    recurseSiblings(root->lastChild, RealChildHead(root), &allVisibleWindowsFound, &
        ↪ numVisibleWindowsFound, currDesktop, client);
    killHiddenAndReviveVisible(allVisibleWindowsFound, numVisibleWindowsFound, client, root);
    free(allVisibleWindowsFound);
}
```

Appendix I

Battery consumption results

Here are the results of the battery consumption between the two scenarios. Figure I.1 is when there is no optimisation in place, and figure I.2 is when the modified version of the display server is used.

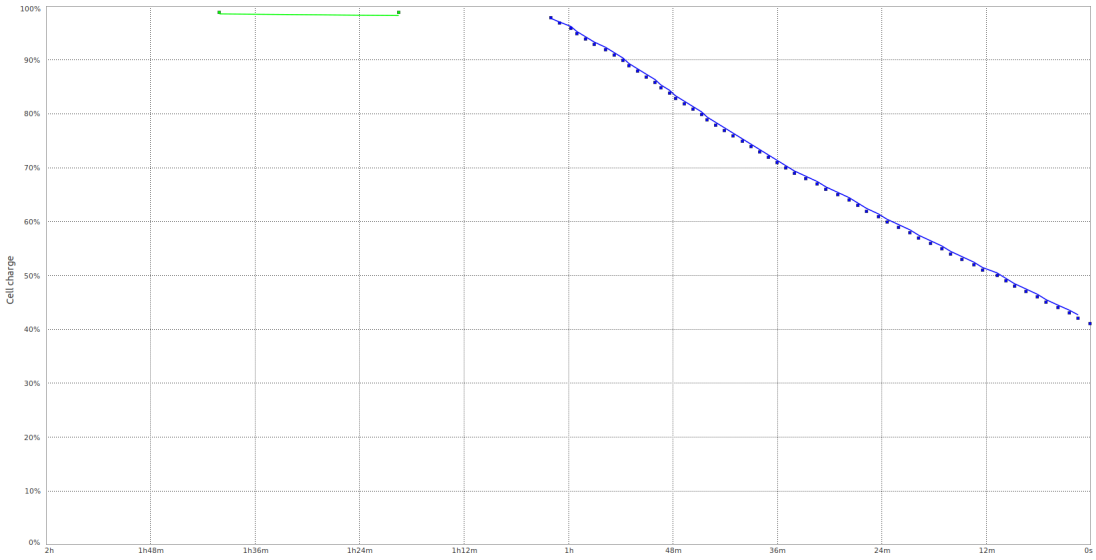

Figure I.1: Battery percentage over time in the normal situation when no optimisation of the visibility is implemented.

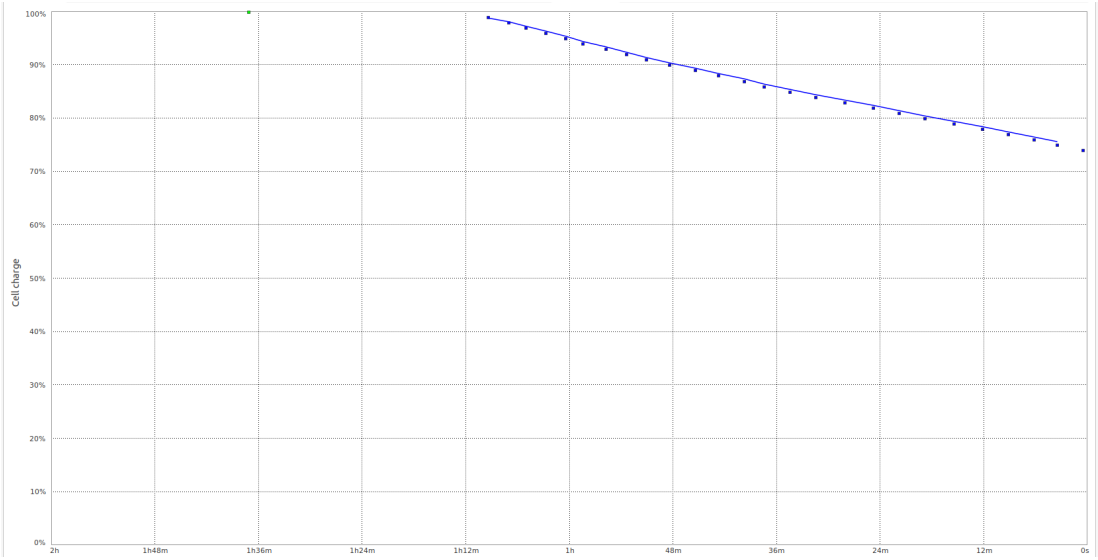

Figure I.2: Battery percentage over time when the modified version of the display server is used.