Freie Universität Berlin

Masterarbeit am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Software Engineering

Design and Implementation of a Prediction Failure Handling Component for a Plan-Based Scheduler

Manuel Leppert

Matrikelnummer: 5177690

manuel.leppert@fu-berlin.de

Betreuer:	Berry Linnert
Eingereicht bei:	Berry Linnert
Zweitgutachter:	Prof. Dr.-Ing. Jochen Schiller

Berlin, September 16, 2021

Abstract

The *Virtual Resource Manager (VRM)* is a Grid management system that aims to bring Service-Level-Agreements (SLA) to the Grid. Clients, as well as service providers could benefit significantly from SLA capabilities, for example by negotiating deadlines as well as penalty fees if that deadline is missed. The VRM architecture utilizes a plan-based scheduling system enabling those capabilities. In order to be able to construct a plan, runtime estimates for jobs are required. Since job behaviour is dynamic and hard to predict, plan deviations of different magnitudes are bound to occur. If the estimation is off by a minor margin, the executing node might choose to accept the deviation and continue running the plan as is. If the deviation is significant, the node has to send a prediction failure signal to the job scheduler. The job scheduler then updates its prediction model and generates a new plan integrating the detected deviation.

The goal of this thesis is to design and implement a scheduling component that determines if a detected plan deviation should be considered a minor deviation or a prediction failure. Additionally it is necessary to define associated behaviour, for example how to assign idle time slots or when to preempt a task. The proposed design is implemented based on an existing prototype that is running as a scheduling policy inside the Linux kernel.

In this thesis, firstly a systematic analysis is conducted of what conditions warrant a prediction failure signal and which parameters influence those conditions. Then a threshold system is developed that incorporates the various conditions into a coherent component that is implemented as an extension of the existing prototype.

The implemented prediction failure handling component is capable of looking at different aspects of the plan and signal a prediction failure if necessary. It also contains self-balancing mechanisms that aim to resolve minor plan deviations locally. The implemented behaviour was developed with the help of a simulation that is intended to give insights into the designed behaviour.

Zusammenfassung

Der *Virtual Resource Manager (VRM)* ist ein Grid-Management-System das darauf abzielt Service-Level-Agreements (SLAs) in einem GRID-Computing Umfeld verfügbar zu machen. Die Möglichkeit SLAs zu formulieren und durchzusetzen kann nützlich für Kunden wie auch Anbieter sein. Um die Umsetzung von SLAs zu erleichtern wird ein planbasiertes Scheduling-Verfahren verwendet, das einige Vorteile gegenüber klassischen warteschlangenbasierten Systemen besitzt. In einem planbasierten Ansatz wird jedes parallele Programm, das von dem System akzeptiert wurde in ein Ausführungsmodell integriert, das auch Plan genannt wird.

Um Pläne zu konstruieren sind Laufzeitabschätzungen notwendig. Da Programmverhalten dynamisch und sehr schwer genau und sicher vorhersagbar ist, kann davon ausgegangen werden, dass Planabweichungen regelmäßig auftreten werden. Nicht alle Planabweichungen haben jedoch die gleiche Signifikanz. Wenn die Laufzeitabschätzungen die reale Laufzeit nur um wenige CPU-Instruktionen verfehlt, dann sollte der ausführende Knoten nach Plan fortfahren. Werden jedoch bestimmte Grenzwerte überschritten, ist es notwendig, dass der Knoten dem Job-Scheduler signalisiert, dass eine größere Abweichung vorliegt. Der Job-Scheduler hat nun die Möglichkeit den Plan an diese aktualisierte Informationslage anzupassen.

Das Ziel dieser Arbeit ist es eine Komponente zu entwerfen und zu entwickeln, die für Planabweichungen entscheiden kann, ob ein Signal an den Job-Scheduler notwendig ist oder nicht. Zusätzlich muss ein Verhalten spezifiziert werden, das vorgibt wie sich der Knoten zu verhalten hat, wenn zb. eine signifikante Zeitüberschreitung aufgetreten ist. Die Implementierung basiert auf einem existierenden Prototypen, der als Scheduling-Policy innerhalb des Linux-Kernels läuft.

Die im Zuge dieser Arbeit implementierte Komponente zur Behandlung von Vorhersagefehlern beachtet verschiedene Aspekte des Plans und sendet ein Vorhersagefehler-Signal falls notwendig. Außerdem implementiert sie verschiedene Verfahren um kleinere Abweichungen selbstständig und lokal auszubalancieren. Das Design und die Implementierung wurden unter zuhilfenahme einer Simulation erarbeitet. Die implementierte Vorhersagefehlerbehandlungskomponente ist als modulare, sich vorhersagbar verhaltende und flexible Komponente geplant und umgesetzt.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

September 16, 2021

Manuel Leppert

Leppert

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Problem statement . . . 3

1.3 Assumptions . . . 4

1.4 Terminology . . . 5

1.5 Requirements . . . 7

2 Plan-Based Scheduling **8**

2.1 Scheduling Problem . . . 8

2.2 Queue-Based Scheduling . . . 10

2.3 Limitations of Queue-Based Systems . . . 11

2.4 Plan-Based Scheduling . . . 11

2.4.1 Basic Concept . . . 11

2.4.2 Challenges for a Plan-Based Scheduler System . . . 12

2.4.3 Plan-Based Node-Scheduler Prototype . . . 13

3 Design of a Prediction Failure Handling Component **14**

3.1 Overview . . . 14

3.1.1 Thresholds . . . 14

3.1.2 Threshold Design Principles . . . 15

3.1.3 Instruction Counting . . . 17

3.2 Local Plan Deviation Handling . . . 19

3.2.1 Discussion t_1 . . . 19

3.2.2 Calculate t_1 . . . 20

3.2.3 Preemption Behaviour . . . 22

3.3 Prediction Failure Handling . . . 26

3.3.1 Prediction Failure Signaling . . . 26

3.3.2 Signaling Conditions . . . 26

3.3.3 Signaling Condition Information Sources . . . 29

3.3.4 Prediction Failure Signal . . . 31

3.3.5 T_2 Considerations . . . 31

3.3.6 Calculating t_2 . . . 32

3.3.7 T_{-2} Considerations . . . 41

3.3.8 Calculating T_{-2} . . . 41

3.3.9 Post-Prediction Failure Node Behaviour . . . 44

4 Implementation **44**

4.1 Layered Implementation Approach . . . 45

4.2 Components . . . 45

4.3 Level One: Simulation . . . 46

4.3.1 Simulation Overview . . . 47

4.3.2 Visualization . . . 51

4.4 Level Two: Userland . . . 51

4.4.1 Overview . . . 51

4.4.2 General Architecture . . . 52

4.4.3 Testing Strategy 55
4.4.4 Level Two to Level Three Migration 56
4.5 Level Three: Kernel 57

5 Evaluation **58**
5.1 Evaluation Functional Requirements 58
5.2 Evaluation Nonfunctional Requirements 60

6 Conclusion **62**
6.1 Summary 62
6.2 Future Work 64
6.2.1 Environment 64
6.2.2 Plan-based Scheduler 65
6.2.3 Prediction Failure Handling 66

A Appendix **68**
A.1 Information Tracking 68
A.2 Compiling and Running the Prediction Failure Handling Component . . 70

Bibliography **73**

List of Figures **76**

List of Tables **76**

1 Introduction

1.1 Motivation

When working on a problem in science or engineering researchers and engineers often reach limits in regards to their data processing capabilities. Consider running a weather simulation in order to predict consequences of climate change. To predict the weather confidently and further into the future, several factors have to be considered. For example, the resolution of the simulation, meaning the cell size of the grid that represents the earth's surface is of relevance. Small but deep valleys for example might average out if the cell size is too big. Still those valleys might reach significantly lower temperatures than the immediate surroundings due to thermodynamic effects. The preserved coldness then could influence the local climate for a longer period of time. A finer resolution would help to better account for those effects, subsequently helping to better understand dynamic aspects of the simulation such as clouds [26]. Another influencing factor on the predictive quality is the amount of parameters that are considered by the simulation. The more relevant parameters, the finer and more an accurate output can be produced. Weather forecasting for example primarily uses atmospheric data points such as temperature, humidity, wind, and more [6], but relevant changes in the atmosphere could also originate from within the ocean. When optimizing the weather model, it would be useful to not only simulate developments above sea level, but also changes in the ocean such as currents or water temperature.
In order to run a simulation with all the relevant parameters in high resolution, a lot of computational power is required. For results to be available in a reasonable time frame supercomputers are increasingly necessary. While high profile applications such as climate simulations require vast resources, a lot of modern applications also profit from the power supercomputers are offering. From training machine learning models on large data sets to running batch jobs for a small business in a friction of the time it would take one off-the-shelf server. A great number of possible use cases for supercomputers in all kinds of different scenarios are imaginable. One major obstacle is the high cost associated with procuring and maintaining supercomputers. While computational power is increasing faster than the price, HPC is still expensive to get into with costs rising especially on the high-end [23]. For most organizations that do not rely on heavy computations as a critical requirement the usage of supercomputers is out of reach.
In the 1990s an idea began to evolve trying to challenge this situation. The concept was called "grid computing" referring to the power grid, where every user can simply plug a device into a power socket and thereby access the utility provided by the grid [9]. The clients benefit from a simple and inexpensive way of getting access to the utility. Suppliers also have an easy way of offering their services with few entry barriers for new competitors. While the chosen metaphor of the Grid might not be applicable in every regard, the general idea has stuck. In order to take this path, grid middleware is required to give clients the capabilities to request resources on the one hand and allow providers to easily offer and allocate their resources on the other hand.
One system that is envisioned to be part of such a grid environment is called the *Virtual Resource Manager (VRM)* [18]. A main objective of the VRM is to bring Service Level Agreements (SLAs) to the Grid. An SLA defines aspects of a provider-client-relationship such as quality, availability, penalty fees, deadlines and so on in concrete terms. Not all

current resource management systems provide the necessary capabilities to enable SLAs in a meaningful way. In this context a resource management system is an interface for accessing the underlying resources. If SLAs would be widely available, the usage of the Grid could become more reliable, flexible and therefore attractive for clients. If a job submission backed by an SLA is accepted by the system, the requirements are made explicit and the system can check whether it can provide the requested service with the given conditions or not. Additionally providers are better able to specify and integrate their resources into a Grid environment. Consider a case where the Grid is running out of GPU resources resulting in high demand for GPU-computations, so data centers might delay currently running jobs and offer their resources to the Grid. Clients could also profit from these dynamics, for example by submitting a job with a deadline far in the future but restrictions on the price. In order to enable these capabilities the *VRM* architecture is relying on a plan-based scheduling approach. Accordingly jobs running on a VRM managed system are not ordered in a queue-based fashion but are integrated into a holistic plan that determines the order of execution in advance.

VRM architecture Figure 1 gives an overview of the general architecture of the VRM.

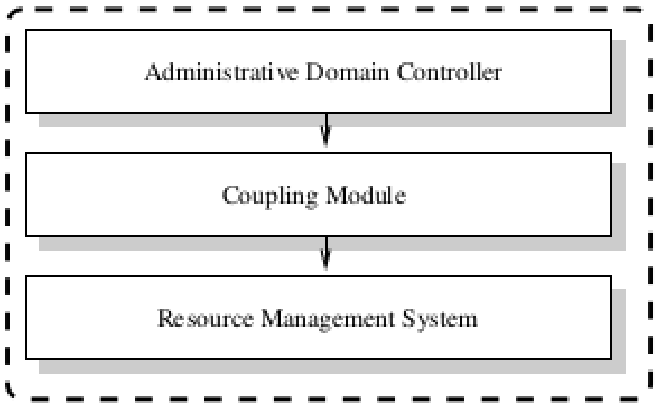

Figure 1: Layers of the VRM architecture, taken from [18]

The VRM consists of a layered architecture that allows for recursive nesting. The objective of the VRM is to give high-level access to virtual resources so that SLA capabilities can be offered. In order to create virtual resources, *Administrative Domains (ADs)* are created. These domains are responsible for combining their underlying resources as well as establishing policies in regard to those resources. *ADs* may be recursively structured. The interface over which the *AD* offers its resources is called *Administrative Domain Controller (ADC)*. The *ADC* of an *AD* also is the unit responsible for handling SLA negotiations for requests regarding its encapsulated resources. For *ADCs* to be able to create virtual resources out of the underlying heterogeneous computing infrastructure, *Coupling Modules (CMs)* are required. The *CMs* act as adapters or brokers that bridge the gap between the *ADC* and the underlying local *Resource Management*

System (RMS). The local *RMS* is the software that manages the actual computational, network, storage, etc. resources. Since the capabilities of different *RMSs* vary, the *CMs* aim to extend capabilities where possible to enable advanced SLA-features.

This thesis is aiming to further explore the plan-based scheduling approach which is particularly suited for enabling SLAs. When a client submits a job the ADC tries to match the job requirements with the capabilities of its underlying system(s). If a match is found and the job is accepted, it is included in the schedule or plan that details the execution model. This execution model lays out how accepted jobs are planned to run conforming to the agreed upon conditions. The execution model determines how the jobs are distributed onto the cluster, which processes run on which nodes and the tasks a process is comprised of. This plan is generated with a prediction model that uses historical data, information from SLA, etc. to approximate the runtime behaviour of the job and its parts. Over time code does change invalidating historical data, input data might influence runtime behaviour in a nonlinear fashion or garbage collection for the executing interpreter might have changed resulting in plan deviations for the job. The generated plan represents a best effort approach that is naturally prone to having flaws. Those deviations however will not be detectable directly by any high level planning component and will first surface on the nodes where the actual computations take place. If tasks are not static and/or involve dynamic inputs, plan deviations are bound to occur. The node has to have a strategy to deal with plan deviations. Since minor deviations can occur for a number of different reasons, the first challenge that the node has to overcome is to decide whether a detected deviation qualifies as a prediction failure or constitutes only a minor haziness. A prediction failure signifies the need for a plan update by the job scheduler. Consider a case where a task is planned to run for 1,000,000,000 instructions, but it takes 1,000,000,100 so the deviation is smaller than $\%0,0001$. It can be argued that this deviation is within the range of what has to be accepted for non-static systems since the deviation is so minor that its consequences are hardly measurable. If the deviation is minor and the node is able to easily deal with it without jeopardizing any other process or the successful execution of the job according to the corresponding SLA, the node might opt to continue with no or minor adjustments as needed.

1.2 Problem statement

An existing prototype that implements the plan-based approach was intended as a proof-of-concept for the feasibility of plan-based scheduling inside the Linux Kernel [10]. It does not contain any form of prediction failure handling. Prediction failure handling is concerned with finding a proportional way of dealing with plan deviations. This thesis is concerned with designing and implementing a prediction failure component that handles plan deviations in a reasonable manner. This primarily entails finding a procedure that allows the node to decide whether a detected plan deviation should be categorized as a prediction failure or not. In order to achieve this, a threshold is developed that will serve as a dividing line. Defining this threshold can be challenging since on one hand it should be predictable so the job scheduler is able to make useful assumptions about the

node's state if it hasn't received any feedback. And also the threshold must be flexible enough to take into consideration the specifics of the current state.
Subsequently system behaviour needs to be specified for different states the node can be in relating to the threshold. For example, if a task is running late, but is still within its minor deviation range, the node does not need to react. If a task is late but its prediction failure threshold won't be reached for quite some time due to the task having a large amount of planned instructions assigned, it could be necessary to implement some sort of preemption to prevent one task's deviation from affecting other processes on the node.
Finally, both the prediction failure detection and prediction failure handling will need to be implemented based one the prototype.

1.3 Assumptions

Since no executable live system is currently easily available, assumptions have to be made in order to establish certain limitations. The assumptions are grouped into three different categories. *Technical assumptions* relate to aspects of the assumed hardware. *Design assumptions* are concerned with the general design ideas and goals. Finally, *environment assumptions* consider the context the plan-based scheduler is envisioned to run in.

- Technical Assumptions

Single Core	: A node on the cluster is assumed to be a single core. The challenge of multithreading will not be addressed within the scope of this thesis.
PMU Capabilities	: The CPU running the plan-based scheduler is assumed to have a way of counting retired instructions. This capability is often provided by a hardware component called *Performance Monitoring Unit (PMU)*. The design and implementation are developed with the assumption that such a component is readily available.
Base OS	: The base operating system on all nodes of the cluster is running a Linux kernel. The implementation is not tailored to a specific kernel version. It is based on the existing prototype of [10].

- Design Assumptions

Resource Sharing	: All processes on a node share the available resources. No process is greedy and the behaviour towards others processes will be viewed cooperatively, e.g. if process A needs less resources than expected and process B more than expected, B consuming some resources that originally have been allocated to A will be allowed within limits. The goal is that no process misses its deadline and not that each individual process finishes as early as possible.
Security	: Security is not regarded an issue. Users' intentions are assumed to be benevolent so no actions are taken to guard against willful disturbances.
Computational Tasks	: Plans contain two types of tasks, computational tasks and communication tasks. This thesis only focuses on computational tasks.

- Environment Assumptions

Time vs Instructions	: It is assumed that there is a conversion rate that allows to transform time units into CPU instructions and CPU instructions into time units. This implies that the CPU is able to retire a constant amount of instructions per time unit, which will also be assumed. This conversion rate is therefore independent of any other influences, such as specific jobs or stages of a job.
Global Time	: There is no global time provided by any higher unit of organization. The only relevant concept of time is provided by the above mentioned instruction rate.
Multiprogramming	: Generally more than one client's job is running on the HPC system at any given time or for the node level: More than one process is assumed to be running on the system at any given point in time.
Plan Deviation	: The plan deviation for tasks follows a normal distribution.
Black Box Rescheduling	: There is a rescheduling component available -often called job scheduler- to which prediction failure signals can be sent and that in return provides the signaling node with an updated plan that is sufficient to deal with the signaled issue.
Multilevel Scheduling	: With systems that allow for recursion, a system might also allow for a recursive job scheduler structure, sometimes called multilevel scheduling. In the context of this thesis, only one level of job scheduling is assumed.

1.4 Terminology

Since the field of Grid computing and plan-based scheduling might not be very widely known and this thesis refers to a developing project that is not fully specified in all regards, a short list of terms that will be used in the thesis is provided below. Most terms will be discussed in more detail at a later point.

- **Cluster:** A cluster (computer) is a parallel computer that is comprised of not strictly defined computation units (which may be highly heterogeneous) that do not use a shared memory space, but are connected by a network for communication [2].
- **Node:** Nodes are the units that form a cluster. In the case of this thesis, a node can more concretely be conceived as one CPU core that is executing the tasks it is assigned to by the plan.
- **Plan:** The plan determines how jobs are mapped onto the cluster. The plan states what processes and ultimately which tasks are to be executed on which node in what order. The plan also contains metadata such as buffer sizes for processes. The plan can be split up and adjusted for each individual node, so the node knows its specific execution plan. This thesis is primarily concerned with the node's plan, which will also be called *scheduling plan*, when a distinction is necessary.
- **Job:** Clients submit parallel programs to the system that are called jobs.
- **Process:** A job consists of a number of processes that are distributed to nodes of the cluster. A process consists of a number of tasks. Every process has a designated start and end task as well as an associated buffer.

- **Task:** A task is the smallest unit of execution in a plan-based scheduling system. Tasks make up processes and tasks start and end with system calls. Contrary to the use in some other contexts, the terms task and process are not being used interchangeably in this thesis. For each task in the plan a number of instructions is given that represents the planned amount of instructions the task is estimated to run before terminating.

- **Plan Units:** Jobs, processes and tasks are entities that are related to within a plan either explicitly or implicitly and therefore are called plan units. A plan can be viewed through each of those plan units and thereby focus on a different level of the execution model.

- **Types of Tasks:** A process consists of communication tasks and computational tasks. Computational tasks represent portions of a process that rely primarily on CPU resources, communication tasks on the other hand rely on IO-based resources such as reading from the hard drive or receiving new data points via network sockets. This thesis focuses on dealing with computational tasks. It is not unreasonable to presume some analogy in the prediction failure handling between the two kinds of tasks, so that the prediction failure handling concepts designed for computational tasks might also be applicable for communication tasks to some degree.

- **Time:** In the following, time is measured and thought of in processor instructions. In the assumptions (1.3) it is stated that we assume a constant conversion between wall clock time units like nanoseconds and CPU instructions. Time can be thought of in conventional units as well, but semantically it is preferable to think of it in CPU instructions instead.

- **Plan Deviation:** A divergence from the amount of instructions that was estimated for a plan unit compared to the instructions that were actually required. Assume a task is planned to need 500 instructions to terminate, but the CPU actually retires 600 instructions until the task terminates. The plan deviation than is 100 instructions.

- **Prediction Failure:** A plan deviation does not equal a prediction failure. A prediction failure is conceptualized as a significant plan deviation. To determine how to quantify "significant" is a key challenge of this thesis and will be discussed in detail.

- **Prediction Failure Signal:** If a prediction failure occurs, a signal is sent by the node to inform the job scheduler that a significant plan deviation has occurred.

- **Rescheduling:** If a prediction failure signal is sent, a rescheduling is triggered. When a rescheduling is triggered, the current plan is updated to reflect a change in the approximated execution model. The new *scheduling plan* for each node is then sent to all relevant nodes.

- **Lateness/Earliness:** Being late means that a plan unit has not terminated in the time it should have according to the plan. Usually this means that the plan estimated the plan unit to finish earlier then it actually did. There are some nuances to this term, since for example a process can be late, while the current task has finished early. So lateness always needs a scope to be understood unambiguously. Earliness is simply the inverse of lateness. Due to this relationship, lateness and earliness can be represented on the same axis. Lateness is represented using positive integers, while earliness extends into the negative dimension.

- **Real/Plan:** The adjectives *plan* and *real* are going to be used to differentiate between the estimation of required instructions in the plan and the actually required amount of instructions during job execution for a plan unit.

- **Job/process/plan-based scheduler**: The above introduced system is conceptualized as having two primary scheduling components. The *job scheduler* or *rescheduling component* that maps jobs onto available resources of the cluster and the *process scheduler* that receives a cropped part of the plan. This node-specific plan also is called *scheduling plan*. It determines what processes are run on a local node and in what order the tasks have to be executed. The *process scheduler* executes those tasks. The *process scheduler* will also be called *plan-based scheduler* in this thesis. Some authors might not consider the outlined scheduler to be only a scheduler but something more instead ([9][p. 28-29]). In the context of this thesis the term scheduler will be used nevertheless.

1.5 Requirements

Based on these assumptions and the general goal of the thesis, the following requirements are set. The requirements are grouped into functional and nonfunctional requirements.

Functional Requirements Functional requirements can be determined firstly by looking at the envisioned environment the plan-based scheduler is designed to run in and secondly by the concrete functionality it has to provide.

REQ-F-0 The prediction failure handling component must provide an interface for interacting with the rescheduling component.

REQ-F-1 The prediction failure handling component must provide the capability to decide whether the current state in terms of plan execution signifies a prediction failure.

REQ-F-2 The prediction failure handling component must provide capabilities for firstly handling minor plan deviations locally and secondly being able to bridge the time between a prediction failure signaling and the arrival of the updated plan.

REQ-F-3 The prediction failure handling component must be able to run inside the Linux kernel.

REQ-F-4 The implementation must be done in a prototypical manner, integrating into a version of the prototype introduced in [10].

Nonfunctional Requirements The nonfunctional requirements are influenced by the not fully implemented environment as well as design goals that can be derived from a not-fully specified and testable environment. The design focus should be put onto making the component as versatile as possible but also implementing a reasonable first attempt at solving the problem.

REQ-NF-0 The prediction failure handling component should be implemented with modularity in mind.

REQ-NF-1 The prediction failure handling component should be configurable to environment details that are not fully known.

REQ-NF-2 If a deadline can be met the prediction failure handling component should ensure that it is met by handling deviations in a reasonable way and not cause significant overhead for the scheduling process while doing so.

REQ-NF-3 The implementation of the prediction failure component should provide some mechanism to easily observe its behaviour.

2 Plan-Based Scheduling

This chapter will give a brief introduction to plan-based scheduling, which this thesis is based on. Firstly, scheduling in general will be discussed. This includes a short peek at HPC scheduling and a short discussion focused on the inherent limitations of queue-based systems. Lastly an introduction of plan-based scheduling as well as a summary of the state of the prototype is given.

2.1 Scheduling Problem

A scheduler is a component that allocates resources to competing entities on a system that shares those resources. Usually the resource in question is the CPU. In regards to CPU scheduling the scheduler has to decide when and for how long a process is allowed to use the CPU. When thinking about allocating computational time to processes a couple of potential goals the scheduler might want to achieve come to mind:

- Distribute resources fairly to all contenders
- Ensure that starvation does not occur
- Establish priority so that more important processes are chosen over less prioritized ones
- Optimize system throughput
- Ensure that no entity monopolizes the resource
- etc., see for example [3, p. 44]

For traditional queue-based scheduling systems a number of widely known algorithms such as first-come-first-serve, priority-scheduling or round-robin scheduling ([27, p. 207-216]) exist. When delving deeper into the subject more specific scheduling problems arise. Multicore scheduling for example deals explicitly with assigning processes to multiple cores ([27, p. 219-223]) or specific runtime restrictions are posed as it is in the case with real-time scheduling ([27, p. 223-230]).

HPC scheduling Another instance of a more specific scheduling problem arises in High Performance Computing (HPC). Importantly, the target entities that are subject to being scheduled in this context are not exclusively processes, but also jobs at a higher level of abstraction. The scheduling of jobs is distinct to the scheduling of processes/threads. The main question posed by job scheduling is: *How can jobs be assigned to the HPC system in the most effective and efficient way (according to some goals)?*. This process involves several steps. These can include for example quantitative partitioning, which decides how many processes a parallel program receives or qualitative partitioning, which defines more specifically which nodes of a cluster are assigned to the program. HPC scheduling in abstract terms tries to find a mapping of programs to the corresponding cluster computer.

[24] describes in some detail popular solutions to the job scheduling problem. One way to find a suitable mapping is by applying a variable partitioning scheme. This means that analogous to the term in operating system's memory allocation, a job receives a partition of the cluster for the time it is estimated to run.

Figure 2: Variable partitioning problem

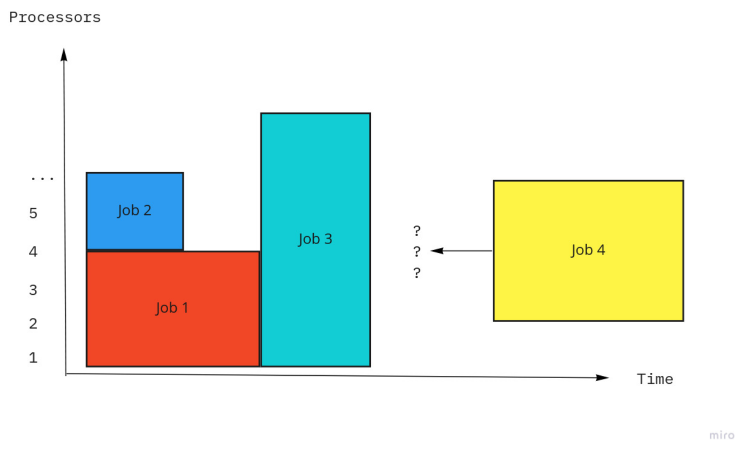

Figure 2 illustrates what the variable partitioning approach revolves around. The x-axis is the time axis indicating when a job is active on the system. The y-axis shows all the computational units of the system. Jobs have to be scheduled in such a way that their resource requirements are satisfied. Job 1 for example is probably running for a longer period of time then job 2 or 3. Job 3 is expected to require more computation units then the other jobs. The jobs 1,2 and 3 are already scheduled to run on the cluster for some amount of time. Now job 4 is queued and has to be scheduled to run.

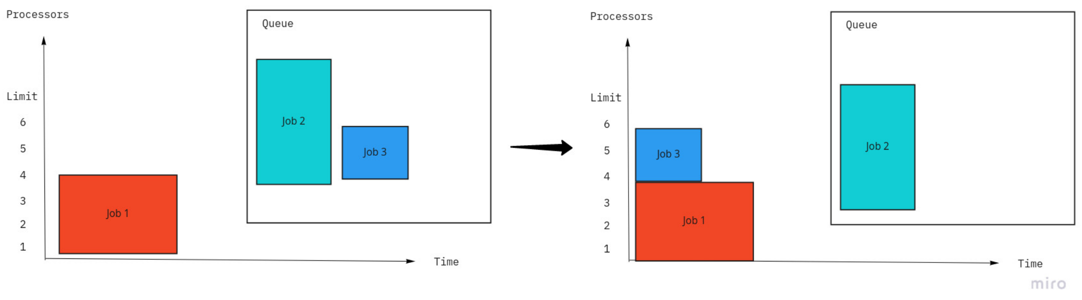

Figure 3: Backfilling

So HPC scheduling systems consist of at least two distinct schedulers. Firstly, a higher level scheduler that receives requests and distributes accepted requests to available nodes. This scheduling component will be called *job scheduler*. In order to execute actual instructions and their associated tasks/processes according to the schedule of the *job scheduler*, there needs to be a *process scheduler* that is running on the nodes. The scheduling system can be for example *queue-based* or *plan-based*. In the following paragraphs, a brief overview of queue-based systems is given in order to contrast the differences in concept.

2.2 Queue-Based Scheduling

Queue-based systems rely on a queue to determine execution order. The first process/-task/job in the queue is the next to be executed. The scheduler's task is to order the queue in a manner that is compatible with its goals. Queue-based systems can be improved and extended upon, for example by introducing dynamic prioritization to more efficiently achieve the scheduling goals. In principle queue-based systems manage a queue data-structure in a certain manner. In a HPC environment for example where achieving a high system utilization is a priority, some additional algorithms can be applied when trying to assign jobs to a cluster. If the queue-head can not be run because the currently available resources can not satisfy the requirements, but the queue contains other jobs that are able to run with the available resources *backfilling* can be applied. Consider the example illustrated in 3.

Job 2 in the queue-head can not be set running, because there are not enough resources available on the system, but at the second place in the queue a job is waiting for which the available resources would be satisfying. So backfilling can be applied to improve system utilization.

Backfilling improves on a simple first-come-first-serve strategy by not only statically moving the planning horizon forwards, but also looking backwards in the queue to fit smaller jobs into gaps that can not be filled currently. [24] provides some additional details on backfilling and its different variants. The topic of queue-based scheduling is a research area on its own and could be discussed in much more detail (also see [4]), but since it is not the focus of this thesis, it will not be discussed any further.

2.3 Limitations of Queue-Based Systems

While [35] states that in 2016 "resource management systems schedule jobs in a queuing fashion", those systems have certain limitations that will be discussed next. Even with added measures to queue-based scheduling systems, the quality of the scheduling outcome is dependent on the order in which jobs are submitted. The jobs earlier in the queue are preferred disregarding additional information that might be available. Also some advanced scheduling requirements can not be solved easily with a queue-based system. In [32] the authors for example describe the problem of *co-allocation.* If the resource needs of a job can not be satisfied by the cluster the job was initially submitted to but other connected clusters are potentially available and the combined resources would fulfill the requirements, it would be useful to merge the resources and then run the job on this newly composited system. The challenge for a queue-based system is that more detailed, additional constraints such as this simultaneous execution of a job on several clusters are not directly supported. The job could be split and sent to the connected clusters, but those job fragments would need to be inserted in the corresponding queues. In systems of this kind, there is no obvious way of enforcing a simultaneous execution required for *co-allocation.* With support for additional constraints, SLAs or SLA-like agreements could be made to further adapt and optimize the execution flow and thereby improve the utilization of HPC systems. Other examples for advanced requirements hard to satisfy with queue-based systems are:

- A job has stages of its execution where some processes need to run on specific nodes with a specific licensed software installed.
- The client has hard real time-like requirements for different stages, where the job has to be finished at a certain wall clock time or the results are useless.
- A client has to save money while not prioritizing an immediate execution, so the job could be scheduled in fragments when parts of the system are planned to be idle.

While it is possible to devise mechanisms to overcome such challenges in a queue-based system, it also is discernible that the ad-hoc queue-based system might not be best suited for those challenges. Plan-based systems might be more adapted to tackle challenges of these kind. In the following paragraphs the plan-based approach will be introduced.

2.4 Plan-Based Scheduling

Firstly the general concept of plan-based scheduling is explained including a short discussion of related challenges. Afterwards, a more detailed view on the current state of the prototype will be given.

2.4.1 Basic Concept

Opposed to queue-based systems, plan-based systems do not use queues in which jobs wait until their required resources are available. Plan-based systems are able to have a wider temporal awareness compared to queue-based systems because they are required

	queuing systems	**planning systems**
planned time frame	present	present and future
submission of resource requests	insert in queues	replanning
assignment of proposed start time	no	all requests
run estimates	not necessary (exception: backfilling)	mandatory
reservations	not possible	yes, trivial
backfilling	optional	yes, implicit

Table 1: Comparing queue-based systems with plan-based systems

to have one additional piece of information. Plan-based systems rely on having runtime estimations available to try to construct a more optimized plan. Therefore it is a requirement to have an estimation of the duration of the job and its parts to apply plan-based scheduling. This additional information is the crucial advantage the plan-based system has over a queue-based system in terms of the challenges mentioned above. Whenever a new job is submitted to the system, constraint checks can more easily confirm that given restrictions regarding execution time, deadline, co-allocation and so forth hold. If passed, the new job is included in the existing plan and the nodes that are participating in the execution of the new job receive updated scheduling plans accordingly. This approach allows for some additional features that would hardly be possible with a strict queue-based system. In [11] the authors provide a short table with comparisons. This comparison is shown in table 1.

As stated before, queue-based systems can be enhanced to implement more advanced features, but in a plan-based system they come more naturally. The principal difference remains that in plan-based systems additional runtime information can be used to calculate ahead and construct a plan that lays out the execution model more holistically and less ad-hoc.

2.4.2 Challenges for a Plan-Based Scheduler System

Plan-based scheduling also poses some additional challenges that are avoided by queuing systems. As [11] mentions, some non-technical challenges might become more relevant. For example in terms of communication, clients might be more inclined to question concrete start- and end-times of a job compared to a queuing system, where those are not as clearly defined. More relevant in this context though are technical challenges. The quality of the execution model highly depends on factors that pose non-trivial engineering tasks. Firstly the collection and management of data regarding runtime information and the resulting prediction generation is demanding. Even if a job's source code has not been modified, variations in input data might lead to unpredictable increases or decreases in runtime. Depending on the nature of the parallel program, even slight changes might turn out to have large impacts. Consider a weather simulation, where a variation in a fraction of a degree in ocean warmth may decide if a hurricane forms or not. If so, the system is pushed into a much more volatile state. Then, a far more computational intensive execution might follow, increasing the runtime significantly. Secondly, plan generation is an instance of the *Job-Shop-Scheduling Problem*, which is in the class of NP-hard problems [28]. Therefore even with sound runtime estimation data, plan generation is in principle a costly task becoming more and more challenging with the addition of extra constraints. Furthermore, a more complex software stack is required to run a plan-based scheduler, since the system has to continuously update and enforce the plan on all participating components. A plan-based scheduler also needs to concern itself with problems such as dealt with in this thesis, where a plan turns out to not be accurate to a sufficient degree. Summed up, a plan-based scheduling approach promises a whole set of new capabilities that would be very hard or impossible to achieve with a queue-based approach at the cost of also increasing complexity.

2.4.3 Plan-Based Node-Scheduler Prototype

As hinted to before, for a plan-based system to work, two scheduling components are required. The job scheduler that generates an execution model/plan and a local component running on the nodes of the system that enforces the execution of processes/tasks according to the plan. One such local component that enforces the plan execution is the prototype developed in [10]. The implementation of this thesis is based on that prototype.

Objective Since most HPC systems use a form of Linux as their base operating system (in the Top500, currently 100% of operating systems are of the Linux family [30]), the prototype was implemented for the Linux kernel. The objective was to test the feasibility of integrating a plan-based scheduler into the Linux kernel that enables the execution of scheduling plans. The main challenge was to harmonize this scheduler with the default Linux scheduler. While the plan-based scheduler is designated to execute tasks according to the plan, all other usual operating system functions are not envisioned to be part of the plan. For example, typical OS duties such as logging or memory clean-ups should be done by whatever other scheduler is the default on the system for such tasks. So since their execution is not included in the plan, it is important to guarantee that their execution still happens in appropriate time frames so that the operating system can run stable.

Implementation Linux conveniently provides a framework for adding additional scheduler functionality in the form of scheduling policies. Those policies are ordered in a list, where the list index indicates the corresponding priority of the scheduling policy. The plan-based prototype is assigned the highest priority of all scheduling policies and therefore plan execution is prioritized. To achieve the above mentioned harmonization, the prototype incorporates a mechanism that allows enabling and disabling itself. In the prototype, this mechanism is time-based. After some amount of time has passed, the plan-based scheduler disables itself. The time periods where the plan-based scheduler is enabled are called *execution time*, when the default scheduler is active, the periods are called *unallocated time*. Via empirical study, it was shown that this approach was feasible. The process descriptor for Linux, the `task_struct` contains information that determines which scheduling policy is responsible for its scheduling. This makes it very easy to assign tasks of the plan to the plan-based-scheduler. Combining those mechanisms, the author was able to implement a scheduling policy suitable for a plan-based scheduling system.

3 Design of a Prediction Failure Handling Component

The goal of this chapter is to combine several aspects a prediction failure handling component requires into one coherent description of a proposed design. Firstly, a general overview and design goals are given in 3.1. This section will also discuss the technical issue of instruction counting capabilities and general design considerations that are prerequisite. It follows 3.2, which deals with pre-prediction failure signaling behaviour. The section covers plan deviations that are so far not deemed to be prediction failures and mechanism to ensure that one task's plan deviation does not significantly affect other plan units. Finally, 3.3 is concerned with prediction failures that can not be handled locally anymore and need to be delegated to a higher level scheduler.

3.1 Overview

This section aims at giving a conceptual understanding of the different types of thresholds and prediction failure handling behaviour. Firstly, an overview of the thresholds and their relationship to each other is given in 3.1.1. There will be a short discussion on the general paradigms in regards to the design of the thresholds in 3.1.2. Concluding this general discussion technical details that impact the design and implementation of those thresholds will be provided in section 3.1.3.

3.1.1 Thresholds

The prediction failure handling component monitors 3 distinct types of thresholds and initiates appropriate action if they are reached.

The thresholds t_1 and t_2 are concerned with plan units that exceed their planned instructions and thereby turn late. T_{-2} is conceptually the inverse of t_2, governing plan units finishing significantly earlier than expected. The following enumeration describes the high-level goal for each type of threshold.

t_1 : If the lateness of a task reaches t_1, the running task will be preempted and the next task of the plan will be scheduled. The plan-based scheduler will try to find a suitable slot for it so it can be run again later. The goal of t_1 is to detect minor deviations and trigger local handling, so that one task's plan divergence does not interfere significantly with other processes.

t_2 : This threshold marks the step from tolerable to intolerable plan deviation. When a plan unit crosses this threshold, the transgression is deemed too significant for the node to handle. A prediction-failure signal is sent in order to trigger a rescheduling of the plan which incorporates the detected deviation. T_2 is composed of several components. Most notably, t_2 consists of t_2*_task*, t_2*_process* and t_2*_node*. Each component tracks the lateness of a different plan unit. Additionally, other components may be added to include further conditions. T_2*_preemptions* for example tracks how often a task was preempted to limit the amount of preemptions before triggering a reschedule.

t_{-2} : As mentioned above, t_{-2} is conceptually the inverse of t_2, meaning this threshold deals with significantly early terminating plan units. T_{-2} has some characteristics that make it harder to find suitable components, which will be discussed later.

These 3 threshold types form the 2-tier escalation model illustrated in 4. The further an execution deviates from the plan, the more likely an escalation becomes. Since t_1 deals with the preemption of tasks, there is no analogous t_{-1} threshold. The goal of those thresholds combined is to have a way of keeping the actual execution close to what the plan determines.

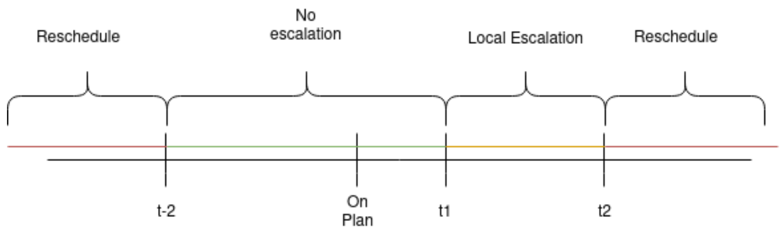

Figure 4: Escalation model

3.1.2 Threshold Design Principles

The component to be designed is intended to be embedded into an environment that is not fully implemented or strictly defined, therefore decisions will have to lean on pre-defined values that try to anticipate this future environment. It needs to be weighted for example, how flexibility is valued compared to predictability or whether additional resources should be allocated using a judgment of fairness or a judgment of neediness. This section lays out a conceptual vision, discusses preferences and develops assessment criteria when goals are in conflict and no solution is obviously most appropriate.

Scope of Plan-Based Scheduler The plan-based process scheduler is the center point in this thesis. Conceptually it is the most low-level entity and also the interface to the actual computational resources the node provides. The node itself is not envisioned as an explicit decision-making component of a distributed scheduler, but instead as an execution unit that passes more intricate scheduling conflicts back to the job scheduler, which has a global view on the system state. Impactful and significant scheduling decisions should therefore be delegated if they arise. Ad-hoc scheduling decisions, which have a need for fast responses will have to be made on the node. This responses however will not attempt to provide efficient local rescheduling.

Depth and Resolution of Prediction Failure Prevention In the listing of the different state keeping variables (at A.1), some amount of configuration and tracking variables are presented, so the prediction failure component has a reasonable base for making decisions. By combining these tracking variables using statistical and stochastic methods, increasingly intricate indicators could be derived, which in the most extreme case could lead to the prediction failure handling component having its own sophisticated prediction model. With the data that is potentially available to the plan-based scheduler, development should be very aware of the risk of feature-creep. As mentioned above, t_1 is the threshold that triggers a task preemption if it is transgressed. T_1 could simply be calculated by adding an amount of instructions to the instructions the plan has estimated for a task. This amount could be static or dynamic, factoring in the current node state, for example, the node state could be factored in by simply letting t_1 scale according to the current system load. Alternatively, the node state could also be factored in by extrapolating the system load based on periodically taken samples. The sampling could of course be improved continuously: The extrapolation algorithm could be improved, the sampling could be done not on a global load state, but could be done on a per-process level, different samplings could be combined, future task execution could be estimated and so on. This would potentially allow for a more and more refined prediction failure handling component. By tracking the progress of a plan in every detail and then feeding those details back into corresponding extrapolation algorithms, a very fine-tuned behaviour could be achieved. Since there is no clear limitation rooted in a fixed functional requirement or a tightly defined environment, limits have to be self imposed. Since the environment and circumstances under which the plan-based scheduler should operate are not tightly defined, in the context of this thesis a solution that fits the general case is the goal. The solution should strive to be modular and flexible so that more intricate solutions can easily be plugged in if required.

Predictability vs Optimization During the development of this thesis system predictability and optimization turned out to be opposing goals. A optimized prediction failure handling component for example can be more successful in detecting prediction failures early, signaling them or even handle them locally. On the other hand, the more details are included in the design, the less predictable the behaviour becomes. With a very low level of resolution t_1, t_2 and t_{-2} might be implemented as fixed values. This is problematic since the accumulated effects of a number of deviated tasks also have adverse effects. With more thought put into unwanted circumstances, more conditions have to be incorporated into thresholds. For example, if a process just started and one of its task is late, then this might not be cause for concern, but if a task is close to its processes deadline, it might become a concern. This could lead to a nesting of thresholds and to less predictability. Based on the conceptual understanding of the node, design decisions should favor *system predictability over a high level of detail* in conflicting cases. The argument being that the job scheduler should handle edge cases not the node. If the node shows a predictable behaviour, then the job scheduler should be able to generate plans that do not cause unwanted behaviour. It is challenging to generate fitting updated plans if the node behaviour is hard to reason about.

Coding Style In terms of implementation style, a readable and clear implementation should be preferred over a runtime optimized solution. In the context of an HPC system, efficiency should always be a factor, but even if the implementation may favor readability and understanding over optimizations for speed, components should generally be designed in a way that would require no heavy computations. One main requirement is to keep deadlines, which should always be considered when designing and implementing the component.

3.1.3 Instruction Counting

The plan-based scheduler envisioned in this thesis heavily relies on CPU instructions as a concept of time, so some thought should be put into the topic. CPUs with a CISC architecture might for example retire fewer instructions per cycle than those based on a RISC architecture. This is due to the historical need for smaller programs in environments with limited memory. CPU vendors therefore would combine frequently used sequences of instructions into single instruction. Simple instructions therefore might take only a single clock cycle, while more complicated once might take multiple. Referring back to the assumptions made in 1.3, it is assumed that the CPU runs with a constant rate of instructions per time unit. No further specification regarding the nature of those instructions is made. While plans are constructed with a fixed rate of instructions per time unit in mind, the node requires a facility for a precise measurement of retired instructions. In the following paragraphs ways of counting instructions and ways of how to utilize those counted instructions to enforce threshold compliance are discussed.

Counting instructions

Measure wall clock time At the most basic level instruction counting, or in this case instruction estimation, can be done by utilizing the system clock and a fixed conversion rate of time units to instructions. This is the approach applied by GNU's *time* program ([19]) that refers to kernel data structures that are updated when system calls such as *wait* are made. This approach uses hardware features like *High Precision Event Timer*[14] in a stopwatch manner. The advantage of this approach is its nearly universal applicability, the major drawback is the inherent unreliability and potentially low resolution. Assuming a fixed conversion rate of *time* $\rightarrow$ *instructions retired*, firstly the precision of the clock has to be sufficiently high and secondly, the timestamps used to calculate the retired instructions have to be clearly attributable to the task or process in question.

PMU For a more precise account of retired instructions some processors (for example by Intel, ARM and SPARC [1, 25, 34]) have a designated hardware component that specifically measures performance aspects. This unit is called *Performance Monitoring Unit (PMU)* and it contains a set of special registers for tracking performance related events such as cache misses, page misses or instructions retired in a given CPU mode on a given core, etc. Depending on the architecture there are some registers that track predefined events and some programmable registers that can be configured such as the Intel Performance Event Select Registers ([12]) to track different kinds of events. One advantage of this approach is that the performance measurement is implemented in hardware, so little overhead is added.

Using the PMU For the targeted Linux platform several profiling tools such as *OProfile* and *perf* that utilize existing kernel space interfaces to PMU registers are available. *Perf* for example is a tool that is able to track such performance statistics [7], using different sources of information such as pure kernel counters (existing solely in software) or PMU registers provided by hardware. As such it could be used as a provider for the instruction counter information that is required by the plan-based scheduler. Since *perf* is a userspace program that is not designed to run inside the kernel, the kernel interface has to be called upon directly. The data structure encapsulating the access to various performance events is called `perf_event`. The relevant `perf_event` for retired instructions is called `PERF_COUNT_HW_INSTRUCTIONS`([33]). Through the usage of the `perf_event_open` system call, a file descriptor can be retrieved which can be queried by using parameters (such as PID, CPU, type of event,...) of the system call. Since the instruction counter is a so-called hardware-counter it is not guaranteed to be available on every platform. Notably the support for PMUs on virtual machines seems to be the exception rather than the rule. A PMU is not provided for example by VirtualBox and Red Hat's KVM also only supports a Virtual Performance Monitoring Unit (VMPU) on some Intel machines ([13]). While generally more reliable, even instruction counting with PMU support might be subject to incorrect reporting. Due to the use of branch prediction, branches might get speculatively executed that ultimately are not selected, while still contributing to the count of instructions. Yet, in terms of reliability and precision, if a PMU is available, this instruction counting approach should be preferred.

Thresholds, Instruction Counting and Interrupts If the plan-based scheduler is called, it has to read the retired instructions from the corresponding interface and update a data structure representing the task that was run previously. This data structure provides a field that allows tracking the sum of the retired instructions per task. This data structure might look like listing 1.

Listing 1: Struct Task

```
struct task{
    // --- attribution variables --
    unsigned long task_id;
    unsigned int process_id;
    // --- tracking variables ---
    unsigned long instructions_planned;
    unsigned long instructions_run;
    char state;
};
```

The challenge when dealing with lateness-related thresholds is that the scheduler in the general case can only detect a threshold crossing after it has occurred. The scheduler gains control when a timer interrupt is triggered or when a task has finished. In the case of late tasks, the scheduler and therefore the prediction failure handling component does not immediately register when a transgression occurred. Depending on hardware support, an option is available to detect those transgressions almost immediately.

Overflow events On some architectures an overflow event can be set up that triggers as soon as a defined value is reached in a certain *PMU* register. This event can be associated with a handler routine. The handler could call the prediction failure handling component. The prediction failure component then updates its tracking data. It could then immediately detect that a transgression occurred. The overflow event could be set to trigger for example when t_1 is transgressed, to gain the capability of precise preemptions. While useful, this feature does not seem to be widely available, in the following, it therefore is not assumed available.

Instruction counting resolution If a CPU with 4 GHz clock speed is assumed as well as the current default configuration of 250 Hz for timer interrupts on an Ubuntu 20.04 system with kernel version 5.8.0-45, a timer interrupt is triggered about every 1,600,000 CPU cycles. Depending on the requirements for precision, this might be precise enough or not. On systems that do not allow for overflow events, a PMU based counting or an estimation using a system timer are the most straightforward replacements. One consideration that should be mentioned is that the Linux kernel timer interrupt frequency can be defined in the kernel configuration file for compiling. Depending on the hardware, frequencies of up to 1000Hz are available. Since more frequent timer interrupts also decrease throughput, this trade-off needs to be discussed especially in the context of an HPC system. This aspect might be a further question of interest and is mentioned in the future works section 6.2.

3.2 Local Plan Deviation Handling

As mentioned in the overview, t_1 is concerned with limiting the possible deviations a single task may add. This chapter will discuss the reasoning for introducing t_1, then an algorithm to set the threshold value of t_1 will be proposed. Lastly, some behavioral details will have to be considered.

3.2.1 Discussion t_1

In the context of this thesis preempting running tasks plays an important role as the first and most readily available method the plan-based scheduler has to contain deviations. On the one hand, preempting entails overhead, since more context switches occur. Therefore the execution time of the plan-based scheduler is more occupied with preparation work compared to running a task until it either finishes or a prediction failure signal is sent. In circumstances that guarantee that transgressions lie within a defined range, letting tasks finish even if they are running late, could be the best choice. In the context of this thesis, where such behaviour is not assumed, tasks could theoretically exceed their planned number of instructions by any amount. Worse even, with the possibility of non-terminating tasks some form of preemption has to be implemented to prevent the possibility of other processes starving. An argument can also be made that in order to maintain an execution as close to the plan as possible, it would be beneficial to have some preemption mechanism to ensure at least a basic form of similarity. While preemption does not guarantee that the exact order of execution is followed, it ensures that at least at the process level this similarity is maintained. Without preemption stretching effects could also cause more severe divergences. Consequently some form of preemption has to be established. T_1 marks the first lateness threshold on the task level. The exact details of how to calculate such a threshold will follow in the next paragraphs.

3.2.2 Calculate t_1

So the first-tier threshold t_1 is concerned with handling tasks that are exceeding their number of planned instructions. With the goals described in 3.1.2 in mind, t_1 is created to ensure basic system stability and predictability. System stability means that rogue tasks should not affect other tasks or processes running on the same node significantly. Predictability in this context means that the real execution of tasks on the node is similar to the execution model represented by the plan and aberrations only occur in a previously defined and limited manner.

When looking for indicators that help pin down the concrete value for t_1, the question arises: What conditions should cause t_1 to trigger and therefore need to be considered in its definition? It should be noted that t_1 should be thought of as *task preemption* and not process preemption, therefore this threshold should consider task details in its determination, not process or node states. Also, task preemption needs to happen when a task is taking some amount of instructions more than it was originally planned for, therefore is running "late". This is achieved by setting t_1 to a value calculated with the task's planned instructions and a σ-multiplier that can be understood as the border between an acceptable and problematic plan deviation. The challenge is to define a suitable value for σ_{t1} or derive σ_{t1} from the current state. Referring back to the design decisions in 3.1.2, σ_{t1} is defined as a configurable constant that does not need to be determined in an additional procedure. Naturally σ_{t1}'s value must be greater than 1. So when σ_{t1} is determined, the next step in calculating t_1 is simply using the planned task length as a basis for σ_{t1} to multiply with:

$$\boxed{t_1\_relative = length\_task\_plan * \sigma_{t1}}$$

$t_1\_relative$ incorporate the idea of an acceptable deviation, but since tasks may vary significantly in their length, it seems advisable to additionally provide a hard boundary that guarantees that independent of specifics the deviation stays within a predictable range. Consider the case, with an average task length of 100 time units, and σ_{t1} set to 1.05, the acceptable deviation would be 5 time units. Now with these settings a task is scheduled that is estimated to run 10,000 time units, resulting in an acceptable deviation of 500 time units. Accordingly to prevent edge cases like this from steering the system significantly of its planned path a safety layer has to be defined. This is achieved by introducing a hard cap that is defined in the constant `PREEMPTION_LIMIT`. This cap is required since no assumptions are made for a minimum or maximum task length and σ_{t1} is defined to be a constant. To incorporate this safety net into t_1, the following adjustment is made:

$$\boxed{t_1\_max = min(t_1\_relative, lenght\_task\_plan + PREEMPTION\_LIMIT)}$$

This upper boundary covers one end of the spectrum, but only applying a σ_{t1} value to the planned task instructions could cause issues with very short tasks. Consider the case, where one task is planned to run in less than one timer tick. Now for a small number of planned instructions, the allocation of extra instructions provided by σ_{t1} would mean that t_1 allows only insignificantly more than planned instructions. It would possibly even lie within the same timer tick range. Imagine a slight change of input data, so that a task that used to run 1 timer tick before now requires three or four timer ticks. It could be argued that for the sake of a predictable system behaviour in those cases, some leeway should be given so that in the described scenario the task would be only interrupted once instead of three or four times. As it will be later discussed in 3.2.3, preempting a task repeatedly entails its own challenges. By granting tasks a minimum of guaranteed preemption-free execution time this effect can be prevented. This lower bound is represented by the constant `NO_PREEMPTION`. On a node with a huge number of scheduled tasks `NO_PREEMPTION` is a sensitive value, due to its potential stacking effects and therefore must be handled with care. Accordingly the final step in determining t_1 then is to ensure this lower bound is kept:

$$\boxed{t_1 = max(t_1\_max, lenght\_task\_plan + NO\_PREEMPTION)}$$

As with σ_{t1} an argument could be made for letting the above mentioned capping constants be defined as dynamic variables instead. Consider two states of a node, one being already behind significantly, one on time or even early. If a node is already very late, resulting in the overall shifting backwards of tasks, it could be argued that preemption should also be more resolute than on a node, where tasks may start early. Since on a late node in average several processes transgress their planned instructions, the node should be more sensitive and eager to limit those latenesses as generally discussed in 3.1.2. One argument against this approach could be that more preemptions also result in more context switches and therefore less time the node spends on the actual execution of tasks, which would then further add to the problem. There are arguments to be made for allowing this kind of behaviour, but it would require a much more delicate procedure for decision making. Furthermore the problem of accumulated lateness is nearly impossible to solve by the node itself and should be a consideration for the t_2 threshold that escalates problems to higher instances. The possible relaxation of t_1 on early nodes could be discussed as an optimization attempt and t_1 might be a feasible starting point when it comes to runtime optimization for the plan-based scheduler.

Check t1 Finally the plan-based scheduler compares t_1 against `retired_instructions_task` to check if the current task should be preempted, an easy comparison that determines if a preemption should be triggered or not can be done:

```
char preempt = retired_instructions_task < t1 ? 0 : 1;
```

Since preempting has some additional consequences, those intricacies are discussed in more detail in the following section 3.2.3.

3.2.3 Preemption Behaviour

When the plan-based scheduler detects that a task has reached its t_1, it preempts the task and attempts to move it to a suitable slot, which could be designated either to a task belonging to the same process or is not allocated to any process (idle time). The search for an appropriate slot is fairly simple, but when thinking about the details, some challenges arise. The following paragraphs will try to dissect what those challenges are and how to reasonably handle them. Firstly, it will be discussed which mechanism determines the next slot for a preempted task. Secondly, side effects that can occur when a task is preempted multiple times will be discussed.

Next slot of process If a task is preempted the plan-based scheduler can look for another slot belonging to the same process as the preempted task. The preempted task can then be inserted before the next task of its process since the plan mandates a strict ordering of task execution. Meaning if $task_n$ of a process is preempted then it is assigned to the slot of $task_{n+1}$ of that process. $Task_n$ will therefore be scheduled to start when $task_{n+1}$ is planned start. Notably, with this approach, a new distinction has to be made. The distinction between slots and tasks. Tasks are placed in slots, but tasks can be moved into other slots through preemptions. So slots can be considered unmovable entities that determine the basic structure of the plan. With this in mind, consider the following scenario: An exceedingly long running $task_n$ with 1000 time units allocated to it is running, but even more time units, say 1100 time units are required for the task to terminate. $Task_n$ crosses its threshold t_1 and after running 1020 time units is preempted. $Task_n$ is pushed to the slot of $task_{n+1}$, which was planned to run 5 time slots. $Task_n$ will transgress t_2_task, where a prediction failure will be sent not before 1200 time slots. This means that the slot allocated to $task_{n+1}$ will not be sufficient for $task_n$ to finish. This example illustrates that there is further need for designing a behaviour to handle this kind of problem. In a real world scenario this kind of constellation where a long task is followed by several shorter ones may not be uncommon. In [27, p. 202-203] the authors discuss research in the field of general operating system processes and it was found that the probability of short CPU usage is much more likely than long bursts. So if a long CPU burst were to happen, than it would not be unreasonable to assume that the tasks that follow are more short-lived. Imagine, for example, a task that consists of heavy computations followed by several small tasks that deal with the result and prepare the next heavy computation task. This pattern could recur any number of times and therefore it is important to handle those cases with some deliberation.

There are numerous possible approaches to this issue, but three are briefly sketched below:

1. The simplest way of handling this would be to let a task only be preempted once and then continue until it either terminates by finishing or triggering t_2. Since this potentially leads to consequences for the whole node, depending on the calculation of t_2 this approach might not be very compelling.

2. It would also be conceivable to grant the preempted task the time the next slot offers and then directly cause a prediction failure if it transgresses on the t_1 im-

posed by the slot that it was inserted to. This handling prevents the snowballing effect described in the paragraph above, but is rigid and therefore may cause some amount of additional prediction failure signals.

3. Another way of dealing with this case could be to set a new t_1 threshold that is based on the slot size of the task the preempted task was inserted to. This behaviour could be repeated again and again until the task finishes or transgresses t_2. The advantage of this approach would be that all the other tasks running on the node could continue with comparable small irritations. A disadvantage is that this could cause a more significant stacking effect, where some tasks are not executed at the time when the original plan had them intended to be executed. If $task_n$ is interrupted repeatedly, increasing amount of tasks would be needed to move to slots further up in the plan. This also requires some advanced tracking capabilities. If using this approach, the base for calculating t_2_task and t_1 diverges, since for calculating t_1 the slot data is relevant and for t_2_task the task data is relevant. This would add an extra layer of complexity to the plan that would not be needed otherwise.

Each of the above mentioned ideas has its advantages and disadvantages. The differences mainly relate to the way each approach deals with successive t_1 transgression and what the base for calculating t_1 is. In the design principles the decision was made that the node should only make decisions that are conceptually within its scope, so a combination of approaches 2 and 3 is chosen. A task will be allowed to be pushed to successors' slots, but the number of occurrences will be capped by a configuration constant called `MAX_PREEMPTIONS` to gain predictability. The checking of this cap will be encapsulated in a t_2-component called $t_2\_preemptions$ which is discussed further in 3.3.6.

The paragraphs above describe the procedure that is followed for pushing tasks to time slots assigned to the same process, but when a task is preempted, it also might be pushed into an upcoming idle time slot. The handling of this case is not as obvious as it might seem, so it will be discussed in the following.

Pushing tasks to idle time slots Idle slots are slots that are not assigned to any process and lie within the so called execution time of the plan-based scheduler. These unallocated slots would leave the CPU in an idle state if the execution would follow the plan exactly. Those idle times are a necessary part of the plan when all currently active processes have to wait for IO-calls to return. Idle time offers the advantage that it is very easy to see the additional resources they provide to the node. Ideally, late tasks can just be inserted in these idle time slots. While this seems very straight forward on the surface two questions arise:

1. What allocation algorithm should be used to assign preempted tasks to idle slots?

2. How much of the idle time slot is assigned to each task in question?

The first question focuses on the notion of ownership or claim. To which process or processes should idle time slots belong? The second question focuses on the problem of partitioning. What should be the size of the chunks the scheduler assigns out of the idle time slot? Should all chunks have the same size and if not, what should determine the chunk size?
Both questions do not have definite answers. Should every process own an equal part of the idle time slot or should the ownership correspond to the planned length of the process or maybe should the neediest process be favored? The scheduler has no further information regarding the estimated runtime of a late task since it already transgressed its planned time. So what is a reasonable amount of additional time? Some possible solutions are sketched out in the following:

1. Fixed pieces of idle time slots based on performance aspects (e.g. costs of context switching) are assigned in a First-come-first-serve manner. If, for example, an idle time slot is 200 in size, the context switching costs are 5 and the fixed piece size is 90 in length, this idle time slot could exactly hold 2 preempted tasks. The earlier preempted task receives the first slot, the succeeding preempted task receives the second slot.

2. Assign each late process equal parts of the unallocated time slot. Assume 5 active, late processes, then each process would receive 20% of the idle time slot. Depending on the size of the idle time slot and the costs of switching between tasks, this might turn out to be inefficient. On the other hand, this would also be a simple and fair procedure.

3. Assign first-come-first-serve time slots based on the node's state (e.g. assign less generous pieces of the idle time slots, when the node is already running late to help late processes catch up more efficiently).

4. Assign all idle time slots in its entirety in a round-robin fashion.

5. Assign all idle time slots to the late processes with the closest deadline.

Each one of the above sketched ideas has advantages and disadvantages and should be evaluated while carefully looking at the requirements as a whole. The round-robin approach for example is easy to calculate and it is predictable, but the CPU might idle while there still could be late processes running on the node. Idea two is fair, since all processes profit to the same extent of idle time slots. Depending on the size of the idle slots and the number of active processes, this concept might turn out to be useless when the emerging idle slots are small. The space of possible solutions is expandable and the ideas above could be improved and reasoned about with increasing detail. Since no obviously optimal solution exists, a look to the defined goals is helpful. Cooperative relations between all entities using a node were assumed in 1.3 and it could be argued that this alludes to a strategy that generally favors the needs of the node over the claims of single processes or tasks. Consider the following scenario: A node is running late. Almost all processes are late and every single one of those processes used up several idle time slots already. Now a task is preempted which belongs to a process that has not received any additional time slots so far. From a fairness perspective, it could be argued that the soon upcoming idle time slot should be assigned to this preempted task since it did not receive any extra slots before, while others already had used several. On the other hand, letting others tasks run more and more late and potentially pushing them towards `t2` or the missing of deadline may not be ideal as well.

Referring back to the assumptions and design goals in section 1.3, a cooperative, predictable concept should be preferred. Since cooperation and predictability are not easily quantifiable values, the proposed solution tries to follow these principles in spirit. Cooperation could be interpreted to mean that processes work together so that ideally all deadlines are kept. This hints towards a solution in which the neediest process is the preferred beneficiary of upcoming idle time slots. Predictability could mandate the effort to let the real execution be as similar to the plan as possible. This could entail that time slots are not to be fragmented but to remain as whole. Combining those two ideas, the proposed solution might work as an easy ad-hoc fill-in solution whenever the plan-based scheduler discovers an idle time slot as its next "task" to execute. The function handling such an occurrence could be implemented like the example in listing 2.

Listing 2: Assigning idle time slots to processes

```
struct PBS_Task* handle_idle_slot(struct PBS_Plan* plan){
    struct PBS_Task* fill_task = NULL;
    struct PBS_Task* next_tasks = get_next_tasks_for_processes(
       plan);
    next_tasks = sort_tasks_by_lateness(next_tasks, plan);
    while (next_tasks->task_id != -1){
        if (next_tasks->was_preempted){
            fill_task = next_tasks;
            break;
            } else {
                next_tasks++;
            }
        }
        if (fill_task == NULL){
            idle();
        } else {
            return fill_task;
}
```

The helper function `get_next_tasks_for_processes(...)` finds the upcoming task for each process. The other helper function `sort_tasks_by_lateness(...)` sorts the task list according to the process latenesses of the plan. The task list will be delimited by a dummy task with *task-ID* -1. The plan-based scheduler can not simply pick the next upcoming task for the latest process, since if the next task was not preempted, it may not be able to run at that point in time, since it might still have to wait for IO-operations to finish. If a task is found, a pointer to the task will be returned. If not, the plan-based scheduler is forced to idle. For this idle time then, the plan-based scheduler could be disabled and the default scheduler could be given priority.

3.3 Prediction Failure Handling

This section is concerned with prediction failures occurring when the plan deviation has reached a significant magnitude. If a prediction failure occurs, a signal will be sent to the job scheduler requesting a rescheduling. This section discusses aspects of the design of this prediction failure component. The following sections go into detail on how to calculate the corresponding thresholds t_2 and t_{-2}.

3.3.1 Prediction Failure Signaling

Compared to t_1, a lot more conditions are conceivable that could warrant a prediction failure signal. Unraveling this conception produces several questions that give a more detailed perspective on the matter:

1. Which signaling conditions does the threshold have to include to achieve the stated goals?
2. Which parameters affect those conditions?
3. How can those effects be reasonably quantified considering the environment?
4. How can the gathered information be combined to enable the scheduler to make a decision for or against a prediction failure signal?

The first question aims to provide a more detailed and concrete view of the semantics underlying the abstract concept of the t_2/t_{-2} thresholds. It alludes to the fact that the plan, despite seeming simple and uni-dimensional, also contains some intricacies that the node would have to manage. Section 3.3.2 is concerned with this question.
The second and third question focus on how to break down signaling conditions into concrete indicators available to the scheduler. The last question relates to the algorithmic procedure of how to combine and balance the different indicators to calculate concrete values that can serve as thresholds. This procedure is explained in 3.3.6 for lateness and 3.3.7 for earliness.

3.3.2 Signaling Conditions

Signaling conditions can be seen as predicates that determine whether a prediction failure signal is to be sent or not. A first step to identify signaling conditions in a systematic manner is by differentiating between plan units, namely tasks, processes, jobs and the node itself. These plan units are in a hierarchical relationship, where tasks are part of processes, processes are part of jobs and also a set of processes runs on a node. The concept of a job is not as relevant for the process scheduler as it is for the job scheduler. Several processes of a job might run on one node, but the node considers deadlines even in the case of only a singular process of a job on the node, so grouping processes together yields no additional advantage. The job-level will therefore be dropped from further consideration. While jobs are not important for the local scheduler, the node itself has some importance for scheduling. On the one hand, plan deviations of singular tasks should be isolated as much as possible to the own process, but side effects are unavoidable, so that under some circumstances the deviations of one process might

Execution Unit	Lateness	Earliness	Timeliness	Preemptions
Task	✓	✓	×	✓
Process	✓	×	×	×
Node	✓	✓	×	×

Table 2: Categorization of signaling conditions

threaten the success of another. Consequently the node state also needs to be factored in to some degree. For each plan unit, a plan deviation in both lateness and earliness is possible.
An additional prediction failure condition should also be coupled to the amount of task preemptions for each task. If a task is preempted repeatedly (stacking preemptions as discussed in 3.2.3), the scheduler should take note and intervene. This aspect is linked to the goal of system predictability. The node has some amount of leeway in the decision-making process, but the actual execution should still be as similar to the plan as possible. By preempting tasks again and again, moving them back further and further, this goal is threatened.
A potential additional aspect that at least has to be considered is the idea of what will be called timeliness in this thesis. While lateness/earliness deal with the relative stretching/shrinking of plan units compared to the plan prediction, timeliness is more concerned with a wall clock concept of time. The idea of timeliness refers to the concept of an implicitly existing time table in which start and end dates for tasks are determined. Since timeliness is a construct that can only be derived implicitly, because it is not intended by the plan, some challenges are associated with it. During the development of this thesis however, the idea of timeliness was dropped due to too many unknown factors making it practically impossible to impose this concept onto a system that was not designed for it.
In a more advanced system additional signaling conditions are thinkable. Consider the mentioned case, where a part of the job has to be run on specific nodes with a special kind of licensed software. The software might be in the process of updating, which was not factored into the plan. Consequently the process can currently not utilize those capabilities, so a prediction failure signal might be warranted.
Other cases such as errors in the plan (duplicated entries, non-existing tasks, etc.), hardware failure and so forth are imaginable. In a future production-ready system, aspects that can be part of a SLA, should also be possible prediction failure signaling conditions. This kind of prediction failures will be excluded from further consideration in this thesis, because of their speculative and potentially arbitrarily expandable nature.

Combining together the different plan unit levels and criteria found above, table 2 can be produced.

A check mark in a cell indicates that the combination of row and column might warrant a signaling condition that needs to be considered.

Lateness The most obvious candidates for signaling conditions are found in the lateness column. If either type of execution unit is significantly late, a prediction failure signal has to be sent.

Earliness Earliness is not as straightforward to handle, since when an execution unit finishes early, deadlines are not in jeopardy. On the contrary, depending on the overall node state, an early finishing unit could take pressure off an overloaded node, but signaling a significantly early finishing task in the general case still is important. If a task finishes significantly earlier than predicted, this could indicate that a relevant change either in the input data or the executing job occurred. The prediction model should be updated to include this observation. Additionally if a node was assigned processes that cumulatively finished exceedingly early, the node might be underutilized which should be prevented. By signaling node earliness, the job scheduler could apply load-balancing measures to the system to improve performance. While task and node earliness are useful signaling conditions to have, the utility of process earliness as a signaling condition is not as apparent, since the process deadline has no inverse concept. One early finishing process also does not provide more useful information about the node state. Therefore the process level is not considered in the case of earliness.

Timeliness The need for timeliness could be inferred by the goals of system predictability and stability. As mentioned above, timeliness refers to the divergence of starting and ending times of tasks implicitly defined in the plan compared to the real start and end times in wall clock terms. Timeliness could also be conceptualized as punctuality. The concept of timeliness is confined to the task level, since adding up start/end time divergences would yield no meaningful insight on a process or node level. A critical requirement for timeliness is a dependable notion of start and end times in a global clock manner. In a distributed system the concept of time is already not trivial. It also is problematic to try to assign start and end times to tasks. If the node would exclusively run the plan-based scheduler, it could be argued that an inference system could more easily be set up. With the default scheduler running as well with a not strictly defined behaviour, at the current state, start and end times are nearly impossible to derive. With the differentiation of computational and communication tasks, the complexity of this issue increases further. While timeliness could be a useful concept, it will be disregarded in the following since no explicit and reliable basis is given or can be constructed. Enforcing timeliness would mean to impose a concept that the plan was not designed for.

Preemptions Including preemptions as another signaling condition would be useful. As mentioned in the general discussion about preemptions in 3.2 stacking preemptions can occur, especially when the distribution of task lengths follows the before mentioned pattern, where a number of short tasks typically follow a long task. In order to prevent a snowballing effect, preemptions should be considered as a signaling condition. Since in the context of a plan-based scheduler, only tasks can be preempted, this aspect can only be related to the task level. It could be argued that processes also could be preempted by halting and disabling them, but this behaviour would need to be initiated by the job scheduler instead of the node. If nodes could decide to initiate process canceling then this would add unnecessary uncertainty to the system state. So since process preemption does not seem advisable, no signaling condition is required.

Following the above categorization, the following signal conditions are selected.

t_2_task : Analogous to t_1 a value needs to be defined that indicates what an acceptable plan deviation is for each task and what should be considered a prediction failure.

$t_2\_process$: In the spirit of trying to keep deadlines when possible, processes that are significantly late or are at a concrete risk of missing their deadline should also be considered in the signaling conditions. Especially for an architecture with SLAs in mind, it is important to be sensitive to missing deadlines.

t_2_node : Node lateness should also be considered. While it is hard to get a clear conceptualization of when node lateness is significant, it should be considered as an indicator for node overutilization. Overutilization might threaten deadlines and stability since little to no leeway is available on the node. If the accumulated lateness is too high, starting and finishing times are shifted more and more in elapsed real-time. To prevent the missing of deadlines, this shifting effect has to be considered and limited. This component of the threshold will be named `t2_node`.

t_{-2}_task : This component is the inverse of t_2_task and should be considered in the same way.

t_{-2}_node : *Node earliness* is the inverse of *node lateness*. If all processes on a node are cumulatively early this is defined as a prediction failure condition. If some amount of tasks finish early and the node is often waiting for new input, resources are wasted. Since a high utilization of the cluster should be aimed for, the node should send a prediction failure signal to trigger a rescheduling, allowing other processes from overutilized nodes to be migrated to the underutilized node or all the processes on the early node can be relocated to other nodes, so the early node can be shut down to decrease operational costs. The component handling this condition will be termed t_{-2}_node.

$t_2\_preemptions$: Task preemptions are an easy to track and evaluate concept that should be included to prevent stacking of task preemptions leading to deviation from the plan.

3.3.3 Signaling Condition Information Sources

With the signaling conditions identified, it has to be evaluated what information is available to decide whether a signaling condition is triggered or not. At a high level glance, the node can use information out of three sources:

1. Information the node receives from external sources that are explicitly meant for the node to process. This is primarily and most importantly the plan.
2. Information the node does not explicitly receive from external sources, but that may be inferable.
3. Information about the node's own state that are produced by using hardware tracking mechanisms (e.g. PMU or clocks) or software interfaces such as *perf*, the `/proc/*`-directory, etc.

Plan The most relevant piece of information given to the node is the (scheduling) plan that is provided and successively updated by the job scheduler. The plan contains a list of tasks. Each task has a unique, identifying number and is part of a process. A task additionally has a planned length given in instructions. The task list is ordered, since it needs to determine the sequence in which tasks have to be executed. Implicitly a scheduling plan contains a definition of each process, since it determines definitely which tasks are part of a process and in what order the tasks of a process have to be executed. For each process only the first task (also identified by the lowest `task_id` of all tasks) is allowed to run, so the execution needs to be in a strict per-process order. Tasks are allowed to be shifted backwards but only if the per-process order of tasks is kept. For example, if a task is exceeding its planned instructions limit, the task may be preempted and moved back to the next time slot that is assigned to its process. Additionally meta-data is included in the scheduling plan, consisting of a per-process buffer size. The buffer size gives the instructions a process has from its last finishing task to its deadline.

Inferable information Information can also be inferred from the environment of the node. The node can collect data about its interactions with the environment. For example it could measure the time elapsed during the request for a rescheduling and the new plan arriving. The node could calculate an average and therefore infer an expected value. Another useful kind of information the node could infer relates to system stability. If a lot of rescheduling is being observed by the node, this could mean that the system is in a volatile state, especially when no new processes arrive at the node and the planned task length is changing frequently. This source of information only provides unreliable indicators, since the node can only observes the effects, without being certain about the cause of those effects, e.g. a cluster might be capable of handling all running jobs easily, but some new jobs arrive in a short period of time causing a lot of rescheduling to occur. Inferable information will therefore need to be handled with care.

Own state The last source of information is the note itself. Since the prediction failure handling component is part of a scheduling policy that is part of the Linux kernel, the node also has access to the information the kernel has access to. This means that scheduling information is available not only for the submitted plans but also for other scheduling policies on the node. The node for example can check run-queues of other scheduling policies. This knowledge might enable a node to more accurately predict its own capabilities. Besides other scheduling information, the node can use CPU facilities for information gathering, e.g. the discussed *PMU* that keeps track of instructions executed, cycles done, cache misses, etc. Another source of information regarding its own state is what is provided by the Linux kernel per default via interfaces such as `/proc` or `/dev` with specific information on the system load; for example in `/proc/loadavg` [20]. Information out of these sources is generally more trustworthy and could be applied when useful.
Another important source of information for the scheduler is information it tracks. Collected runtime data is important, for example to be able to answer questions regarding more high level plan units such as process and node states. This list of self tracked information is detailed in A.1.

3.3.4 Prediction Failure Signal

A signal has one singular purpose and that is to convey all the necessary information to generate a new schedule that is adapted to whatever prediction failure was expressed. Since there are several signaling conditions mentioned in 3.3.2 questions regarding information details and the format of the signal arise. The answer is plain when the signal is caused by one single task. The node can provide the `task_id` and the final number of instructions that were executed before t_2 or t_{-2} was reached. The job scheduler then can easily deduce why the node gave a prediction failure signal. The information the node has to provide is not as clear when no single task triggered t_2 or t_{-2}, but the accumulation of minor plan deviations did. The node could provide a fully detailed list of deviations experienced since the last signal. Or the node could analogous to the above mentioned single-task approach attach the sum of deviations for a process or even the node since the last update. The obvious problem with the full-detail approach is that there could be millions of tasks with small deviations so collecting and transmitting the associated data could require some resources. The job scheduler also would have to unpack and process the data received by some number of nodes, which would in turn also use up resources on the side of the job scheduling system. Too little information on the other hand limits the capabilities of rescheduling. Since the question of level of detail is not easily answered and the job scheduler is also only sketched in this thesis, a simplistic approach will be taken, where a signal will only include what process caused the prediction failure and whether the cause was earliness/lateness. $T_2\_preemptions$ will be interpreted as an issue of lateness.

3.3.5 T_2 Considerations

Prediction failure signaling conditions relating to lateness will be represented and summed up by t_2. In the most simple case t_2 is a fixed number of deviating instructions off from the plan. This deviation could be defined as a constant in a configuration file. The obvious shortcoming would be that this does not allow for discrimination between fundamentally different situations. A process that has only little time left to its deadline would be treated identically to a process that might still have hours or days to its deadline. Dynamics have to be factored into the threshold calculation by recalculating thresholds for different processes and tasks in different situations. Both will be necessary in order to be able to monitor the above-mentioned conditions. In the following paragraphs the calculation of t_2 will be discussed.

Process state: A threshold has to factor in the overall state of a process. It has to recognize that a process being early or late can be evaluated differently in proportion to its progression. If a process is nearing its deadline and has still not cut into its buffer, then chances are good that the process will finish similar to what was predicted. Likewise, if a process that has just started already behaves significantly different then what the scheduling plan has predicted, there might be a reason for concern, or on the other hand, if a process still has hours or even days to run, then early deviations might be canceled out by later deviations in the opposite direction.

Node state: The node's state has to play a part in the definition of t_2 as well. If the resources of the node are utilized near total capacity, a more careful estimation of the threshold compared to a node where only one process is running with a lot of idle time between computation and communication tasks.

Communication costs: The node knows very little about its environmental state. Still it should try to factor in external information, for example the time required for rescheduling. The longer the rescheduling time, the more conservative a node has to be when calculation thresholds. This problem worsens, when it becomes clear that both factors that determine the time it takes to reschedule (round trip time and computation time of a new task) can be subject to fluctuation. The network's capacity may change in unexpected ways and generating a new plan could be trivial or exceedingly complex depending on the overall situation.

Signaling Overload: A node should also take care that its behaviour does not add unnecessary disturbance to the system when avoidable. Since the main interaction between a node and its environment is through getting passed plans and signaling prediction failures, the node has to regulate its signaling behaviour. If task t_n signals an error, then task t_{n+1} should be more reserved with signaling a prediction failure or even buffer the signal for a certain cool-off time. If nodes are able to request reschedules for every task, then this behaviour could lead to an unintended denial-of-service attack on the job scheduler.

3.3.6 Calculating t_2

T_2 is the threshold for signaling a prediction failure. In 3.3.2 major signaling conditions are listed and 3.3.5 mentions what challenges have to be factored in.

T_2 will consist of the different components introduced above, each focusing a different aspect of plan deviation. Lateness is also defined differently on each level. The following sections define the calculation of the threshold components and the appropriate checking mechanism that allow the plan-based scheduler to determine if a t_2 prediction-failure signal should be sent or not.

t_2_task :

Task lateness Task lateness firstly needs to be defined in order to check the threshold against it. If a task has terminated its lateness can be calculated with:

$$\boxed{task\_lateness = instructions\_retired - length\_task\_plan}$$

Following this definition for finished tasks, if $task\_lateness$ has a positive value, the task was late, if not, it was either exactly on time or it terminated early. If a task is currently running and $instructions\_retired < length\_task\_plan$ the `task_lateness` defaults to 0. As soon as *instructions_retired* > *length_plan*, *task_lateness* is updated to reflect the turning late of the task accordingly. Therefore tasks that have not been started yet also have a task lateness of 0. Semantically this makes sense since a task that has not been started yet can neither be early nor late. If the *task_lateness* calculation would not consider this fact odd behaviour would be the consequence.

The value of `t2_task` is determined in a similar fashion as `t1`. A scalar is used accompanied by lower and upper bounds to prevent edge cases from steering the node off by a significant margin. The scaling factor therefore is called σ_{t2}. As with `t1` firstly the relative value is calculated:

$$\boxed{t_2\_task\_relative = length\_task\_plan * \sigma_{t2}}$$

σ_{t2} is a value that defines the maximum relative deviation a task is allowed to have before a prediction failure will be signaled. σ_{t2} is given as a floating-point number that has to be bigger than 1, which represents 100% of the planned instructions. $\sigma_{t2} > \sigma_{t1}$ must be true also. Furthermore there should be a relevant difference in size between `t1` and `t2_task`. Having both values at a near identical value would not be an efficient configuration. To ensure a gap of a certain size, `T2_SPACER` is introduced. It is an offset constant that determines a fixed amount of instructions that guarantees that `t2_task` has a meaningful lower bound.

$$\boxed{t_2\_task\_min = max(t_2\_task\_relative, length\_task\_plan + T_2\_SPACER)}$$

Now that the lower bound is set, `t2_task` has to be adjusted to fit within a reasonable upper bound using `T2_TASK_SIGNALING_LIMIT`:

$$\boxed{t_2\_task = min(t_2\_task\_min, T2\_TASK\_SIGNALING\_LIMIT)}$$

Now with `t2_task` calculated it can be checked against the current value of `task_lateness`. If `task_lateness` > `t2_task` then a prediction failure will be sent. If it is lower or equal, the scheduler will continue running as is.

$t_2 - process$ Above the task level, processes are the next plan units that need to be checked. Firstly, lateness on the process level has to be defined. The value of *process_lateness* captures this process-individual lateness. This value is calculated by taking the sum of each task's lateness that belongs to the process.

$$\boxed{lateness\_process(process) = \sum_{task \in process} lateness(task)}$$

Individual tasks are conceptualized as being relatively short running. If a task terminates its lateness and `t2_task` are of no further concern. The lateness on the process level on the other hand has to factor in accumulating effects of individual lateness and therefore needs to have a broader sense of time that spans a process's full duration. Since these runtimes might become extended, running for hours or even days, accumulated latenesses may persist over some time. This aspect makes defining `t2_process` challenging but also offers some liberties. If the respective time horizon is large enough and the plan deviation adheres to a normal distribution earliness and lateness of tasks will average each other out, but prediction failure handling should not be left to chance and the node should have a way to deal with process lateness in a proactive way. The primary way of balancing out process lateness locally is provided by utilizing the buffers each process has.

Buffer(s) The buffers that can be used to balance process lateness stem from two connected, but conceptually different mechanisms. As mentioned before, one aspect of the buffer relates to the divergence between the ending task of a process and the process's deadline. This buffer will be called *plan buffer*. In some other contexts, for example, Real Time Scheduling or project planning, where a schedule with deadlines exists, this concept has also been called *slack time*. The utility of the *plan buffer* can be most easily described by using a metaphor: If a task's real instructions exceed the plan instructions, the task can be imagined stretched. Assume the plan determines that a task should take 100 ticks, but it takes 110, than the task was stretched by 10%. Now as long as a process's ending task is not pushed beyond the deadline by those stretching effects, the behaviour in terms of process lateness is acceptable. A confident prediction is nearly impossible to make in the general case, so the scheduler should account for this unreliability by applying a variety of layers of safety margin. The *plan buffer* therefore has an important role to play in this evaluation process since a process with a big *plan buffer* can be handled in a more loose way than a process whose last task lies in close vicinity of its deadline. Conveniently the *plan buffer* is submitted as a simple number of instructions so the plan-based scheduler can easily utilize it after applying the above mentioned safety margin.
The second buffer aspect relates to another mechanism that is introduced to provide an inherent safety margin. The job scheduler underestimates the computational capabilities of the node by some amount. The node may be able to run an average of 100,000 instructions per tick, but the job scheduler assumes it can only do 90,000 instructions. The scheduler, therefore expects a task that is estimated to need 1,000,000 instructions to terminate in $11,\overline{111}$ ticks, when the node would actually only need 10 ticks. The $1,\overline{111}$ difference is an additional buffer available for the node to balance out lateness. This extra margin will be called *capacity buffer* in the context of this thesis. The *capacity buffer* is assumed to be a constant defined for each node, that the plan-based scheduler has knowledge of. The constant will be called `CAPACITY_BUFFER`. It is defined as a floating point number that describes the ratio of planned node capacity to the actual node capacity. In the example above, the `CAPACITY_BUFFER` would be defined as $1,\overline{111}$.
Both buffer aspects are incorporated into t_2*_process* using a two-step procedure. While the *capacity buffer* can easily be accounted for, the *process buffers* has to be further processed to include the mentioned safety margins.

The `t2_process` threshold consists of two stages which can be best described by answering two questions:

1. Is the process transgressing its *allowed capacity buffer*?
2. Is the process transgressing its *allowed plan buffer*?

Only if both questions are affirmed a prediction failure signal will be sent. The idea for splitting up the threshold check in two stages is based on the realization that the plan-based scheduler has to only concern itself with the relatively cumbersome *plan buffer* calculation if the *allowed capacity buffer* is depleted. While it is fast, easy and reliable to do a *capacity buffer check*, the *allowed plan buffer check* is more heavily based on assumptions that might not hold. It also requires more processing steps and is less intuitive.

Allowed Capacity Buffer The first check is straightforward to explain and compute. As stated above, the job scheduler underestimates the node's capacity so the $capacitybuffer$ is directly emerging from this difference. The idea for the *capacity buffer* check is to keep track of the degree to which the *capacity buffer* is used up. As long as $capacity\_buffer >= lateness\_process$ it is assumed that the process will be able to keep its deadline. This assumption could be challenged by referring to the fact that the missing of a deadline could be caused by other late processes on the node. Through the stretching of tasks, the completion times of all processes are pushed back in a wall clock sense. If this effect is cumulatively significant enough, a process could miss its deadline despite being on time. To prevent this effect from causing a missed deadline, t_2_node is designed to keep a limit on the overall node lateness. $T_2\_process$ therefore is envisioned to only focus on its own lateness.

The *capacity buffer* per task is calculated with

$$\boxed{\text{capacity\_buffer(task) = (CAPACITY\_BUFFER - 1) * planned\_instructions(task)}}$$

In order to have a meaningful comparison with process lateness, $capacity\_buffer(task)$ has to be summed up for every task that has already been run up until the currently active task. The resulting sum represents the *capacity buffer* available to the process up until the present point in time. Now this value is compared against $lateness\_process$ as it is defined above. Now either

1. $capacity\_buffer >= lateness\_process$ is true, meaning that the process lateness is not significant enough for it to threaten the process' deadline or

2. $capacity\_buffer < lateness\_process$ is true, resulting in the check of the *allowed plan buffer*

Allowed Plan Buffer Since the *plan buffer* is calculated by taking the difference between the planned termination of the last task of a process and the deadline given, the buffer only represents a nominal frame without any further context. A process might have a significant *plan buffer*, but if the node has no resources to spare, the *plan buffer* only exists as a number. In a scenario with no idle times even a minor transgression of the *capacity buffer* could potentially cause the deadline to be missed. This would be an extreme case according to the assumptions made 1.3, but it makes clear that a buffer determined by a pure count of instructions should be investigated further and additional safety should be added to prevent the missing of a deadline. The modifications that should add extra safety margins are laid out in the following. Figure 5 gives an overview of the steps involved.

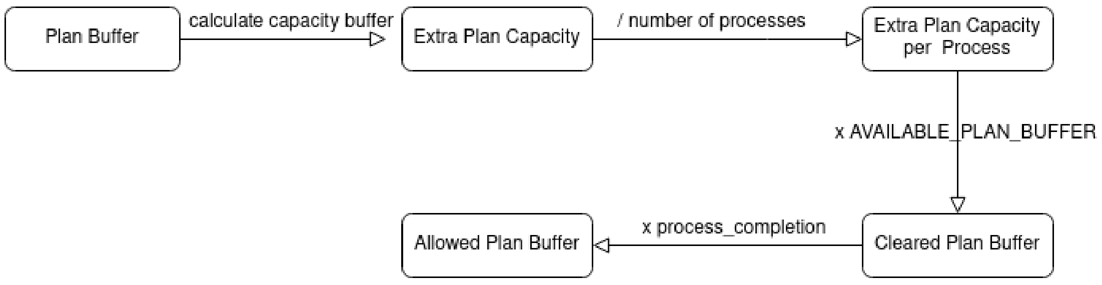

Figure 5: Buffer modification steps

The first step converts the *plan buffer* into what is called *extra plan capacity* simply by calculating the amount of *capacity buffer* that should be available for the corresponding time period. This amount can be calculated by applying the formula of the *capacity buffer* from the task to the buffer:

$$extra\_plan\_capacity = (CAPACITY\_BUFFER - 1) * process\_buffer$$

One potential issue that comes to mind is that idle times are not fully accounted for. Idle times are included in the plan, for example when all processes have finished their computational tasks and wait on the next communication tasks to terminate. As discussed in the preemption behaviour section, idle times are not explicitly claimed by any process (3.2.3), but are assigned in an ad-hoc manner to processes running late. Since stretching/shrinking effects occur with plan deviations, consider the following case depicted in figure 6.

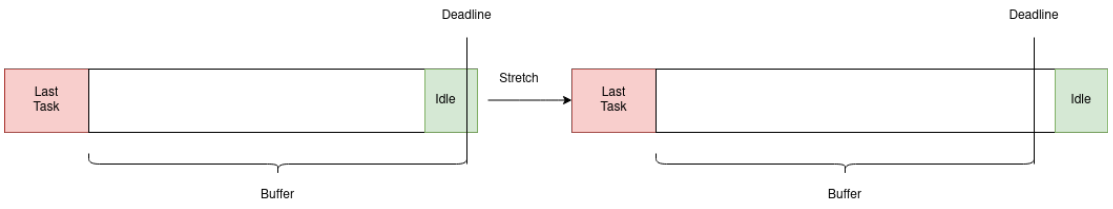

Figure 6: Stretching effects on a buffer

An idle slot may lie near or at the very end of the given plan buffer. If stretching would occur, the idle time might be pushed past the deadline. This case illustrates the challenge of dealing with idle times and buffer modification. One potential solution could be to sum up all process latenesses (which would have to include already finished processes) to estimate this stretching/shrinking effect. But even when applying this procedure it is not guaranteed that there will be idle time, even if computational tasks are on time or early. Maybe IO-operations took shorter than estimated, so one task may be able to run earlier than planned. Also if idle time would fully count towards the capacity buffer, this addition would potentially add large amounts of extra instructions, resulting in a significant impact by idle times. For these reasons, a more speculative consideration is dropped in favor of the more conservative choice, where it is assumed that the buffer does not contain any idle times. Nevertheless, a more productive usage of idle times should be considered when attempting optimizations.

Since we assume a node that is generally running several processes concurrently, the *extra plan capacity* has to be granted to potentially all other processes. To get the *extra plan capacity* of the process, the whole $extra\_plan\_buffer$ is simply divided by the number of currently active processes on the node:

$$\boxed{extra\_plan\_capacity\_per\_process = \frac{extra\_plan\_buffer}{\text{number of processes on node}}}$$

A potential inconsistency arises from the fact, that in the calculation for the *capacity buffer* the full availability of `CAPACITY_BUFFER` is assumed, but in the current step of the buffer modification process this capacity is assumed to be evenly divided among all processes still running at that point in time. In the assumptions stated in 1.3 a normal distribution is assumed, so in the average case, it can be estimated that about half of the tasks do not use any part of their *capacity buffer*. In order to account for this fact as well as add another safety layer if this assumptions turns out to not hold, extra measures have to be taken to increase the probability that the estimated additional resources are actually available. This extra step introduces a new margin, determined by `AVAILABLE_PLAN_BUFFER`:

$$\boxed{cleared\_plan\_buffer = extra\_plan\_capacity\_per\_process * AVAILABLE\_PLAN\_BUFFER}$$

`AVAILABLE_PLAN_BUFFER` is defined as a constant. `AVAILABLE_PLAN_BUFFER` could also be designed as a variable that reflects the scheduler's estimation of capacity buffer usage, capacity buffer sharing, idle times, and so on. If a node for example has 50% idle times, it could be argued that increasing the `AVAILABLE_PLAN_BUFFER` is a legitimate adaptation. For the current purposes `AVAILABLE_PLAN_BUFFER` will be treated as a configuration variable that has the singular purpose of preventing the above described cases from running the node off track.

The final step is to adjust the `cleared_plan_buffer` to the process' progression. If a process just started running, it should not be allowed to consume the full `cleared_plan_buffer`. Especially in the beginning, deviations should be treated more sensitive, since large deviations could indicate major changes in program behaviour. The process progress can be simply calculated by dividing the retired instructions by the planned instructions.

$$\boxed{allowed\_plan\_buffer = cleared\_plan\_buffer * \frac{instructions\_planned}{instructions\_retired}}$$

Finally `allowed_plan_buffer` can be used as the basis for further `t2_process` calculations.

$t_2\_process\_calculation$ On the process level additional factors should be taken into consideration. Since processes are conceptualized as longer running units, the system stability should be observed. The proposed mechanic is inspired by the biological phenomenon of stress. If a threat is perceived, stress hormones are released that modulate behaviour. If the stress inducing incident is dissolved, the behaviour slowly stabilizes. The mechanics of this phenomenon are adapted to inhibit a signaling overload at the rescheduling component. The node has a `stress` level, that is set to `STRESS_RESET` every time the node receives a new plan and each point in `stress` causes an offset of `t2_process` by the value of `STRESS_GAIN`. This interaction turns `stress` into an inhibition factor. Consider the case where a job was structurally rearranged while it still does the same calculations only in a different order. It requires approximately the same computational resources, but the execution flow has changed because for example an intermediate result is needed earlier. All nodes running this job will detect this change and will signal prediction failures. Depending on the implementation of the rescheduling component, the rescheduling component might not be able to deal with a lot of almost simultaneously arriving rescheduling requests. For each incoming rescheduling request the job scheduler would have to determine, if the issue has already been dealt with or if the plan needs to include the newly received information. So every time a new request is received, this might cause a delay to the creation of the new plan, causing potentially more nodes to signal prediction failures. Therefore it could be argued that having a "calm down" period, where nodes refrain from signaling prediction failures is preferable. Since `t2_process` is envisioned as a threshold component that has a long planning horizon, it can be argued that especially here such a mechanism could be useful in stabilizing the system. The value of the threshold `t2_process` is therefore simply increased by the value of $stress * STRESS\_GAIN$. One issue with having a stress system as proposed is that especially for very short running processes with close deadlines, a plan received at an adverse time might prevent the node from sending a prediction failure signal at an appropriate point in time because of this temporary inhibition effect, resulting in a missed deadline. For this reason, the stress system should be easy to deactivate. This can be done via changing the `STRESS_GAIN` value. Setting it to 0 would disable the behaviour fully. So depending on the specifics, e.g. very short running processes, the behaviour can be adjusted accordingly.

Another influence to be considered is encapsulated by `RESCHEDULE_TIME`. Especially in cases where there is a pressing threat of missing a deadline, the time until the prediction failure is sent, the updated plan is generated and the node in turn receives and integrates the updated plan might be relevant. Therefore `t2_process` needs to be reduced by the time (in instructions) the rescheduling process is estimated to take.

Factoring in these aspects, `t2_process` can be calculated by the following formula:

$$\boxed{t_2\_process = allowed\_plan\_buffer + (stress * STRESS\_GAIN) - RESCHEDULE\_TIME}$$

An edge case that is bound to occur regularly for which the formula is problematic, is, when a new process is submitted to the node. If the formula is applied as shown above without safeguards, `t2_process` may turn out to be vastly undersized. When a process starts running with this formula for the threshold, `allowed_plan_buffer` could evaluate to 0, since every process starts with 0% completion. Since the node just received an updated plan including the new process, $stress \approx STRESS\_RESET$, then the value of `t2_process` is determined by $t2\_process = 0 + (STRESS\_GAIN * stress) - RESCHEDULE\_TIME$. Depending on the configuration, namely how the stress values compare to the rescheduling time, at this point `t2_process` might evaluate close or even below 0 with small values for `STRESS_RESET`/`STRESS_GAIN` and/or a large value for `RESCHEDULE_TIME`. To prevent this behaviour, two possible solutions come to mind:

1. Guaranteeing a minimum process progress, to ensure that the `allowed_plan_buffer` term has a significant enough "initialization value"
2. Guaranteeing a minimum value for `t2_process` similar to `T2_SPACER`

While option 1 is more sensitive to the details of the corresponding process, short processes or processes with very small buffers still may end up with very small or negative `t2_process` values. While this formula might still give opportunity to improve upon, the second option will be used with a guaranteed minimum size of `t2_process`. It is called `T2_MIN_PROCESS` and asserts that `t2_process` has a minimum value:

```
t2_process = t2_process > T2_MIN_PROCESS ? t2_process :
    T2_MIN_PROCESS;
```

Looking at the other extreme, at very long running processes and/or processes with vast buffers, it could be argued that symmetrical to `T2_MIN_PROCESS` there should be `T2_MAX_PROCESS` marking an upper bound. On the one hand, by not limiting `t2_process`, it allows the process to use up significantly more resources than it was intended to have. Within the limits of `t2_task`, the process could potentially accumulate a lateness equal to `T_2_TASK_LIMIT` * `number_tasks`. While this might not be a probable case under the chosen assumptions, it obviously could be problematic. However according to 3.3.2, `t2_process` should be concerned with missing process deadlines and not directly with resource overutilisation. The upcoming threshold component `t2_node` does consider this case. While it can be argued that `t2_process` should consider this case, this issue is delegated to `t2_node` to keep the threshold components modular.

Finally the predicate $t_2\_process < lateness\_process$ can be checked. If it evaluates to false, execution is continued as before. If it turns out to be true, a prediction failure signal will be sent.

t_2_node As discussed in the analysis of the signaling conditions and the discussion on `t2_process`, a threshold component should keep overutilization of the node in check. To do so `lateness_node` needs to be defined. Since `lateness_process` keeps already track of lateness on the process level, `lateness_node` needs to sum up the latenesses of (active) processes:

$$\boxed{lateness\_node = \sum_{process \in plan} lateness(process)}$$

To determine the threshold `t2_node` a scalar similar to σ_{t1} and σ_{t2} on the task-level can be used. This scalar is called `T2_NODE_LATENESS_CAP`. The challenge then again becomes finding a suitable value. Potential anchor values are buffers. `T2_NODE_LATENESS_CAP` could be set to the ratio of the process with the earliest deadline to the length of the process. If a process is estimated to run 1000 ticks, the buffer is set to 100 ticks, T2_NODE_LATENESS_CAP should be set to $110\% - safety\_margin$. It would then be reasonable to assume that the stretching effect would cumulatively not threaten deadlines to be missed. Since t2_node is not envisioned as a mechanism to secure deadlines, other anchors can be chosen as well, for example based on the assumed variance of the normal distribution.

Similar to t2_process the stress level should also be included for analogous reasons. If a job changed in structure and all participating nodes detect and signal the change, this could cause stability problems for the whole system so the *stress system* is included again:

$$\boxed{t_2\_node = planned\_instructions * T_2\_NODE\_LATENESS\_CAP + (stress * STRESS\_GAIN)}$$

As it was for other threshold components, the formula above is problematic in certain circumstances. Consider a freshly booted node, where no or nearly no instructions have been retired so far. The node could potentially attain a state where t2_node would be close to 0. To prevent a prediction failure signal t2_node is checked against T2_NODE_LOWER_BOUND, which represents the minimum possible value of t2_node.

```
t2_node = t2_node > T2_NODE_LOWER_BOUND ? t2_node :
    T2_NODE_LOWER_BOUND;
```

The value of T2_NODE_LOWER_BOUND will be calculated by multiplying T2_PROCESS_MINIMUM with the number of currently active processes:

$$\boxed{t_2\_node\_lower\_bound = T_2\_PROCESS\_MINIMUM * number\_cur\_active\_processes}$$

When a lower bound exists, the question if an upper bound is also required arises. In the case of t2_node it can be argued that an upper bound would not provide additional value. This threshold component's goal is to ensure that the accumulated lateness is not crossing a certain border which is to be imagined as node overutilization. Since no assumptions were made regarding the average length of a process, it would be challenging to define a fixed upper cap for node lateness. So in this sense, T2_NODE_LATENESS_CAP can already be considered the upper bound. Establishing a separate, fixed upper bound for t2_node could maybe be useful when reasoning about system predictability, but in terms of the designated goal for this threshold component it is not a necessity.

t_2_preemptions The t2_preemptions threshold component is a pragmatic way of dealing with the problems of stacking tasks that might occur due to the proposed preemption behaviour in 3.2.3. As it was seen before, every model can be designed more nuanced and flexible, as is the case with t2_preemptions. Depending on system and process state, t2_preemptions could be defined more flexibly by reflecting those states. In favor of predictability, for this thesis t2_preemptions is defined as a static value that is checked against the current task's preemptions. It can be checked as follows:

```
signal_t2_preemptions = cur_task->preemptions >= t2_preemptions
   ? 1 : 0;
```

3.3.7 T_{-2} Considerations

The threshold t_{-2} has to signal significant plan deviations that are caused by earliness instead of lateness. Yet, t_{-2} is not simply a change of sign compared to t_2. There is no counterpart to `t2_process`, because processes can not be too early to miss a deadline for example. Also no "negative" preemptions are possible, so `tm2_preemptions` does not translate either. This leaves `tm2_task` and `tm2_node` as relevant conditions to handle. Analogous to their counterparts in terms of lateness, `tm2_task` catches major plan deviations on a task level, e.g. when a task takes 5 timer ticks to finish compared to the planned 500. The `tm2_node` component captures the problem of cumulatively occurring shrinking effects as compared to the stretching effects of `t2_node`.
One perceived difference between earliness and lateness that should be briefly mentioned here is that earliness often does not seem to be considered as problematic as lateness. When looking at literature (e.g. [22]), a tendency becomes clear. In other fields that heavily rely on plans, for example project planning, preventing and managing lateness seems to be the primary concern. In a real world scenario it may be more probable that an unexpected event causes a delay than a speedup, but when thinking about efficient resource allocation, earliness has to be addressed as well. One important difference is that it is trivial to resolve earliness, where the acting unit just needs to turn inactive for some amount of time to eliminate earliness, while resolving lateness requires finding extra resources to recover. Yet, idling is producing unnecessary costs that should be avoided. That said, the implementation is done in a way where threshold components can be enabled or disabled if it turns out that signaling `tm2_node` for example is not yielding desirable behaviour.

3.3.8 Calculating T_{-2}

t_{-2}_task Since `tm2_task` is conceptualized as `t2_task` focused on earliness instead of lateness, the argumentation and definition can be mirrored. The design for `tm2_task` is in a sense trivial, since `t2_task` is envisioned to capture pure plan deviation. It is not related to any other aspect of the plan, but only focuses on the task at hand. If a task was planned to take 100 ticks, but the task caused a prediction failure, when it still was not finished after 150, than the argument for `tm2_task` should be analogous. If 150 ticks cause a prediction failure signal, then taking only 50 ticks should also cause a prediction failure signal.
Since `t2_task` is calculated by multiplying the planned instructions with σ_{t2}, `tm2_task` should be calculated by multiplication with a corresponding σ_{t-2} value. A caveat in this regard is the fact, that for σ_{t-2} to capture early finishing tasks, σ_{t-2} has to be in the value range of $0 < \sigma_{t-2} < 1$, while σ_{t2} has a potentially unlimited value range. So when σ_{t-2} is derived from σ_{t2}, this has to be accounted for. For the scope of this thesis, σ_{t-2} is simply derived by taking the distance of σ_{t2} only in the negative direction with the additional limitation that $1 < \sigma_{t_2} < 2$:

$$\boxed{\sigma_{t-2} = 1 - (\sigma_{t2} - 1)}$$

With σ_{t-2} defined, `tm2_task_relative` can be calculated analogous to `t2_task_relative`:

$$\boxed{t_{-2}\_task\_relative = instructions\_planned * \sigma_{t-2}}$$

As with `t2_task` it makes sense to apply some upper and lower bound checks. In the case of `tm2_task` these boundaries are swapped, since the t_2 lower bound is numerically the new upper bound and vice versa for the t_2 upper bound. As with σ_{t-2}, the upper bound is derived by projecting the distance to the planned instructions into earliness.

$$\boxed{T_{-2}\_TASK\_SIGNALING\_LIMIT = -1 * T_2\_TASK\_SIGNALING\_LIMIT}$$

A potential issue here is the possible value range of `TM2_TASK_SIGNALING_LIMIT`. If `T2_TASK_SIGNALING_LIMIT` has a certain size, then `tm2_task` might potentially be uncapped, when $instructions\_planned + T_{-2}\_TASK\_SIGNALING\_LIMIT <= 0$ is true. Nonetheless, `tm2_task_max` is calculated by first getting the signaling limit:

$$\boxed{t_{-2}\_max = instructions\_planned + T_{-2}\_TASK\_SIGNALING\_LIMIT)}$$

Then feeding t_{-2}_max into a function that returns the biggest of its parameters:

$$\boxed{t_{-2}\_task\_max = max(t_{-2}\_task\_relative, t_{-2}\_max)}$$

The lower bound for `t2_task` was defined by T_2_SPACER. Since there is no preemption equivalent for early finishing tasks and therefore no t_{-1} to relate to, again the bound is mirrored using the planned instructions for the task.

$$\boxed{T_{-2}\_TASK\_SIGNALING\_START = -1 * T_2\_SPACER}$$

Finally, `tm2_task` is calculated:

$$\boxed{t_{-2}\_task = min(t_{-2}\_task\_max, instructions\_planned + T_{-2}\_TASK\_SIGNALING\_START)}$$

Each time a task finishes early, the following check is triggered:

```
    signal_tm2_task = tm2_task > retired_instructions ? 1 : 0;
```

t_{-2}_node Since the problem of missing a deadline due to shrinking effects of tasks is not a concern, a conceptual anchor is missing that was useful when thinking about lateness. The plan does not contain negative buffers, that signify a relevant underutilization, but since `t2_node` was determined by using a configuration variable `T2_NODE_LATENESS_CAP` this concept can be inverted and turned into another configuration variable that will be called `TM2_NODE_EARLINESS_CAP`.
When thinking about upper and lower bounds for `tm2_node` the bounds are swapped. Earliness is not tracked separately. Earliness is negative lateness, resulting in the fact

that $t_{-2}\_lower\_bound > t_{-2}\_upper\_bound$ since the lower bound is closer to 0. So analogous to `t2_node` the calculation is done as follows:

$$\boxed{t_{-2}\_node = planned\_instructions\_finished * T_{-2}\_NODE\_EARLINESS\_CAP}$$

`tm2_node` is then checked against its lower bound:

$$\boxed{t_{-2}\_node = min(t_{-2}\_node, t_{-2}\_lower\_bound)}$$

`Tm2_node` then can be compared to the node lateness as defined for t_2_node.

As argued in the general t_{-2} discussion, earliness is easier to resolve than lateness. For this reason, `t2_node` is intended to be disabled as long as $stress > 0$.

Accumulating earliness One additional challenge to consider arises with `tm2_node` or more specifically with `lateness_node`. For simplicity, consider a scenario where only one job is running on the system. This job shows the behaviour depicted in figure 7.

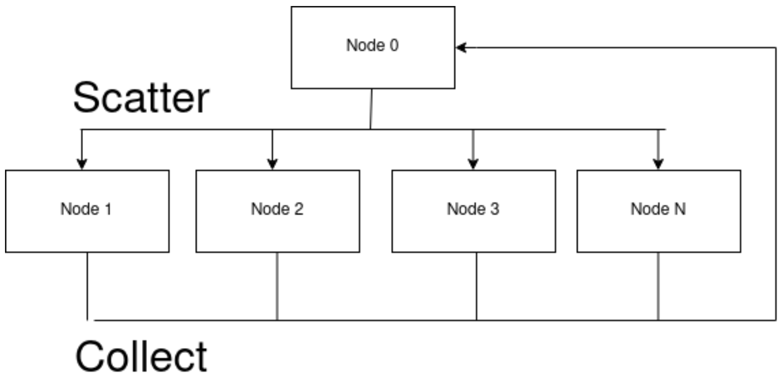

Figure 7: Job behaviour that accumulates earliness

A coordinating node is scattering tasks to all participating nodes. All participating nodes execute their tasks and wait for the coordinating node to collect those results. The coordinating node collects the results, computes the next inputs and scatters those again to all participating nodes. These steps are repeated for some iterations. Now consider the case where one of those participating nodes receives mainly tasks that take less time than those of the other nodes, the node would be idling until the coordinating node would collect its results. This pattern repeats itself over and over again, accumulating earliness on the node. With the way process and node lateness is defined, this node accumulates more and more earliness to the point where a prediction failure signal needs to then be sent. It is easy to see how earliness increases with each iteration, but also how the idling periods act as a synchronization mechanism. So earliness is increasing, but intuitively it would not be true to state that the node is getting more and more early. Programmatically this issue is easily fixed by updating `lateness_node` in the following way $lateness\_node+ = idle\_time$ every time the CPU was idle for some time period. Since lateness is the positive dimension of `lateness_node`, adding idle time to it would diminish earliness. This behaviour would make sense in the above sketched scenario, but consider a process whose communication task has to wait for a TCP handshake to finish. The CPU would idle, but a communication task is still running, accordingly idle time can not be the sole factor. Adding lateness, in this case, does not make sense, since the process does not deviate from the plan. The node only happens to be in a state, where its CPU has nothing else to do.

3.3.9 Post-Prediction Failure Node Behaviour

If t_2 or t_{-2} is triggered, a signal will be sent to the rescheduling component. After the node has requested a rescheduling, it has to bridge the gap between the signaling and the arrival of the new plan in an appropriate way. Appropriate would mean that the node continues on advancing its current plan in a way that optimizes the likelihood of keeping its deadlines. The key question becomes: How does the node proceed in relation to the plan unit that caused the prediction failure. In the case of t_2_node and t_{-2}_node not much room for variations exists since the resulting prediction failure can not be attributed to one process or task. For all other thresholds, the question is more nuanced. For example, resuming the task responsible for the signal might threaten node stability, since it could potentially be erroneous and stuck in an infinite loop. Consequently the deviation would increase and affect other processes on the node. One option would be to disable the responsible execution unit and wait for the job scheduler to decide whether it is allowed to continue to run. The obvious problem with disabling it would be that the process might miss its deadline due to this disabling, while it otherwise would have been able to keep the deadline. The source code of parallel programs can change, the parallel program may receive simpler or more complex input data to process, so deviations can not be regarded as errors that allow the node to discard the process. Yet, by disabling a processes, a potential error would be isolated and further dispersal of the error would be prevented. This issue could be addressed and factored in by the rescheduling component. It could try to reschedule in such a way, that those other processes are shifted forwards, the transgressing process' tasks are then more densely inserted in the upcoming slots. Since advancing a process not only is achieved by allocating computational resources to it, but also relies on IO-operations and synchronization with other processes of the job, this assumption might be considered simplistic. So while there might be reasons to choose this approach, for this thesis and referring back to 3.1.2, the node delegates this decision back to the job scheduler. The stress system prevents immediate rescheduling requests. If a task is stuck in an infinite loop, it will receive further resources repeatedly. It will potentially trigger t_2_task or $t_2\_preemptions$ again and again. In general, this behaviour should increase the likelihood of the responsible plan unit to terminate successfully.

4 Implementation

This chapter aims to give an overview of the implementation approach and the implementation itself. In the first section the implementation approach as well as an overview of the relevant components are given. The following sections will focus on the different implementation levels. The code will be provided via addresses to git-repositories containing the components.

4.1 Layered Implementation Approach

The software implementation of the proposed design comes in three layers, that are ordered by their closeness to the running kernel code. Since kernel-level code is harder to test and debug, it could be argued that it is appropriate to write and verify as much code as possible in an environment that is more fault-tolerant than an operating system kernel. It became clear during the design phase that there had to be a sizable amount of additional code accompanying the prediction failure handling component since at least a minimal environment is to be simulated to be able to run the component. As detailed in A.1 there are also some parameters that change and influence the behaviour of the prediction failure handling component. Therefore a need arose to develop a flexible and transparent implementation that is easy to modify. This three-layer implementation model follows:

Level One: High level, dynamically adjustable simulation which mocks missing environment interactions and allows observing the behaviour of the designed mechanism easily

Level Two: Implementation aid for kernel-level code that focuses on easier testing

Level Three: Kernel-level prediction failure handling prototype

The goal for Level One is to develop an evolutionary prototype (as described in [17]), which allows for quick and easy changes to test out different behaviour. After some consideration, Level One was planned to be implemented in the Python scripting language. One of Python's design philosophies is to be a language with "batteries included" ([16]). Python comes with a range of functionality and formerly popular third party libraries in its standard library. Python is also a dynamically typed, interpreted language that allows for fast changes and requires little boilerplate code. Additionally a lot of well established visualization libraries such as Matplotlib are available, therefore Python seems to be a fitting choice to empirically examine the design choices made. For Level One the design was implemented as a high-level prototype that can be easily written and rewritten to gain some practical insights into the encountered challenges.
Level Two was planned as a way to facilitate kernel development. Ideally, most bugs are fixed on Level Two. The implementation is based on the structure of Level One, but since Level Two will be written in C, there had to be adjustments. The C programming language is conceptually different from Python, which warrants some fundamental structural changes. Additionally, the Level Two implementation was planned and programmed as if it were code that could run in the kernel to assure that the gap between Levels Two and Three is as minimal as possible. This will be discussed in more detail in the corresponding section.
Finally, the Level Three implementation is the prototype that actually runs inside the Linux kernel. It has to be based on the existing prototype, as well as contain some accompanying functionality that allows the copying of the plan from user- into kernel space amongst other things.

4.2 Components

An advantage stemming from a layered implementation approach is that not every component needs to be implemented as part of the kernel source tree, but can be implemented in the most suitable context. For example, the plan generation can be implemented in Python, which supports duck typing so that plans can be dynamically enhanced, which would be more difficult to achieve with a statically typed language. In the following, a list of all planned components is given as well as an overview on what level what component is implemented.

Plan Generation : A component is required to generate plans with planned/real instructions that also considers the assumptions made in this thesis (multiprogramming, normal distribution of deviations, etc.).

Plan Input : This component receives new or updated plans, parses them into the appropriate data structure and makes them available to the plan-based scheduler.

Signaling/Rescheduling : The prediction failure handling component should be able to request a rescheduling, but since no job scheduler can be queried in the current state, this component is also tasked with simulating the rescheduling process.

Instruction Counting The instruction counting component is concerned with providing an interface to read retired instructions for the plan-based scheduler.

Prediction Failure Handling : This central component does the actual prediction failure handling within the plan-based scheduling system. This component should be composed of several sub-components that implement the requirements set in 1.5. This specifically includes a state tracking system that gathers the required information, a component that determines the threshold values, checks the appropriate thresholds against the node state and finally implements the proposed behaviour when plan deviations occur.

Visualization: Since running the plan-based scheduler is assumed to produce a lot of different data points (such as threshold development, lateness, preemptions, ...), a visualization component should be implemented that allows for observing and analyzing the implemented behaviour.

Ideally, as many of those components as possible are not directly implemented in the kernel, but in the simulation or as loadable kernel modules. Loadable modules offer a flexible way to add capability and extend the functionality of the kernel during runtime. This means that the operating system does not need to be rebooted every time the code changes and since modules can be designed to do one specific task exclusively, the code is modular by design and faster to compile. Linux moduls are in essence simply object code artifacts that are linked to the kernel by programs available in a Linux environment [5]. Table 3 gives a quick overview, which component is implemented on which level.

4.3 Level One: Simulation

As stated above, the main requirement for the Level One implementation is to have a simulation that allows testing different behaviors in a fast and flexible manner. It should also provide some environmental functionality such as plan generation and visualization that is most easily achieved by a high-level prototype. The source code for the simulation can be found at github[1].

[1] https://github.com/sherlockhomeless/master_simulation

Table 3: Overview of implementation plan for each level

	Level One	Level Two	Level Three
Plan Generation	✓	×	×
Plan Input	×	×	✓
Signaling/Rescheduling	✓	✓	✓
Instruction Counting	×	×	✓
Prediction Failure Handling	✓	✓	✓
Visualization	✓	×	×

4.3.1 Simulation Overview

The simulation incorporates every aspect that is relevant for running a prediction failure component for a plan-based scheduler. Referring back to the component table above, the simulation contains components for plan generation, signaling or rescheduling and prediction failure handling. The visualization is designed as a separate program that uses the logs that the simulation produces to create graphical representations of the run. The `simulation.py` script initiates the simulation. If the script is given a path to a plan as a command-line parameter, the script loads this plan and simulates a run. If it is not given a parameter, a new plan will be generated based on the default configuration. The main purpose of the `simulation.py` script is to construct a ProcessRunner instance that orchestrates the simulation run. In the following, the most relevant parts of the simulation will be briefly introduced.

ProcessRunner A ProcessRunner is an object that simulates the passing of time and assembles the different parts of the simulation to one coherent entity. Most notably, it has a method `run_tick()` that updates the retired instructions and thresholds, checks for transgressions and initiates the appropriate actions if thresholds are exceeded. It will be discussed in more detail in the next paragraph. `ProcessRunner` also defines the preemption behaviour including handling idle times. It is also the class that manages the logging process. The main logger is kept as a member variable called `self.log_unified`, which does a per-tick log of the most relevant values.

run_tick()-Loop The method where the behaviour of the prediction failure signaling component is most easily observable is within `run_tick()`. The following flow diagram gives a high level overview of the checks and actions this method applies.

The `run_tick()` loop could be optimized by only updating thresholds when it is actually required. For example, all thresholds are updated at every tick, while `t2_task` would only need to be updated when `t1` has been transgressed for example. For keeping a close log on all relevant values, it is useful to always update values according to the current state. Besides determining the basic flow of the simulation, `run_tick()` also provides some additional functionality. It includes an inner function `hold_at_tick(tick_count: int)` for example that allows for stopping at any given timer tick to inspect the state with a debugger.

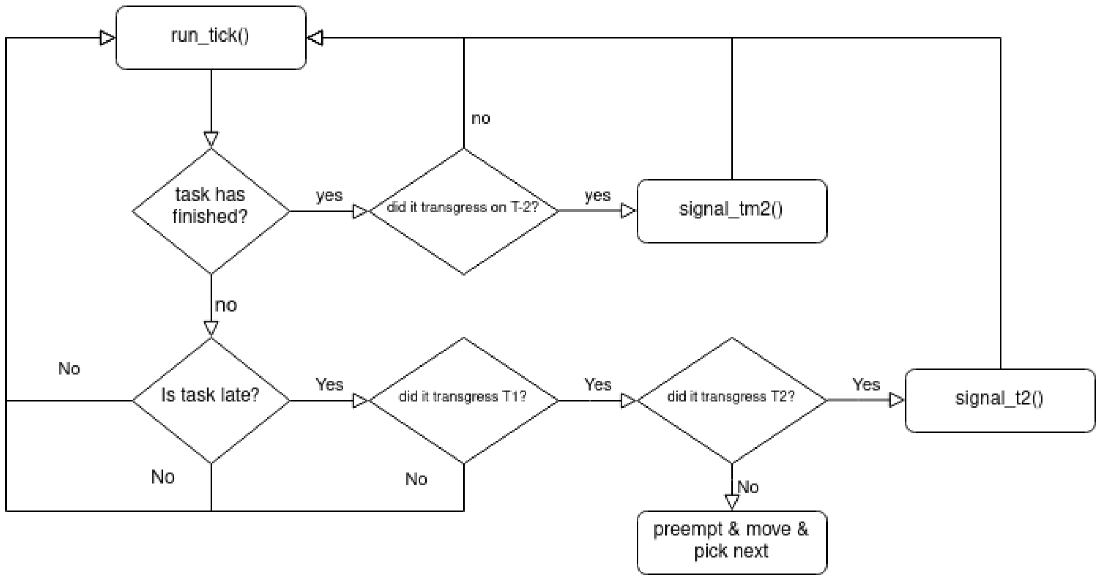

Figure 8: Flow diagram for run_tick() method

Plan Units Another necessary part of the prediction failure handling component are definitions for the plan unit data types. In the simulation, those plan unit classes contain additional member variables that are not present in the C implementation. The purpose for adding those variables is to facilitate comprehension of system state with a quick glance. Code listing 3 shows an excerpt of the `Task`-class definition. While all of the task states shown in the listing can also be determined by the values of the planned and retired instructions, having an explicit boolean is adventagous. Additionally some intricacies are included in `Task` and `Process` classes. For example, due to the design of the preemption behaviour, instructions must be tracked for tasks and slots, so the `Task` class contains a helper class called `InstructionTracking` that manages this.

Listing 3: Example member variables of the class Task

```
...
self.task_finished = False
self.is_running = False
self.finished_early = False
self.finished_late = False
self.finished_on_time = False
self.is_late = False
self.was_preempted: int = 0  # counter for the amount of
    preemptions
self.was_signaled: PredictionFailureSignal = None
...
```

Central Configuration Since it was a goal to keep the implementation as flexible as possible, most relevant parameters are kept in `config.py`, which contains a list of variables that are read by other objects and functions to determine their behaviour. The file also contains a function that updates variables that depend on other values, which is required for running unit tests with a coherent configuration. For example, the instructions per tick rely on the HZ rate of the timer interrupts and the instructions per second. Most notably all the relevant threshold constants are set in this file. Type hints are included to prevent misconfiguration.

Plan Generation The Level One implementation also contains the plan generation component. Since the plan for this thesis needs to include the additional item `instructions_real` to simulate plan deviations, the original plan is extended by this value. Conceptually the plan can be divided into two sections. The first section contains meta information, namely the number of processes and the buffer size for each process, the second part of the plan is the usual list of tasks with process-ids and task-ids as well as the number of instructions predicted by the plan and the "real" number of instructions. The following grammar defines the structure of a valid plan: A plus sign indicates that the previous element has to occur at least one time. The definition is as follows:

$$\langle Plan\rangle \rightarrow \langle Meta\rangle;;\langle Tasks\rangle \tag{1}$$
$$\langle Meta\rangle \rightarrow \langle NumberProcesses\rangle;\langle ProcessMeta\rangle+ \tag{2}$$
$$\langle ProcessMeta\rangle \rightarrow, \langle Process-ID\rangle, \langle Process-Buffer\rangle; \tag{3}$$
$$\langle Tasks\rangle \rightarrow \langle Task\rangle;+ \tag{4}$$
$$\langle Task\rangle \rightarrow, \langle Process-ID\rangle, \langle Task-ID\rangle, \langle LengthPlan\rangle, \langle LengthReal\rangle \tag{5}$$
$$\langle Process-ID\rangle, \langle Task-ID\rangle, \langle NumberProcesses\rangle, ... \rightarrow INTEGER \tag{6}$$

Each plan is terminated by a new line (`'\n'`). Since the machine readable plan is always on only one line, the plan is reprinted below in a more human readable form for debugging purposes. Listing 4 shows this representation. The first and second number are the `process-id` and `task-id`. The following two numbers represent `plan-length` and `real-length` of the task.

Listing 4: Line by line representation of the plan

```
0 0 69008107 101667267
-1 -1 362737754 368037833
1 1 282040993 256276959
-1 -1 225627115 212803897
2 2 69054780 70507765
1 3 339626092 322762296
...
```

The plan generation is done in the class `Plan` in `plan.py`. The plan-generation procedure is defined in `generate_plan()`. This method receives a list of parameters such as the number of processes, the minimum/maximum amount of tasks per process, the percentage of idle time in the plan, buffer information, etc. The simulation creates a plan according to those parameters. The plan generation has some subtleties that need to be mentioned:

- Since the number of tasks per process is randomly generated within the given boundaries, there might be relevant size differences between processes. During the plan generation in `generate_realistic_plan()`, the simulation tries to arrange the tasks in such a way, that tasks of different processes are alternated between. Due to differences in the process size, at the end of the plan there might occur little to no switching.

- Idle slots have the process-id -1 and are not explicitly represented by an instance of the `Process` class.
- Each task has a plan length and a real length. The real length is calculated by applying a normal distribution to the planned length. The real length of tasks can also be set via keyword-argument. The `config.py` file has a sigma parameter `TASK_SIGMA` that determines the variance of the normal distribution. Figure 9 shows an example histogram that depicts the distribution for the plan deviation with the current configuration for `TASK_SIGMA`. Different deviations are sorted into 1000 bins and the number of occurrences per bin is shown on the y-axis. The x-axis shows the distance to the actual planned task length, which was set to 100 timer ticks.

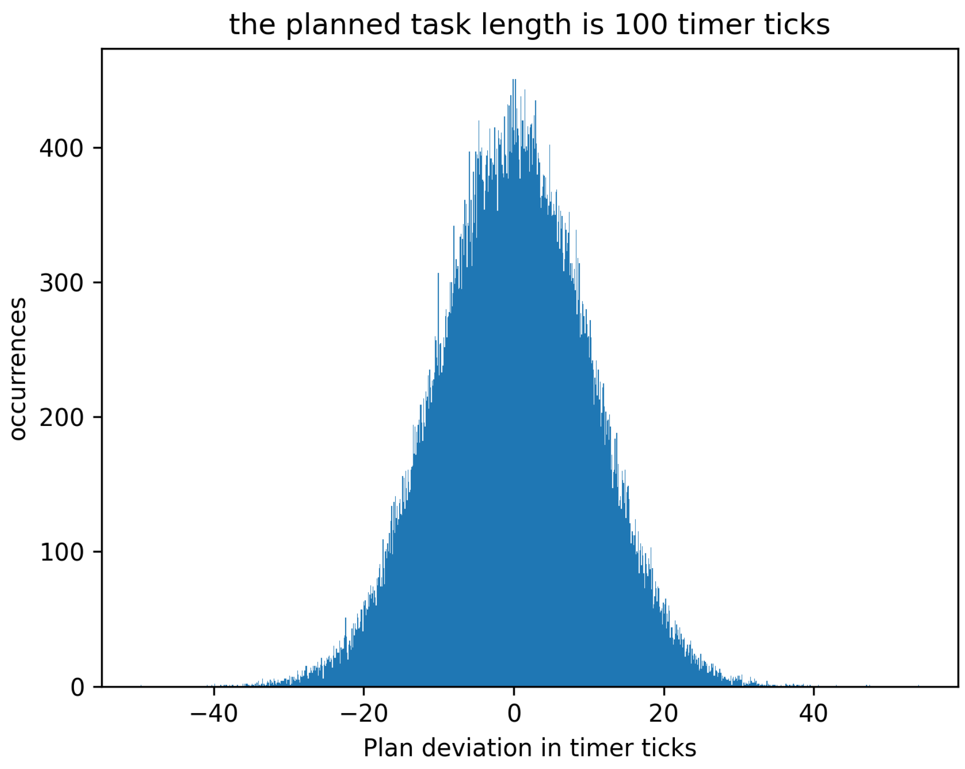

Figure 9: Normal distribution of the plan deviation

Logging Logging plays an important role in the context of the simulation since the logging outputs are the basis for generating visualizations. The various log files are all located in the `/logs` folder of the project directory as defined by the default configuration in `config.py`. Different log files are created for different events, but the most important and complete log file is called `unified_tick.log`. This log lists relevant values on a per-tick basis. A distinction is made here between *sum* and *pure* threshold values. The *sum* value represents the threshold as is including the number of planned instructions, the *pure* value excludes the number of planned instructions to enable comparison of different values from various tasks or processes.

4.3.2 Visualization

Using the logs produced by the simulation, a second program was written that draws graphs portraying different aspects of the simulation. The code for this component is put inside the `/visualization` directory. It contains a parser within `log_parser.py` that is tasked with reading the log files as well as the script that creates the final graphics in `vis.py`. The visualizations are stored in the `/pics` folder.
The value of the extended logging and visualization to this thesis is two-fold: Firstly it aids debugging by providing a facility to trace back and observe unintended behaviour on either the source code or the threshold configuration level. During development, for example, in certain situations $t_1 > t_2\_task$ evaluated to true, which means that a prediction failure has been sent before the task was preempted. Secondly, this approach helped to evaluate different threshold strategies. For example, the visualization could show highly volatile increases and decreases in threshold values. Since system stability and predictability is an important goal of the prediction failure handling component, visualizations especially help to interpret the state of the component.

4.4 Level Two: Userland

4.4.1 Overview

The primary objective of Level Two is to provide implementation help. Since the Level Two implementation is written as a userland program, common techniques and tools are applicable, such as the use of a debugger. The first step is to rewrite the Python-based simulation to C code. A suitable structure is defined based on the general structure of the simulation code. Since the simulation code does not rely heavily on object-oriented features such as inheritance-hierarchies, the translation is in large parts a conversion of syntax and high-level features provided by Python into lower-level C. In a second step, the Level Two code was modified to prepare it for migration into the kernel. This is achieved by adhering to a set of additional restrictions. For example, to use as few standard library functions as possible. Those functions are not available in the kernel and have to be replaced by either the kernel equivalent of the function (e.g. `printf()` by `printk()`) or have to be reimplemented. This also means for example to forgo floating-point operations, since running those in the kernel is generally not recommended ([31]). By adhering to these restrictions, the Level Two and Level Three code should remain very similar. In the following, an overview is given regarding the most important data structures and functionality of the Level Two implementation. The source code can be found at github[2].

[2] https://github.com/sherlockhomeless/master_level2

4.4.2 General Architecture

The primary data structures, namely the plan, process and task are defined in a unified header file called `pbs_entitites.h` which is stored locally on the development machine and symlinked to wherever it is required. This ensures that a change to a data structure is applied to all the components it is used in. For example, the Kernel module that loads the plan from userspace has to know the exact memory layout of all relevant data types in order to work correctly. If the parser uses a different struct definition than the plan input module errors are bound to occur. In order to avoid possible name collisions, the corresponding data structs have *PBS_** prefixed. Figure 10 gives an overview of the most important members.

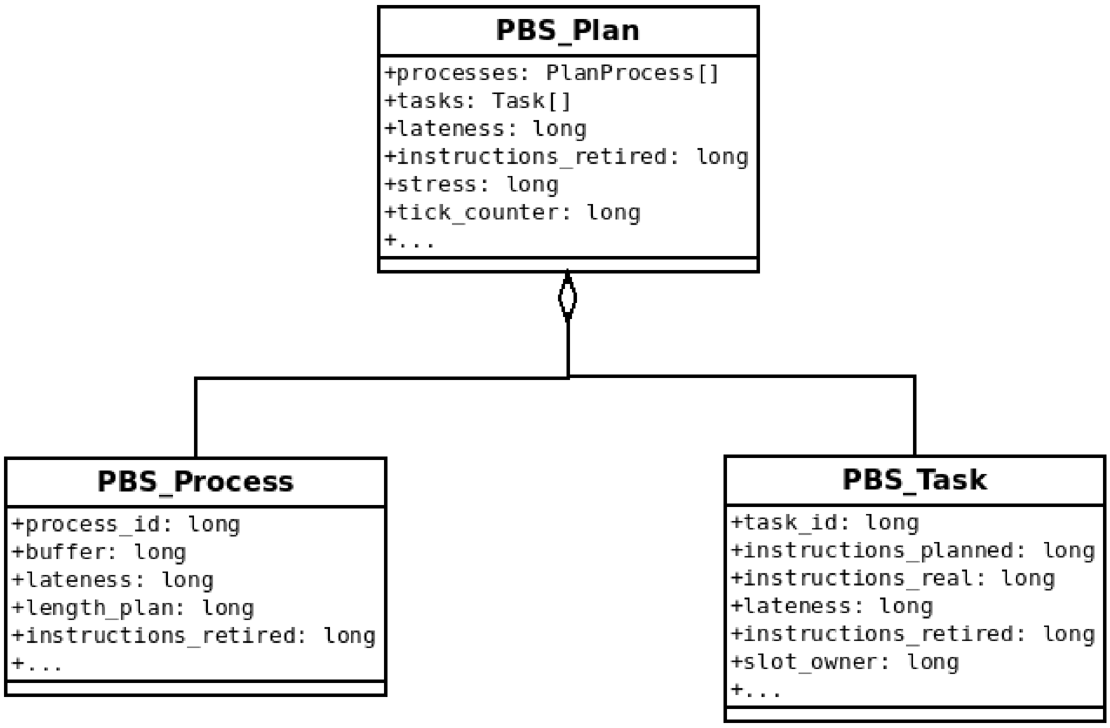

Figure 10: Main data structures representing a plan

Data Structures In the following paragraphs the primary data structures will be introduced briefly. The data structures have not changed compared to the simulation, but an emphasis was put into simplification. The plan unit data structs are defined in `pbs_entitites.h`, but the associated functionality that is required for maintaining those structs during runtime is kept in separate source and header files named after the plan unit, for example, `task.c/h`.

Task and Processes While processes contained a list of their tasks in the simulation, this connection has been removed, since firstly this connection served primarily convenience reasons and secondly, Python automatically manages the memory and object references, while C does require manual updates of pointers. In order to achieve the same level of functionality, a more sophisticated concept for memory management

would have been required, which in turn would have added additional complexity that is not strictly required.
The PBS_Process-struct developed into a simple data type that is only used for keeping track of its retired instructions, process-lateness, progress, etc. Apart from updating those values, little logic is required for its associated `process.c` file.
Tasks do require some more additional functionality since they are the actual units of execution in the context of the plan. This means that in addition to tracking information, they have to provide further capabilities that allow for execution management. One important aspect is handling preemptions (3.2.3). In order to track preemption related information, two fields are required:

1. Retired instructions of the task itself

2. Retired instruction of the slot the task is running in

While in most cases, both fields have the same value, as soon as a preemption is triggered, the handling of tasks and associated threshold calculation requires a more elaborate handling.

Plan The plan data structure is conceptualized as a consolidation unit that ties tasks, processes and node-state-tracking information together. The `PBS_Plan` struct maintains pointers to the currently active task and process. Processes are stored in an array where each process' index is determined by its ID for reasons of simplicity. For productive systems one additional level of indirection should be introduced to provide more flexibility. Tasks were originally stored in manually managed contiguous memory, with `PBS_Task* tasks` pointing to the currently executed task. The task list was implemented using a ring buffer that is constructed by reading the maximum plan lenght out of `config.h`. During implementation, this approach was refactored since memory management for user space and kernel space programs is handled differently. This resulted in a significant deviation between Level Two and Level Three implementation. The manually managed memory was therefore replaced by assigning a fixed length array of tasks to the `PBS_Plan` struct. In order to be able to determine the last entry of the plan, a terminating dummy task is appended. If the plan-based scheduler finds a task with an ID of -2 the list of tasks is terminated. As with `PBS_Task`, `plan.c/h` does contain some additional functionality to manage plans. Most notable, since the plan is thought of as the consolidation unit of all plan units, it provides a unified interface for updating tracking information.

Behaviour While the preceding paragraphs primarily focus on data structures, most of the relevant capabilities such as threshold calculation, preemption behaviour, prediction failure signaling and so on are implemented in several other source files that will be introduced in the upcoming paragraphs.

config.h Since it was a design goal of the prediction failure handling unit to be flexible and adjustable a lot of its behaviour can be modified, enabled or disabled. Sizes, limits and other attributes are configurable. The file where those configurations are kept is `config.h`. It is the equivalent of the `config.py` file. It simply contains a list of `#define` statements. Code listing 5 shows a short excerpt of the file.

Listing 5: excerpt from config.h

```
[...]
// --- T2 ---
#define T2_SIGMA 150 // percentage as int; max allowed
    deviation % of a task from its plan
#define T2_SPACER (5 * INS_PER_TICK ) // raw number
    instructions; Distance t1 -> t2_task
#define T2_TASK_SIGNALING_LIMIT  (PBS_HZ * INS_PER_TICK)// raw
    number instructions; t2_task max value
#define T2_CAPACITY_BUFFER 10 // percentage as int,
    underestimation of node computational capacity
[...]
```

PB-Scheduler Functions in the *pb_scheduler* source file are related to capabilities the scheduler would have to provide. An example would be the *schedule()* function that is called every time the plan-based scheduler is active and a timer interrupt needs to be handled. It also contains logic that is required for updating tracking information and states. Additionally it changes the currently active task if its predecessor has finished or was preempted and therefore handles the "run-queue". It also is responsible for handling idle times as discussed in the design.

Threshold Checking Threshold checking contains functionality that allows the plan-based scheduler to decide whether the current state of the task, process or node warrants a preemption or prediction failure signal. The `threshold_checking.c/h` files provide an interface the plan-based scheduler can use to evaluate its state. The functions are of two types:

1. Functions with a `calculate_*`-prefix calculate actual threshold values in instructions. These functions are useful for understanding the behaviour of the prediction failure component.

2. Functions with the `check_*`-prefix are intended to provide a simple, binary interface, to allow for an easy integration.

In `config.h` one can enable and disable different threshold checks such as *t2_task*, *t2_node*, etc., by setting the value of the corresponding `#define`-statements to either 0 or 1.

Prediction Failure Handling The prediction failure handling is concerned with implementing behaviour that can be initiated after a threshold was transgressed with sending a prediction failure signal or preempting the current task. These capabilities are contained in `prediction_failure_handling.c/h`. Some additional functionality is also required due to the need for managing the memory of the prediction failure handling component. In essence the functionality provided by this unit simply implements the prediction failure handling proposed in the design.

Prediction Failure Signaling The `prediction_failure_signaling.c/h` units are concerned with mocking an environment for the plan-based scheduler that at the current moment is not available. The prediction failure signaling procedure is hidden behind both the functions `reschedule()` and `receive_new_plan()`. The "rescheduling component" sketched by these functions simply applies a shrink/stretch constant to all the tasks of a process which is associated with a prediction failure signal. Code listing 6 shows the loop that iterates through all tasks applying the stretch accordingly. The variable `cur_task` is a pointer that is moved along the list of remaining tasks of the plan. If it currently points to a task that has the same `process_id` as the id that is associated with the prediction failure, the `instructions_planned` field of the task is updated. Due to floating point operations not being available, the 100% mark is represented by the long value of `100` and not a float or double `1.0` value.

Listing 6: applying stretch to tasks

```
while (cur_task->process_id != -2){
    if (cur_task->process_id == target_pid) {
        cur_task->instructions_planned = (cur_task->
           instructions_planned * stretch_factor) / 100;
    }
    cur_task++;
}
```

PMU-Interface The only objective of the *pmu-interface* is to read the correct amount of retired instructions from the performance monitoring unit for the interval between the last and current call to the plan-based scheduler. It will be explained in greater detail that a pmu-interface was not implemented due to challenges to which no obvious solution could be found in the evaluation chapter. For this reason a pmu-interface therefore is mocked as well. The source file `pmu_interface.c` provides one function called `get_retired_instructions()` that simply returns the value of `INS_PER_TICK` that is set in `config.h`.

4.4.3 Testing Strategy

The Level Two implementation contains some code geared towards testing the implementation. The project contains a `main.c` file with an `int main()` function. If build and run, firstly a set of unit tests is executed and then a test run is initiated. The two corresponding functions are called `run_unit_tests()` and `test_run()`. The `run_unit_tests()` function contains explicit tests for functions that contain more complex logic. The function `test_run()` initiates a full test run. Validation is done with *assert()*-statements from the standard library. The `assert()` function takes a boolean expression. If the expression evaluates to true, the program continues. If not, an abort signal is sent and execution is terminated. With this approach, asserts can be removed using a simple search and replace, making it easy to keep the source code kernel-ready.

4.4.4 Level Two to Level Three Migration

In order to write code that can be easily pushed into the kernel source tree, some extra steps have to be taken. Firstly, the Level Two code should be as similar to the future kernel code as possible. Secondly this code must be adjusted, ideally in an automated manner, to account for inherit disparities, e.g. the standard library not being available. Consequently standard library calls must be substituted by a function that does the equivalent on the kernel side. Finaly, the code must be inserted correctly into the kernel build system. This means that the build system must be able to find all relevant declarations and definitions. This three-step process is depicted in the following paragraphs.

Increasing Code Similarity Firstly the user land code has to be made as similar as possible to kernel code. With a growing divergence between both code bases, debugging will become more challenging and time intensive if two code bases have to be updated for every change. The goal, therefore, is to have identical code where possible. Some steps that facilitate this goal, for example choosing the ISO C89 standard for development, since this is also the standard used in the Kernel [15] were taken. The Level Two code also contains some additional macros that allow kernel-level-style code to be written for user space, e.g. the header *kernel_dummies.h* contains empty macros that are required on the kernel side. For example, macros such as *KERN_INFO* or *KERN_ALERT* that are used for `printk()` calls.

Filter Script Naturally some divergences still are bound to occur. To address this a script is called that translates the Level Two sources into Level Three sources. The script `make_src_kernel_ready.py` tries to achieve this goal. It takes two parameters, where the first parameter is the location of the Level Two sources, the second parameter is the target folder where the rewritten source files ought to be copied to. In a first step it filters the input folder on a file level to exclude files such as *main.c* or the *.git*-directory. The remaining files are the inputs to a filter pipeline. The script applies a set of rewriting-rules to every line in the remaining sources, similar to what typical awk/sed scripts would do. In this step for example `printf()` calls are replaced by `printk()` calls and standard-library imports are removed. The code-snippet 7 shows how this replacement works. The function `iterate_lines()` reads all lines of the list of files given in `target_files` and applies the lambda function that is given as the second parameter.

Listing 7: Example filter step

```
iterate_lines(target_files, lambda line: line.replace('
    printf', 'printk'), print_log=False)
```

Kernel Source Tree Integration When the source files are updated and conform to kernel-source-code standards, the build system needs to be made aware of those files. Since every level of the source tree contains a Makefile, the prediction failure handling source code files need to be included. In order to keep both sources close, the prediction failure handling code is copied into `/kernel/sched/prediction_failure_handling`.

The Linux kernel build process is done recursively, so further subdirectories can be included simply by adding the subdirectory to the list of objects to compile by *obj-y += prediction_failure_handling/* ([29]). The subdirectory has to contain a Makefile that is being called. The script `make_src_kernel_ready.py` also automatically creates a Makefile that includes all the relevant source files.

4.5 Level Three: Kernel

Since the Level Two implementation already contains most functional code that is required to run the prediction failure handling component in the kernel, this section is about the additional component that is able to copy plans from user into kernel space. Since running the prediction failure handling component with an appropriate plan is not trivial, the appendix A.2 contains a section on how to set up a test run.

Plan Input In order to enable the plan-based scheduler to run, it must be able to receive customized plans that include both planned and real instructions for each task. Since this functionality is not a main concern of this thesis, but a requirement regardless, the following approach is motivated primarily by convenience and simplicity. The corresponding git-repository can be found at github[3].
The capability to input new plans into the kernel is achieved by implementing a *Loadable Kernel Modules (LKM)*. The module is defined in `pbs_plan_input.c`. This file contains the implementation of a character device that can be read and written to in the `/dev` directory ([3]). Character devices are devices that deal with data in a stream-like, byte by byte fashion. So a new plan can be fed into the kernel simply by writing to `/dev/pbs_plan_in`. The copying of the plan into kernel space is done in a very simple manner. Firstly, a small user land program reads the plan and parses it into the C struct defined in the above mentioned `pbs_entitites.h`. The source code for this helper program is located in the project directory under `write_plan_userland/`. This approach was chosen to minimize the code that has to be run inside the kernel. Since the plan reading module is represented as a file, the user land program can simply write to it via `fwrite()`. On the kernel side, the module receives the plan via `copy_from_user()`. Inside the kernel source tree, a plan variable is declared. Using the exported function `get_pbs_plan()`, the LKM receives a pointer to this global plan variable and fills it with the data it receives from the user land program. The char-device is only opened once, but after 32768 or 2^{13} Bytes written, the `plan_write()` function that implements the writing to the file is called again, most likely due to buffer limitations. The module needs to factor in this behaviour and therefore keeps track of the already written bytes, so when the associated *.write()* function is called, the pointer determining the target location is adjusted accordingly. As long as both user land and kernel space use the same `config.h` and `pbs_entitites.h`, the writing to kernel space should work, still this is a technical detail to be aware of. If a read operation is performed on the character device, this pointer is reset in order to be able to test different plans.

[3]https://github.com/sherlockhomeless/master_read_plan

5 Evaluation

This chapter will give an overview of the degree of fulfillment of the requirements defined in 1.5 and discuss related challenges. Firstly, the functional requirements will be discussed, in the second section the nonfunctional requirements are assessed.

5.1 Evaluation Functional Requirements

REQ-F-0: Interfaces for Integration This requirement requests that the prediction failure component should provide a clear interface for interacting with the rescheduling component. The architecture for the prediction failure component was straightforward to define and develop since every component had a clear cut scope and was therefore delimited. The prediction failure handling component checks all the relevant threshold conditions on each call. For example, if a task has finished, it tests if it finished early and if so, if it finished earlier then `tm2_task`. Every time this check results in the need to signal a prediction failure, either `signal_t2()` or `signal_tm2()` is called. So if the interaction with a rescheduling component needs to be advanced, both of those functions comprise the interface determined in the requirement. In the current implementation both functions only handle logging and apply a simplistic rescheduling, but it would be easy to swap the current stub implementation out for a call to an actual rescheduling component. All the related functions are located in `prediction_failure_signaling.c` and can be easily modified. Receiving new plans is done via the functionality provided by the plan-input LKM described in 4.5. This workflow is not very convenient since the LKM in its current form only copies a binary form of the plan into the kernel memory space. This implementation is functionally working, but to have a more sophisticated interaction its capabilities would need to be extended. In the future works section an outline is given of what an extension could possibly add (6.2.2).

REQ-F-1 & REQ-F-2: Threshold and Prediction Failure Handling Implementation The thresholds are implemented in `threshold_checking.c`. The functionality encapsulated in this file fulfills the requirements formulated in **REQ-F-1**, since it provides a way of deciding whether the current state signifies a prediction failure or not.
Similarly the handling is implemented in `prediction_failure_handling.c` for **REQ-F-2**.

REQ-F-3 & REQ-F-4: Kernel-level code based on prototype The goal was to provide an implementation of the design that was a) running inside the Linux kernel and b) is integrating with the existing prototype. The prediction failure handling component is running successfully inside the kernel and is integrated with the prototype. The integration of the prediction failure handling component is somewhat loose though. The two main reasons for the loose integration are that firstly, the plan-based scheduler prototype so far is not running and executing actual tasks. It runs dummy tasks that consist of an empty for-loop which is iterated over a set amount of times. In 6.2.1 an extension is depicted that would allow a better integration of the prediction failure handling component into the prototype. At the moment the prediction failure handling component is only integrated by a simple hook in `pb.c`, which is called whenever the plan-based scheduler is the highest prioritized scheduling policy. Listing 8 shows this hook.

Listing 8: The prediction failure handling component is called directly from the prototype

```
if (current_mode == PB_EXEC_MODE){
    if (handle_prediction_failure != NULL){
        handle_prediction_failure();
    }
    // run real task here
    picked = pb->proxy_task;
}
```

When the `handle_prediction_failure()` function is called, it increases an internal instructions counter by the amount defined in `config.h`. The prediction failure component also uses its own plan that is adjusted for the purpose by including an extra field that determines the actual amount of instructions the task has to run. The prediction failure handling is solely based on this deviation.

The second reason for the loose integration is that a reliable instruction counting mechanism has not been implemented. Ideally, the prediction failure handling component would only need to be called when certain events occur. For example, when the amount of retired instructions exceeds the amount of planned instructions or when an idle time slot is next up according to the plan. As stated above, the prediction failure handling component makes a simplifying assumption, that every time it is called, a fixed amount of instructions has been retired. It is assumed that the retired instructions were spent on the task that is currently at index 0 in the plan. Since instruction counting is of crucial importance to the mechanics of prediction failure handling it was originally intended to be implemented, but development proved to be challenging. The instruction counter was designed to be implemented as a Loadable Kernel Module. Since LKMs are relatively portable, it was first developed to run natively on a desktop computer with an i5-4460 CPU, but problems occurred when the development was shifted to run and build on an AMD Ryzen 5 2500U and on a Virtualbox VM. When running the binary on the i5-4460 processor, it was able to read some amount of instructions, but on the Ryzen5 and the VM, the same code caused an `"Error opening leader 1"` message. Since the source code by itself does not contain much logic, it was assumed that the hardware support was missing on the mobile CPU and the VM. Code listing 9 shows the relevant source code snippet.

Another hint was given by `perf` itself. With the command `"sudo perf list hw"`, all available hardware events can be listed. On the desktop CPU, this command yields amongst others the line `"instructions [Hardware event]"`. On the VirtualBox VM this command returns empty, which indicates that no virtual PMU is available. When listing not only hardware events, but also kernel-events no equivalent of retired instructions was found. The development for such an LKM was dropped and the focus was put on the prediction failure handling component, since testing custom kernels directly on the host machine proved to be cumbersome. In the future works section (6.2) an instruction counting component is sketched out that could be a beneficial addition to the existing prototype.

Listing 9: opening a perf event for retired hardware instructions

```
memset(&pe, 0, sizeof(pe));
pe.type = PERF_TYPE_HARDWARE;
pe.size = sizeof(pe);
pe.config = PERF_COUNT_HW_INSTRUCTIONS;
pe.disabled = 1;
pe.exclude_kernel = 1;
pe.exclude_hv = 1;

fd = perf_event_open(&pe, 0, -1, -1, 0);
if (fd == -1) {
    fprintf(stderr, "Error opening leader %llx\n", pe.
        config);
    exit(EXIT_FAILURE);
}
```

5.2 Evaluation Nonfunctional Requirements

The fulfillment of nonfunctional requirements is by nature harder to evaluate than functional requirements. In the following paragraphs considerations regarding nonfunctional requirements will be given. Also since the implementation can be divided into the simulation and the runnable prediction failure handling component, this discussion will only focus on the Level Two and Level Three code base. Since it is derived and connected to the simulation, there are tight relations, but due to different language capabilities, diverging goals, etc. the arguments would need to be quite abstract to apply to both implementations.

REQ-NF-0: Modularity Implementing a modular prediction failure handling component was aided by the nature of the task itself. The prediction failure handling component can be easily dissected into different capabilities that share no complex interdependencies. For example, the PMU interfaces can easily be abstracted into a one-function interface that simply returns an integer for the retired instructions of the currently active task. The plan parsing and copying into the kernel can be easily handled by one LKM. The plan units are strictly hierarchical and can therefore easily and clearly be encapsulated. A plan consists of processes and processes consist of tasks. Also the main required functionalities of the prediction failure component can be assigned to the three main subcomponents `threshold_checking`, `prediction_failure_handling` and `prediction_failure_signaling`. Since the implementation relies on the standard C way of uncoupling the interface from the implementation by means of header and source files, components are easily swappable if at a later point in time assumptions are updated.

REQ-NF-1: Configurability Configurability is achieved by having the relevant parameters defined as macros in a separate header file called `config.h`. This file contains administrative configuration parameters to, for example, enable logging or change the assumed timer interrupt frequency. It contains general assumptions such as the CPU's instructions-per-tick capabilites, the rescheduling time, etc. Finally, it contains prediction failure handling configuration parameters for determining threshold behaviour such as the upper and lower bounds, acceptable deviations, variables that are used for the stress-system, etc. This file is heavily used throughout the implementation and via it major behaviour changes can be configured.

REQ-NF-2: Keeping Deadlines The requirement states that if a deadline can be kept, the prediction failure handling component should not be the reason it is missed. Specifically two aspects are focused in the requirement: Firstly the handling should not be the cause for missing a deadline and secondly the overhead should be reasonable. Regarding the overhead, the current implementation of the prediction failure handling component contains parts that add avoidable overhead. For example, if a task is preempted and pushed backwards in the plan, the prediction failure handling component moves the task into the assigned location further back in the plan and then moves every other task before the insertion index one index forward. Depending on the concrete case this behaviour could cause some unnecessary overhead that could be avoided for example by using a linked list instead of an array. So in terms of performance, the prediction failure handling component does add some overhead and thereby could be the cause for a deadline to be missed. However, it can be argued that fixing this overhead is a matter of optimizing the implementation and is not inherent to the behaviour of the prediction failure handling component. Since it was planned to implement the component in a prototypical manner these types of overhead are excluded from the evaluation. One inherent overhead that is a consequence of the design is that some amount of information tracking needs to be done. In every design, even with static thresholds, at least the amount of retired instructions per task has to be tracked. In terms of memory usage, the overhead should be acceptable considering that the target hardware is envisioned to be part of a HPC-system. In terms of inherent overhead, there are two main parts that could contribute. One the calculation of the threshold values and the other is checking whether thresholds are kept. Both are closely related in terms of inherent overhead.

In the current implementation, all thresholds are updated every time the prediction failure handling component is called. This behaviour could be optimized. There are thresholds that do not have to be updated regularly, but at least require a check whether or not they need to be updated. Consider `t1` or `t2_task`; neither need to be updated as long as the current task is not late, but it has to be checked if the current task is late. So all lateness-related thresholds would only be updated if the current task is adding lateness and all earliness related thresholds would only need to be updated if a task has finished with $planned\_instructions > retired\_instructions$. With the availability of programmable PMU overflow events this check could be avoided. Checking the current state of a planning unit against its matching thresholds would also only need to happen if the task turned late or finished early. Yet, due to the implementation being done in a prototypical manner, continuous updates are useful for logging purposes. Minimizing this overhead can also be easily achieved by enclosing the threshold updates and checks in conditional statements.

Disregarding the performance aspect: Could the proposed prediction failure handling component be the cause for a deadline to be missed due to its behaviour? It is easy to construct cases where this would be true, for example, when a process consists only of a few short-running tasks and no buffer exists. Any lateness of any process in-between would cause a missing of the deadline since a rescheduling would most likely not be possible in such short time periods. Another aspect to consider is finding appropriate configurations for the proposed mechanisms. For example, if the minimum deviation before a preemption is triggered, is set very high, a lot of lateness could be accumulated without triggering cautionary measures. If on the other hand, this value is set too low, even minor transgression would cause preemptions and thereby a lot of unnecessary overhead. The challenge of finding a suitable configuration is closely related to the specifics of the processes running on the node. In general, the proposed prediction failure component should provide enough flexibility to adhere to deadlines that are reasonable to keep. In the future works section a possible addition to the proposed design is given, that suggest partly adding configuration responsibility to the job scheduler. The job scheduler could for example tighten upper boundaries to make the nodes more sensitive to deviations if a deadline is in jeopardy.

REQ-NF-3: Observability The intention of the original requirement was to somehow provide an easy way to debug the prediction failure handling component's source code. Kernel level code is by nature hard to debug since conventional debugging tools such as gdb are not available and even debugging by print-statements has its limitation since the `printk` buffer is implemented as a ring buffer with a limited size. This observability is achieved as a side-effect of the choice to implement the component first in user space and then port it to kernel space. Due to this choice, the code can be debugged and observed by conventional tools.

6 Conclusion

6.1 Summary

The goal of this thesis was to design and implement a prediction failure handling component for a plan-based scheduler that runs as a Linux kernel scheduling policy. The thesis is conceptually divided into three parts that are heavily interconnected. Firstly, a threshold system was designed that allows to distinguish acceptable plan deviations from prediction failures. Secondly, behaviour was defined that is associated with different states in regards to plan deviations or prediction failures. Lastly, the designs were implemented based on the existing prototype introduced in [10]. Each of the three parts will be discussed and reflected in the following. A future works section will mention some possible additions to the plan-based scheduler and the prediction failure component.

Threshold Design The design of the threshold itself proved to be challenging and was finally solved by coupling simpler sub-thresholds into a more complex construct, where each sub-threshold focuses on one particular aspect of a prediction failure. The main challenge was to balance competing goals. Most notably, a decision often had

to be made between solutions, where one was more flexible and dynamic on the one hand, but also less predictable and less robust on the other hand. During the first attempt at solving this problem, the approach focused on defining one formula that incorporates all relevant aspects and produces a number that can be compared against the retired instructions of the current task. This threshold's behaviour turned out to be too unpredictable because different parts would trigger due to different conditions that were not easy to predict. This approach was therefore dropped and substituted by a more simple and modular approach of multiple thresholds with sub-thresholds that each would focus on a specific aspect. The basic pattern of having fixed upper and lower boundaries combined with a scaling part in-between is reused and should be sufficiently easy to reason about for a rescheduling component. Threshold components can also be enabled and disabled if they are deemed unfit or unnecessary in the current situation. The challenge with the proposed solution is that it is dependent on finding a reasonable configuration. In the future works section, an addition is proposed that could help resolve this challenge. Another issue was preparing a solution that fits the general case, even when it is not known at the moment. It is risky to make too many assumptions that might not hold because it might result in very little guidance to build upon. For example, a normal distribution for the plan deviations is assumed, but the variance for this distribution function still has to be estimated using a rule-of-thumb approach. Additionally some aspects that would have been very useful to include are out of reach due to having no concept of wall-clock time in the plan. All aspects that relate to this time concept are assumed to be abstracted and handled by the rescheduling component, but developing this clear conceptual distinction was a step by step process.
The definition of the corresponding behaviour also proved to be more challenging than initially expected. For example, deciding on how to assign idle time slots lead to more principle-based questions. For example, in a cooperating system, should additional resources be allocated to the greedy but also most needy process or to the process that shared its resources in the past generously and now itself is in need? Since no testing environment was fully available these types of questions were not trivial to answer. Arguments for each side can be further expanded and developed into arbitrary depths. Limited empirical testing was available through the means of the simulation, but results were often inconclusive and more experimentation capabilities would have been required. Consequently some decisions on how the node should behave in a certain situation formed primarily because a decision had to be made and not because one solution was obviously superior. In a not strictly specified environment questions of this kind often have to be based on assumptions and a design-target direction. So it has to be kept in mind that some designs made for the current implementation are based on principle considerations rather than on classical engineering reasoning.
Regarding the layered approach of implementing the prediction failure handling component, some clear advantages and disadvantages are identifiable. The simulation helped facilitate the decision for choosing a more simple approach in regards to the threshold design. Due to the dynamic nature of the Python programming language, changes could be applied fast and results could be checked easily through the use of available convenience functionality. For example due to the Python object model, every object has a method called `__repr__()` that controls the textual representation of an object if printed to the terminal or to file. This representation can be adjusted easily to reflect a

change of focus. What also turned out to be very helpful is the possibility of extending function signatures with keyword arguments. So clients of a function do not necessarily need to change their calls, but still additional flexibility can easily be inserted if opportune. On the other hand, this kind of extensibility led to ad-hoc changes that were implemented for momentary use, then assumed to be permanent by other parts of the software leading to increasing complexity when adaptations were necessary. So on one hand the flexibility was very useful for writing a prototype fast, but it also had its drawbacks in regards to maintainability. Some refactoring was done with type hinting so it became more clear how to use the available functions correctly. The lack of clear interfaces still proved to be difficult. In hindsight, the simulation should have been retired earlier since it outgrew its original purpose and more functionality should have been directly implemented into the *Level Two* implementation. Not doing so lead to unnecessary additional work. The implementation of the prediction failure component in *Level Two* and *Level Three* turned out to be surprisingly useful and efficient. After some initial problems with programming inside kernel space and some iterations of adjustments the conversion was working reliable. Programming for the kernel became increasingly smoother as the work progressed, even to the point where it was questionable whether a Level Two implementation was even necessary. Yet, the userland implementation still provides very useful features, such as a much better logging and debugging capability, which are impossible inside the kernel. Also the *Level Two* implementation is a more accessible than the *Level Three* implementation since it does not rely on having a plan input kernel module or on the prototype to run.

6.2 Future Work

While working on this thesis some challenges emerged that could be useful to tackle in order to facilitate advancements for the plan-based scheduler. The three main categories where additional research and development could be invested are relating to the environment the plan-based scheduler is running in, the plan-based scheduler itself and the prediction failure handling component. Adding more features to the plan-based scheduler and its environment should be considered with some care to prevent introducing too much additional complexity. With a modular approach and clearly defined interfaces though, some additions might improve further development.

6.2.1 Environment

As mentioned in 5.1, it would be very useful to develop an environment in which the plan-based scheduler could run in. By creating a simulated environment the development of the plan-based scheduler would profit in at least two aspects: Firstly, this environment could be used as the common ground for further development of the plan-based scheduler, as implementations relate strongly to their environments. If the environment is too simplistic, the implementation might turn out to be simplistic. If the environment is too complex, the implementation might become overly complex. So by defining a simulation environment those factors are fixed and therefore a common ground to establish base assumptions is laid. Consequently when building an environment principal questions have to be answered making the task not simply a programming task. Additionally, such an environment can be the base for building a test suite, which could be important help for future endeavors. In more concrete terms, the following paragraphs give suggestions as to what could be done in this regard.

Job Scheduler The most relevant component for the plan-based scheduler to interact with would be a corresponding job scheduler that provides plans and reschedules when requested. Developing such a component could be done in varying degrees of complexity. In its most basic form it would just provide a plan (list of tasks) and runnable processes that correspond to the plan. The main objective would be to generate this plan and those processes in such a way that the plan-based scheduler could actually run them. Technically this means, that `struct task_struct *picked` in the `struct task_struct * pick_next_task_pb(...)` function must be set to an actual runnable task that would run for the planned amount of instructions including a deviation that should be configurable. So the challenge firstly would be to create those runnable tasks, make them available in a transparent way to the plan-based scheduler and then connect the parts accordingly. Ideally this tool should have an interface that would allow generating easy test scenarios. For example by specifying that a plan should be generated where 3 processes are on time but one of the processes has a long running task that exceeds its planned instruction amount by 300%. The component would then generate the plan and create tasks available to the plan-based scheduler. The rescheduling could work in a very similar fashion since the node can just assume that the new plan takes into consideration all signaled prediction failures.

Include Communication Task As mentioned before in the terminology or the discussion about `t2` (3.3.5), the plan-based scheduler does not only handle computational tasks, which this thesis and [10] was concerned with, but also *communication tasks*. Including this notion in the plan-based scheduler would allow for more realistic behaviour. Not every component of the plan-based scheduler needs to know about the distinction. The scheduler by itself, for example, should not need to care if the next task is of communication or computational nature, but to improve prediction failure handling for example, it would be a useful notion for deciding if a computational task can run because its preceding communication tasks are finished.

(Empirical) Research into job behaviour With the access to a production cluster system, it could be very useful to gather statistical data on job and process behaviour. The task of how to approach such a measurement is a challenge by itself, but if some questions could be (partially) answered regarding job/process behaviour and deviations between different runs, the resulting insights could be put into great use by integrating them as assumptions for further development. Some very coarse data can be found (for example [8]), but it is not obvious how to derive a useful model for the challenge at hand. Some additional research and maybe an implementation in the form of a plan generation unit might be useful.

6.2.2 Plan-based Scheduler

Plan Input As with the prototype of this thesis and already mentioned in the future works section of [10], the plan input is done provisionally via LKM. It would be useful to have a designated interface for plan input that should ideally also work for the purpose of plan modification (so a job scheduler for example can more easily be integrated). In order to improve and facilitate development for the plan-based scheduler, this would be a useful addition. This plan input component could be extended by providing additional capabilities, for example:

- Defining a flexible plan data structure that ideally could be easily extended for different use-cases. For example, in the case of this thesis, the plan does not only need to contain a plan-length, but also a real-length for each task. Other examples could be to include task annotation, which would allow printing a string to a log if the task is run. While this extensibility might not be functionally necessary, it might prove to be useful if it is considered in the design.

- Providing an interface for moving a plan data structure in and out of the kernel. Reading the current plan from user space might also turn out to be valuebale.

Instruction Counting Another useful and eventually necessary functionality the plan-based scheduler is depending upon is a robust instruction counting component. The component should have an minimal interface, e.g. one function that takes a `task_id` and returns the number of instructions retired on this task. The instruction counting interface could also be designed even simpler, by just counting the instructions that have been retired in the plan-based schedulers mode, since it was last queried. Meaning, the instruction counting component could simply be a counter that is reset by the plan-based scheduler, which is responsible for interpreting the indications by the instruction counting unit. Another approach could be to broaden the responsibilities of the instruction counting component to serve more generally as the information tracking component. Then it could for example be used in postmortem analysis by the job scheduler. Either way, the instruction counting component would have to have some way of reading or estimating appropriate retired instructions. Ideally this component could deal with different underlying CPU capabilities. Those capabilities could range from full support providing the mentioned overflow interrupts (3.1.3) to no capabilities at all, which seems to be the case for some VM hypervisors. To account for this, an instruction counting component could implement a multi-layered fallback model, where the component would use the best available option for the underlying architecture and then try to substitute missing functionality by filling the gap with estimations, e.g. on a VM with no available hardware counter for retired instructions, the component could approximate the Instruction-Per-Cycle and then apply this indicator to the clock speed to estimate the retired instructions for a time frame. As mentioned in 3.1.3, when following an estimation-based approach, it might be useful to experiment with different timer interrupt settings for the Linux kernel to increase or decrease resolution.

6.2.3 Prediction Failure Handling

As already hinted to in the evaluation, the current implementation of the prediction failure handling component contains some amount of provisional solutions that could be improved.

Prediction Failure Signaling Protocol A challenge when designing the prediction failure handling component was the fact that the node is envisioned to run in a complex and dynamic environment. The node forms a cluster with other nodes, so the behaviour of each node should be considered with respect to that. In the current version this fact is considered with the stress mechanic that forces the node to forgo a prediction failure signal if a new plan was recently received. This behaviour has some potentially detremental side effects. With the current conception, the plan-based scheduler is in binary state in regards to prediction failures. Either the current state warrants a prediction failure or it does not. If the prediction failure component does not signal a prediction failure, then the rescheduling unit will not receive information about deviations and therefore can not improve the schedule. Somewhere in the development of the plan-based scheduler this procedure could be replaced with a more sophisticated solution for example by designing a protocol for requesting reschedules. Instead of requesting reschedules, the protocol could provide a multi-tier approach to the prediction failure signal. Level One signals could just be read by the job scheduler as typical `INFO`-messages, Level Two signals could inform the job scheduler that certain pre-thresholds have been reached, for example 25, 50 and 75% of `t2_process` or `t2_node` have been respectivley. The job scheduler respectivley might already include those updates in its next plan. Every plan could then be sent with an attachment that would state what Level Two signals were already dealt with in the current plan. Another, maybe simpler approach to such a protocol could be that the job scheduler sends configuration information with the plans as well, determining values such as `T2_MAX_PREEMPTIONS` or `T1_SIGMA`. With this approach, the job scheduler would be able to adapt node behaviour in a flexible manner. Pursuing such a protocol might not be of the highest priority, but it could be worth investigating the issue. Finally, the protocol itself could also be expanded to include various other aspects like enabling a health check or check-points for example.

Provide statistical information/ postmortems The job scheduler generates the plan according to a prediction model that utilizes data of previous runs. In order to be able to provide data for the job scheduler, the plan-based scheduler has to first gather and compile it and then offer it in a convenient manner. Since conceptually it would not be advisable to do any sort of data processing that can also be done outside the kernel inside the kernel. The kernel should just provide an easy and clear mechanism to provide data and do housekeeping according to a reliable mechanism to clear old data. While this addition might not provide too much in terms of functionality, it could be a catalyst for further development.

Improve Prediction Failure Handling As noted before, some aspects of the prediction failure handling component are implemented in a naive and simplistic way not only in performance terms, but also conceptually. One example is the handling of idle time slots. Currently, the whole slot is simply assigned to the next process that a) has a preempted task coming up and b) is late. This mechanism could be improved with the current plan-based scheduler. The stress system could be further improved also. Currently, every new plan causes the scheduler to jump to its defined stress level, but maybe the plan was generated due to a new accepted job. The plan-based scheduler could check if the existing plan-lengths stayed the same and decide based on its findings.

If the plan-based scheduler would also contain *communication tasks* this kind of issue could be revisited more thoroughly.

A Appendix

A.1 Information Tracking

Following the convention in programming, constants are written all capitalized and variables with lower case letters. Since we assume that there is a fixed conversion rate that allows us to switch between measurements of time and instructions, it is possible to have mixed units, but for clarity and uniformity the unit of measurement will be instructions.

The following components are kept per node and are valid for all processes that might run on the node.

unsigned long IPS: Instructions per second. This constant is used to convert time units into instructions and vice versa. It is possible to estimate this number by monitoring the different processes running on the system, but in the context of this thesis it is assumed to be independent of the currently running process or phase of the process.

uint HZ: The frequency of timer interrupts per second. This number is set at compile time, but can be dynamically changed via kernel configuration parameters. The current standard is 250, so that a timer interrupt is generated every 0.04 seconds [21].

uint INS_TICK: Instructions per tick, calculated by IPS/HZ.

uint RESCHEDULE_TIME: The time that passes between the occurrence of a prediction failure and the receiving of an updated plan. In reality this component would be dynamic, depending on the overall load of the plan generating component, network capacities, the complexity of rescheduling in the current state,... For this thesis it is assumed to be static, since there will be no integration of this code into an actual running system at this point.

int lenght_task_plan : Number of instructions for a given task according to the plan.

int lenght_process_plan : Sum of all instructions planned for a given task according to the plan.

int lenght_plan : Sum of all planned instructions for the current plan.

int retired_instructions_plan/process/task : Number of instructions retired for the corresponding plan unit.

int task_lateness : Deviation of length plan, number is positive if a task is late, negative if it finished early, else it is 0.

int process_lateness : Lateness on a per-process-level, analogous to *task_lateness*

int node_lateness : The sum of all process latenesses of every currently active process on the node.

uint MAX_TICKS_OFF : The maximum number of timer ticks a task is allowed to be of its planned length before it will get preempted.

uint MIN_TICKS_OFF : Analogous to *MAX_TICKS_OFF*, it is the amount of timer ticks a task may run without being preempted.

float SIGMA_T1 : Similar to ticks off, σ is a factor that describes how big the plan-deviation for a given task may be before it is preempted.

uint PREEMPTION_LIMIT : This limit is a hard cap for preempting a task. If the task reaches this limit, it will be preempted. It is calculated by $PREEMPTION\_LIMIT = T_1\_MAX\_TICKS\_OFF * INS\_TICK$

unit NO_PREEMPTION : This boundary is on the opposite side of scale compared to the above *PREEMPTION_LIMIT*. No task will be preempted as long as it stays within this boundary. It is calculated analogous to *PREEMPTION_LIMIT*: $NO\_PREEMPTION = MIN\_TICKS\_OFF * INS\_TICK$

int MAX_PREEMPTIONS : This constant limits the number of preemptions for each task. If a task is preempted more often than `MAX_PREEMPTIONS` allows, a prediction failure is sent.

float SIGMA_T2: Maximal lateness a task is allowed to reach before triggering a prediction-failure signal.

float SIGMA_TM2: Maximal earliness a task is allowed to finish with before triggering a prediction-failure signal.

uint T2_SPACER: Timer ticks that offset t_2 from t_1 so that unfavorable circumstances are mitigated.

int T2_TASK_SIGNALING_LIMIT The upper bound for `t2_task` given in instructions.

int TM2_TASK_SIGNALING_LIMIT The upper bound for `tm2_task` given in instructions, meaning the maximum number of early instructions that a task is allowed to finish before causing a prediction failure.

int TM2_TASK_SIGNALING_START The upper bound for `tm2_task` given in instructions, meaning the maximum number of early instructions that a task is allowed to finish before causing a prediction failure.

float AVAILABLE_PLAN_BUFFER : Percent of the buffer transmitted that is allowed to be used up.

float CAPACITY_BUFFER : The speed up factor that describes the ratio of planned computational capacity of the node to the actual computational capacity.

int T2_PROCESS_MINIMUM : Minimum value for the 2nd tier `t2_process` check.

uint stress : Keeps track of time passed since last received plan update. The value is set to a number of timer ticks, that are counted down in specified time intervals after a prediction failure.

int STRESS_RESET : The value that stress is set to after the node receives a new plan.

int STRESS_GAIN : The value that each point in stress adds to thresholds working with the stress system.

int T2_NODE_LOWER_BOUND: This value determines the lowest possible value of `t2_node`.

int T2_NODE_LATENESS_CAP: Scalar for Node lateness.

int TM2_NODE_EARLINESS_CAP: Scalar for Node earliness.

Beside the node-global constants and variables, there also need to be elements in place which track per-process information.

uint process_length: The planned total number of instructions that the sum of all tasks of a process has according to the plan.

uint instructions_done: : A counter that tracks the amount of already executed instructions for each given process. Every time tasks of the respective process receive CPU time, this number is incremented.

unit instructions_left: Same as *instructions_done* except that the counter decreases with executed instructions.

float process_completion: The (planned) process completion can easily be calculated by $process\_completion = instructions\_done/process\_length$. It is simply the ratio of the already completed instructions to the planned process length.

uint buffer_size: The amount of additional instructions a process has. This number is submitted to the node by the VRM and represents the difference between the deadline given by the user and the ending time of the last scheduled task.

int process_lateness: The accumulated difference of planned instructions compared to actual instructions. This variable tracks if a process is on time, early or late.

A.2 Compiling and Running the Prediction Failure Handling Component

The kernel modules as well as the code in the kernel source tree were developed using a virtual machine (VM). To ease development and automate the process as much as possible, two additional tools were used.

Vagrant : Vagrant is a tool that allows managing virtual machines through configuration files. If all of the configuration is kept in files, the creation of virtual machines can be done automatically. This is useful especially for testing purposes. If a VM stops to behave as intended, it can be deleted and replaced in a few seconds, avoiding a time intensive debugging process. This proved to be especially useful for testing kernel modifications

Ansible : While Vagrant by itself simply provides an interface to a VM provider such as *VirtualBox*, the VM needs to be provisioned with the required build and debugging tools. Ansible is responsible for attaining this goal. Among its capabilities are installing a capable text editor, the required kernel headers, copying a script to setup tmux on the VM and more.

The current configuration for Vagrant and Ansible can be found at Github[4]. One detail that needs to be mentioned is that in the case of a VirtualBox VM, the VM by default comes with one 10GB partition, which is not enough virtual harddrive space to compile and build the Linux kernel with the default configuration.

Compiling the kernel Once the machine is set up the kernel can be compiled. The git repository containing the latest version of the prediction failure handling component can be found at Github[5]. If the above-mentioned Ansible/Vagrant setup is used, the repository is already cloned in `/home/vagrant/kernel_src/master_thesis_linux`. It is important to clone the branch `pb` to obtain the code for the plan-based scheduler. In order to compile the kernel, firstly, a configuration file needs to be generated. This can be simply done by changing into the top level source directory and run the command `make menuconfig`, choosing the appropriate options and saving them to file. Afterwards, the compilation can be started with `make`. If compiling on the vm, only one core is available, but if the kernel is compiled on a multi core system, the compilation can be sped up using the `-j N` parameter, where N is the number of available cores. Depending on the base hardware and the configuration file, the compilation can take several hours. Also consider that enough space has to be available on the hard drive to store all the object files. Since the Linux kernel also relies on modules, those have to be also installed with `sudo make modules_install`. Finally, the new kernel has to be installed by running `sudo make install`. Afterwards, the system can be rebooted and the new kernel can be selected in the boot menu. To verify that the correct kernel is booted, the command `uname -a` can be executed. The output will contain a version number with the postfix `pbfail` from the `extraversion` parameter.

Plan input module In order to allow the prediction failure handling component to run, it first must receive a plan that also contains plan deviations. Those plans are loaded via a separate kernel module. The module can be found at Github[6]. This

[4]https://github.com/sherlockhomeless/master_vm_provision
[5]https://github.com/sherlockhomeless/master_thesis_linux
[6]https://github.com/sherlockhomeless/master_read_plan

repository contains the kernel module as well as a userland program that writes to the character device implemented by the module. Firstly, the module has to be compiled and inserted. The project contains a `Makefile` that allows compiling the module. The module then can be inserted using the command `sudo insmod pbs_plan_input.ko`. Secondly, the userland program has to be compiled. The program takes a path as a parameter that points to an instance of a plan. As mentioned above, the plans are generated by the simulation. The plans usually have the filename `plan.log`. Invoked with sudo-rights and a correct path, the userland program then feeds the plan into kernelspace. If done correctly, the kernel log will print something similar to listing 10.

Listing 10: kernel log messages printed by the plan input module

```
[ 9992.195142] [PBS_plan_write] 36864 + 4032 = 40896
[ 9992.195143] [PBS_plan_write]0: fixing pointers, before:
    cur_task=00007fff319f7010, cur_process=00007fff319f5d50
[ 9992.195144] [PBS_plan_write]0: fixing pointers, after:
    cur_task=ffffffff81ccefd0, cur_process=ffffffff81ccdd10
[ 9992.195144] [PBS_plan_write]0: plan_ptr=ffffffff81ccdd00,
    processes=ffffffff81ccdd10, tasks=ffffffff81ccefd0
[ 9992.195145] [PBS_plan_write]0: 1st_process=ffffffff81ccdd10,
     2nd_process=ffffffff81ccdd40
[ 9992.195145] [PBS_plan_write]0: 1st_task=ffffffff81ccefd0,
    last_task=ffffffff81cd47d8
[ 9992.195146] [PBS_plan_write]0: 1st_process: id=0,
    num_tasks_remaining=100, instructions_retired=0, buffer
    =7297012721
[ 9992.195146] [PBS_plan_write]0: 1st_task: id=0,
    instructions_planned=1140367204, instructions_retired=0,
    lateness=0
```

Preparing the prediction failure handling component If a prediction failure handling compatible plan is loaded into the kernel, the prediction failure handling component needs to be enabled. In the file `pb.c` in `kernel/sched` the code of listing 11 is included.

Listing 11: code in pb.c that calls the prediction failure handling component

```c
void (*handle_prediction_failure)(void) = NULL;
EXPORT_SYMBOL(handle_prediction_failure);
...
static struct task_struct * pick_next_task_pb(struct rq *rq,
        struct task_struct *prev, struct rq_flags *rf)
{
    ...
    // continue executing the task in PB_EXEC_MODE
    if (current_mode == PB_EXEC_MODE){
        if (handle_prediction_failure != NULL){
            handle_prediction_failure();
    }
    ...
}
```

The prediction failure handling is enabled if the `handle_prediction_failure` void pointer is set from null to the address of the actual prediction failure handling function. To change this value to the correct entry function, the module `lkm_run` is set up. It can be found at Github[7] and sets up a way of running the prediction failure handling component. It creates a device at `/dev/run`. If the device is read from, the function from code listing 12 is called.

Listing 12: the function simulates 100 timer ticks for the prediction failure handling component

```
static ssize_t dev_read(struct file *filep, char *buffer,
    size_t len, loff_t *offset){
     int i;
     for (i = 0; i < 100; i++){
          pbs_handle_prediction_failure(plan_ptr);
     }
   return 0;

}
```

Preparing the prototype Since the prediction failure handling implementation hooks into the existing prototype, this prototype can also be used to run the prediction failure handling component. Due to the current implementation being a fork of the original prototype, the requirements to run the prototype are already included. Automatically generated kernel module files are ready to compile in the path `master_thesis_linux/pb_utils/mod_gen/mods`. These modules start the plan-based scheduler on insertion.

Bibliography

[1] Arm Ltd. *Arm CoreLink CMN-600 Coherent Mesh Network Technical Reference Manual*, 2018.

[2] Heiko Bauke and Stephan Mertens. *Cluster Computing*. Springer, 2006.

[3] Raghu Bharadwaj. *Mastering Linux Kernel Development: A Kernel Developer's Reference Manual*. Packt Publishing, 2017.

[4] Danilo Carastan-Santos, Raphael De Camargo, Denis Trystram, and Salah Zrigui. One can only gain by replacing EASY backfilling: A simple scheduling policies case study. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, may 2019.

[5] Jonathan Corbet, Alessandro Rubini, and Kroah-Hartman Greg. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[6] EPCC. Weather simulation – how does it work? available at https://www.futurelearn.com/info/courses/supercomputing/0/steps/24057 (last accessed 31.08.21).

[7]https://github.com/sherlockhomeless/lkm_run

[7] Stephane Eranian. *Linux kernel profiling with perf*, 2015. available at https://perf.wiki.kernel.org/index.php/Tutorial (last accessed 31.08.21).

[8] Dror Feitelson. Logs of real parallel workloads from production systems. Technical report, The Rachel and Selim Benin School of Computer Science and Engineering, 2019. available at https://www.cs.huji.ac.il/labs/parallel/workload/ (last accessed 31.08.21).

[9] Ian Foster and Carl Kesselman. The history of the grid. *Advances in Parallel Computing*, 2011.

[10] Kelvin Glaß. Plan based thread scheduling on hpc nodes. Master's thesis, Freie Unversität Berlin, 03 2018.

[11] Matthias Hovestadt, Odej Kao, Axel Keller, and Achim Streit. Scheduling in hpc resource management systems: Queuing vs. planning. *Workshop on Job Scheduling Strategies for Parallel Processing*, 2862:1–20, 06 2003.

[12] Intel Corporation. *6th generation intel core: processor family uncore performance monitoring reference manual*, April 2016. available at https://www.intel.com/content/dam/www/public/us/en/documents/manuals/6th-gen-core-family-uncore-performance-monitoring-manual.pdf (last accessed 31.08.21).

[13] Herrmann Jiri, Zimmerman Yehuda, Parker Dayle, and Radvan Scott. Virtualization tuning and optimization guide. available at https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_tuning_and_optimization_guide/sect-virtualization_tuning_optimization_guide-monitoring_tools-vpmu (last accessed 31.08.21).

[14] The kernel development community. *High precision event timer driver for linux.* Available at https://www.kernel.org/doc/html/latest/timers/hpet.html.

[15] Greg Kroah-Hartman. HOWTO do linux kernel development. available at https://www.kernel.org/doc/html/v4.16/process/howto.html (last accessed 31.08.21).

[16] A.M. Kuchling. Python advanced library, 2000. available at https://www.python.org/dev/peps/pep-0206/ (last accessed 31.08.21).

[17] Sherrell Linda. *Evolutionary prototyping*, pages 803–803. Springer Netherlands, Dordrecht, 2013.

[18] Barry Linnert, Matthias Hovestadt, Odej Kao, Axel Keller, and Lars-Olof Burchard. the virtual resource manager: an architecture for SLA-aware resource management. *4th Intl. IEEE/ACM Intl. Symposium on Cluster Computing and the Grid (CCGrid 2004)*, 2004.

[19] *time(1) — linux manual page*, March 2019.

[20] LSB Workgroup, The Linux Foundation. *Filesystem Hierarchy Standard*, 2015.

[21] man-pages project. time - overview of time and timers, April 2020. available at https://www.top500.org/statistics/details/osfam/1/ (last accessed 31.08.21).

[22] Abdelhamid Mellouk. *Real-time Systems Scheduling 1*. ISTE Ltd, 2014.

[23] Timothy Prickett Morgan. Bending the supercomputing cost curve down, 2019. available at https://www.nextplatform.com/2019/12/02/bending-the-supercomputing-cost-curve-down/ (last accessed 31.08.21).

[24] Ahuva Mu'alem and Dror Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *Parallel and Distributed Systems, IEEE Transactions on*, 12:529 – 543, 07 2001.

[25] oracle. *Oracle SPARC Architecture 2011*, January 2016.

[26] Oliver Peckham. Summit snables unprecedented weather simulation, 2020. available at https://www.hpcwire.com/2020/09/02/summit-enables-unprecedented-weather-simulation/ (last accessed 31.08.21).

[27] Abraham Silberschatz, Peter Galvin, and Greg Gagne. *Operating System Concepts*. Wiley, 2014.

[28] Yamada Takeshi and Nakano Ryohei. *Genetic algorithms in engineering systems*, chapter Job-shop scheduling. The Institution of Electrical Engineers, 1997. available at http://www.kecl.ntt.co.jp/as/members/yamada/galbk.pdf (last accessed 31.08.21).

[29] the kernel development comunity. *Linux Kernel Makefiles*. available at https://01.org/linuxgraphics/gfx-docs/drm/kbuild/makefiles.html (last accessed 31.08.21).

[30] top 500 statistics: operating system family / linux, 2021. available at https://www.top500.org/statistics/details/osfam/1/ (last accessed 31.08.21).

[31] Linus Torvalds. Kernel mailing list excerpt, March 2003. available at https://yarchive.net/comp/linux/kernel_fp.html (last accessed 31.08.21).

[32] Smith Warren, Foster Ian, and Taylor Valerie. Scheduling with advanced reservations. *IPDPS*, 10 2000.

[33] Vincent Weaver. *PERF EVENT OPEN(2)*, August 2021. available at https://www.man7.org/linux/man-pages/man2/perf_event_open.2.html (last accessed 31.08.21).

[34] Thomas Willhalm and Roman Dementiev. Intel Performance Counter Monitor - a better way to measure CPU utilization, August 2017. available at https://software.intel.com/content/www/us/en/develop/articles/intel-performance-counter-monitor.html#cpu_utilization (last accessed 31.08.21).

[35] X. Zheng, Z. Zhou, X. Yang, Z. Lan, and J Wang. Exploring plan-based scheduling for large-scale computing systems. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 259–268, 2016.

List of Figures

1 Layers of the VRM architecture, taken from [18] 2
2 Variable partitioning problem 9
3 Backfilling 10
4 Escalation model 15
5 Buffer modification steps 36
6 Stretching effects on a buffer 36
7 Job behaviour that accumulates earliness 43
8 Flow diagram for run_tick() method 48
9 Normal distribution of the plan deviation 50
10 Main data structures representing a plan 52

List of Tables

1 Comparing queue-based systems with plan-based systems 12
2 Categorization of signaling conditions 27
3 Overview of implementation plan for each level 47