Master's thesis at the Software Engineering Research Group

of the Institute of Computer Science

Plan Based Thread Scheduling on HPC Nodes

Kelvin Glaß
Student ID: 4659865
kelvin.glass@fu-berlin.de

First Reviewer: Barry Linnert
Second Reviewer: Prof. Dr.-Ing. Jochen Schiller
Supervisor: Barry Linnert

Berlin, March 19, 2018

Abstract

The Grid is a distributed computing infrastructure with the focus on resource sharing. The Virtual Resource Manager (VRM) is a new approach of a Grid middleware that can be used to share resources with a guaranteed Quality of Service (QoS). The QoS is negotiated and specified in a Service Level Agreement (SLA) for each request which is sent to the Grid. This thesis focuses on resources of High Performance Computing Systems (HPC Systems). During the SLA negotiation process a program execution model (termed *plan*) of the application that should be executed is created in the HPC System. This *plan* is used by the HPC System in order to determine whether it is able to execute the application as specified in the SLA. The *plan* is a graph that represents the application as multiple *task* nodes that potentially interact with each other. Each *task* has a certain start and end deadline. These deadlines indicate when the computation of a task has to start and when the computation should be completed. A plan can be subdivided into schedules (termed *scheduling plans*) for each node in an HPC System. A node of an HPC System has to use this *scheduling plan* as a thread schedule that specifies which HPC application dependent thread has to be executed at a certain time. Therefore, the *tasks* of a *scheduling plan* have to be mapped to threads on the corresponding node.

Current HPC Systems are using Linux as Operating System (OS) on their computing nodes. The Linux default scheduler, the Completely Fair Scheduler (CFS), is designed to be as fair as possible, in order to avoid the starvation of threads. Furthermore, the scheduler allows the user to have a multitasking experience even on a single CPU system. The requirements of a scheduler that processes a *scheduling plan* are different, because the predetermined schedule has to be processed as precise as possible and therefore always has to comply with the deadlines of the *tasks*. Moreover administration and monitoring threads have to be executed that are not part of the *scheduling plan*. The requirements of both scheduling policies are disparate, because the primacy of the *plan* related threads disagrees with the requirement of fairness and the requirement to avoid the starvation of threads. During the execution of *task* threads other threads that are not included in the *scheduling plan* are blocked. The execution time of *task* threads can be much higher than the execution time of other threads. Therefore the scheduling of all threads is not fair. Furthermore, it is possible that threads that are not included in the *scheduling plan* are starving during the *plan* processing, because the *unallocated times* in the *plan* are not sufficient to execute all waiting threads. Therefore the harmonization of both policies depends on the amount of threads that are not related to the *plan* and depends on the structure of the *plan*. Therefore, it is necessary to create a scheduler that processes the *scheduling plan* and fairly assigns the time slots that are not

allocated by the *scheduling plan* to threads which are not related to a *plan task*. The challenge of creating such a scheduler is to harmonize the contrary requirements of the CFS and the *Plan Based Scheduler (PB Scheduler)*. The challenge of this task is to switch between both schedulers as defined in the *scheduling plan* by the deadlines. Furthermore, it is necessary to analyze whether such a scheduling handling restricts the structure of the *scheduling plan*.

Zusammenfassung

Das Grid stellt eine Infrastruktur für Verteilte Systeme dar. Diese setzt ihren Fokus auf die gemeinsame Nutzung von Ressourcen. Ein neuer Ansatz für eine Grid Middleware ist der Virtual Resource Manager (VRM). Dieser ermöglicht es, Ressourcen mit einer definierten Quality of Service (QoS) zu teilen. In dieser Arbeit stehen Ressourcen von High Performance Computing Systemen (HPC-Systemen) im Vordergrund. Die QoS wird in einem Service Level Agreement (SLA) spezifiziert. Darüber hinaus werden in dem SLA potentielle Strafbeträge definiert, welche im Falle einer Nichteinhaltung der QoS zu zahlen sind. Bei der Nutzung des VRM, im Bereich der HPC-Systeme, wird im wesentlichen Rechenkapazität für die gemeinsame Nutzung bereitgestellt. In diesem Kontext ist es das Ziel eines ressourcennutzenden Clients, eine Applikation möglichst schnell auf einem HPC-System auszuführen. Wenn ein Client eine entsprechende Anfrage an den VRM sendet, beginnt der Prozess, welcher die SLA zwischen dem VRM und dem Client aushandelt. Während dieses Prozesses wird auf jedem HPC-System im VRM ein Programmausführungsmodell (genannt *Plan*) entsprechend der Applikation erstellt. Anhand dieses Modells wird ermittelt, ob die Applikation entsprechend der SLA auf dem System ausgeführt werden kann. Die Applikation wird dabei durch einen Graphen modelliert. Dabei stellen die Knoten des Graphen *Tasks* dar. Die Kanten zwischen den Knoten repräsentieren dabei die Interaktion zwischen den Tasks. Jeder *Task* besitzt eine definierte Start- und End-Frist. Diese Start-Frist (bzw. End-Frist) gibt an, wann die Ausführung des *Tasks* beginnen (bzw. spätestens abgeschlossen seien) muss. Solch ein *Plan* kann in Teilpläne (genannt *Scheduling Pläne*) unterteilt werden. Diese *Scheduling Pläne* werden spezifisch für die Knoten eines HPC-Systems erstellt. Ziel ist es, dass ein *Scheduling Plan* als Vorlage für die Ausführungsreihenfolge der Threads auf einem HPC-Knoten dient. Dabei werden nur Threads betrachtet, die Teil der entsprechenden HPC-Applikation sind. Damit die Applikation anhand des *Plans* ausgeführt werden kann, müssen die *Tasks* des *Scheduling Plans* auf Threads des entsprechenden HPC-Knotens abgebildet werden.

Aktuelle HPC-Systeme verwenden für ihre Knoten Linux als Betriebssystem. Der Completely Fair Scheduler (CFS), welcher Linux als Standardscheduler

dient, verfolgt das Ziel, die Threads bei der Zuweisung von Rechenressourcen möglichst gerecht zu behandeln. Diese Strategie wird verwendet, um das Verhungern von Threads zu vermeiden und dem Nutzer das Gefühl einer parallelen Ausführung von Threads zu ermöglichen (selbst wenn das verwendete System nur einen Prozessor besitzt). Die Anforderungen an einen Scheduler, der den *Scheduling Plan* als Vorlage verwendet, unterscheiden sich deutlich von den Anforderungen des CFS's. Dies leitet sich daraus ab, dass der vorberechnete *Scheduling Plan* so genau wie möglich abzuarbeiten ist. Dementsprechend ist es wesentlich, dass alle Fristen der *Tasks* eingehalten werden. Darüber hinaus müssen Administrations- und Überwachungs-Threads ausgeführt werden, welche nicht Teil des *Plans* sind. Daher ist es notwendig, einen Scheduler zu entwickeln, welcher sowohl den *Scheduling Plan* korrekt abarbeitet, als auch die nicht im *Plan* allokierten Zeiten möglichst gerecht auf die Threads, die nicht im *Scheduling Plan* eingeplant sind, zu verteilen. Die Herausforderung bei der Erstellung eines solchen Schedulers ist es, die gegensätzlichen Anforderungen zu harmonisieren. Daher muss entsprechend der im *Plan* definierten Fristen, die Threadausführung entweder auf Basis des *Scheduling Plans* oder des CFS's durchgeführt werden. Darüber hinaus ist es notwendig, zu analysieren ob solch ein Scheduler die Struktur des *Scheduling Plans* einschränkt.

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 oder in den Fußnoten 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.

19. März 2018

Kelvin Glaß

Contents

1 Introduction **1**
1.1 Virtual Resource Manager 3
1.1.1 Architecture 3
1.1.2 Plan Creation 5

2 Assessment Criteria on Scheduling Plans **8**
2.1 Criteria of a Stable System 9
2.2 Analytical Approach of Criteria Definition 9
2.2.1 Minimal Task Execution Time 9
2.2.2 Maximal Task Execution Time 10
2.2.3 Minimal Unallocated Time 10
2.2.4 Maximal Unallocated Time 11
2.2.5 Criterion of a Safe Plan 11
2.2.6 Criterion of a Maximal Plan 11

3 Plan Based Scheduler **13**
3.1 Requirements 13
3.2 Choice of the Base Operating System 13
3.3 Architecture of the Linux Scheduler 14
3.4 Implementation of the Linux Scheduler 15
3.4.1 Representation of Threads in Linux 15
3.4.2 Implementation of the Architecture 16
3.5 Architecture of the Plan Based Scheduler 18
3.6 Implementation of the Plan Based Scheduler 20
3.6.1 Enhancement of the Linux Runqueue 20
3.6.2 Determination of the Current Scheduling Mode 22
3.6.3 Implementation of the Plan Based Scheduler Module . 22
3.7 Implementation of a Plan Based Scheduler Prototype 26
3.7.1 Simplified Thread Handling 26
3.7.2 Dynamical Plan Initialization 26
3.7.3 Kernel Thread Execution Time Measurement 27
3.7.4 Results 29

4 Empirical Analysis **32**
4.1 Analysis of the Maximal Task Execution Time 32
4.2 Analysis of the Minimal Unallocated Time 32
4.3 Results 33

5 Conclusion **36**

6 Future Work **37**
6.1 Design of a Scheduler for SMP Systems 37
6.2 Implementation of a Fully Featured Scheduler 37
6.2.1 Enhancement of the Scheduling Functionality 37
6.2.2 Implementation of the User Interface 37
6.2.3 Design and Implementation of an Error Handling . . . 38

A Appendix **39**
A.1 TOP500 Statistics 39
A.2 Development and Test Infrastructure 39
A.2.1 Virtualization 40
A.2.2 Hardware 40
A.3 Linux Module Generation 40
A.3.1 Linux Module to set a Scheduling Plan 41
A.4 Measurement Results 43
A.4.1 Matrix Multiplication Results 43
A.4.2 Prime Number Generation Results 44
A.4.3 Result Analysis 44

References **46**

List of Figures **47**

1 Introduction

In the last years High Performance Computing Systems (HPC Systems) became popular in medium-sized and large companies. The HPC applications became an integral part of a development cycle. One example of use is the calculation of simulations in the automotive industry. Instead of performing multiple real world tests, the HPC calculates multiple simulations that considers relevant physical factors in an appropriate degree of detail. The models that are used in a simulation are created by a department that considers the visual and technical design. The results of the simulation can be used by this department to optimize the model. Therefore, it takes time until the modification derived from the former simulation results are applied to the next model version. Thus, it is likely that a delay between two simulations exists.

Often each department of a company (as in the example the design department) has its own HPC System in order to guarantee exclusive access. This is done, because each department needs the guarantee that its jobs are calculated until a certain deadline, because a delay of the calculation would cause a delay in the whole development cycle and could block other departments that are part of it. If the HPC System is shared it is possible that the deadline expires before the job is calculated, because other queued jobs are blocking the HPC System.

However, the purchase as well as the maintenance are costly and requires trained personnel. Furthermore, it is likely that the exclusive handling leads to multiple systems with low utilization, because each department has to process the results of the previous HPC application run or simply has to process tasks that are not dependent on an HPC System. The economic objective of each company is to minimize their costs and maximize their benefits. The same holds for their HPC Systems. Therefore, the companies have to maximize the utilization of the systems and minimize the cost. The sharing of the systems would maximize the utilization and the need of HPC Systems would shrink and also the cost, but the departments still need the guarantee that each job will hold its deadline. Furthermore, it would be useful to compensate peak demands with external resources that are dynamically reservable. A solution that fulfills these requirements has to share internal resources (the company's HPC System) as well as external resources (systems that can be used to compensate peak demands). Therefore, a Grid Computing solution could be applied.

The concept of Grid Computing is proposed by Ian Foster and Carl Kesselman in [1]. The Grid is a distributed computing infrastructure with focus on resource sharing. Therefore, the objective of a Grid is to provide access

to resources for multiple different clients. Furthermore, a resource has not to be a physical resource. As described in [2], other kinds of resources are for example informational resources (e.g. databases) or individual resources (e.g. people and their expertise). Current Grid Computing systems do not support such a guaranteed deadline handling as described before. However, the Virtual Resource Manager (VRM) of [3] introduces such a concept with its Service Level Agreements (SLA) that are negotiated between the VRM and a client.

During the SLA negotiation process a *plan* is created based on different prediction models and historical data which are part of the client's request. This *plan* is a model that represents how the application of the job could be executed on the HPC System. Therefore, it is used to analyze whether the system can fulfill the corresponding SLA. This *plan* is divided into multiple *tasks* where each *task* has its own absolute start and end time. If the SLA is accepted by the system and the client, this *plan* will be used as a specification how the application has to be executed in order to fulfill the SLA. This *plan* will be divided into HPC node specific *scheduling plans*. A *scheduling plan* and its corresponding threads have to be executed by the node Operating System (OS). The thread scheduler that uses the *scheduling plan* as a schedule is termed *Plan Based Scheduler (PB Scheduler)*. Besides the *plan* related threads, the OS has also to execute other threads as administration and monitoring threads that are vital in order to monitor the HPC nodes. The common requirement of a scheduler that schedules arbitrary threads is fairness and therefore the avoidance of the starvation of threads. The requirements of the *PB Scheduler* are to execute the *scheduling plan* as precisely as possible and to avoid exceeded deadlines. Therefore, a *scheduling plan* thread which has to be executed should preempt any other thread. This leads to contrary requirements, because the primacy of the *task* threads disagrees with the requirement of fairness. Futhermore, the execution of a *task* thread blocks the execution of other threads. Therefore, it is possible that other threads are starving.

The main topic of this thesis is to design a scheduler that harmonizes the contrary requirements of the *scheduling plan* processing and the scheduling of arbitrary threads. It is possible that system dependent constraints of the *PB Scheduler* restrict the structure of the *scheduling plan*, because the scheduler is probably not able to execute arbitrary *scheduling plans*. Therefore, it is necessary to identify such constraints and derive the structure of a feasible *scheduling plan*.

This thesis focuses on OS specific components as the scheduler, but is located in the domain of Grid Computing. The remainder of this thesis is organized as follows: In order to illustrate where the main topic of this thesis is located,

the subsequent sections explain the basic architecture of the VRM as well as the HPC specific *plan* and the node specific *scheduling plan*. The subsequent Sec. 2 identifies possible sources of constraints that could restrict the *scheduling plan* and uses an analytical approach to verify whether they restrict the *scheduling plan* in a real system. In Sec. 3 the architecture and implementation of the base OS scheduler is described and analyzed. Based on this result the design of the *PB Scheduler* is introduced and it is explained how the design could be implemented. The following Sec. 4 uses a prototype implementation based on the knowledge of the former section to augment the analytical results of Sec. 2 with empirical data. The last Sec. 5 and 6 summarize the results of the previous sections and highlight open research aspects.

1.1 Virtual Resource Manager

The VRM is a Grid middleware that orientates on the requirements of [2] and therefore introduces SLAs in order to guarantee a certain Quality of Service (QoS). An SLA is a definite contract that contains inter alia information about the service the client expects as well as penalty fees that have to be paid if the rendered service diverges from the agreed service. This can also comprise the definition of a deadline of a computational job. The VRM does not substitute the software that is currently used on HPC Systems, but has to abstract from their Resource Management Systems (RMS). A RMS is a software that has the task to manage the resources of an HPC System. This means that the system reserves resources (as e.g. CPUs or Storage) for a new job of a client and also releases these resource when the job is completed.

The VRM consists of Administrative Domains (AD) which are used by the negotiation process that is responsible to negotiate an SLA that complies with the requirements of the client and fits with the offered services of an HPC System.

1.1.1 Architecture

The AD is a central component of the VRM. An AD comprises the resources for which a certain set of policies is applicable. These policies specify the visibility and access rights of the resources. As depicted in Fig. 1 an AD is modeled by an Administrative Domain Controller (ADC) that is connected to multiple Active Interfaces (AI). Each interface is also connected to a Resource Management System.

1.1.1.1 Administrative Domain Controller

The ADC has the main task to publish information about its associated resources (according to the specified policies) and receive and process requests

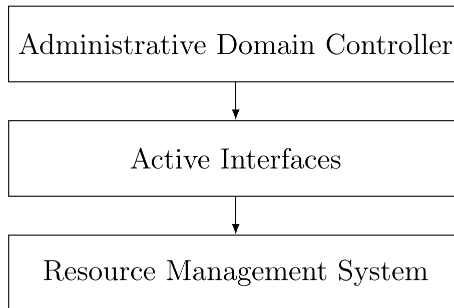


Figure 1: Architecture of an AD.

from the outside. Furthermore, ADs can be nested (an AD is used as resource by another AD) so that the VRM consists of a hierarchical AD structure. The hierarchical structure can be used to aggregate and manage systems that are part of the same organizational unit (e.g. systems of the same company). Moreover the ADC is responsible to manage the negotiation process for its underlying resources and sub-ADs. The information published by the ADC can be used by the negotiation process to determine whether an AD could fulfill an SLA. Furthermore, it is possible that resources in an AD are composed in order to create virtual resources (e.g. to create a big virtual 32 bit system of a 32 bit system and a 64 bit system) which fulfills more complex SLAs.

1.1.1.2 Active Interface

The AI has the purpose to abstract from different RMS and provide a unified interface. This interface is implemented through the functionality of the RMS of the corresponding resource. Current RMSs use an advanced reservation policy with a precise request structure [3]. In contrast, features as diffuse requests or dynamic resource allocation [4] would allow more complex SLAs. A diffuse request is a request that allows a scope of possible results (e.g. request 32-64 CPUs). A dynamical handling would allow to assign new resources to a job during its execution. The features of the RMSs are different, hence it is not always possible to implement the interface through an RMS. Therefore, the systems are classified into systems that already provide the functionality of the interface, systems that provide enough information to implement the AI and systems that do not provide enough information to implement the interface. The last kind of systems are also integrable into the VRM but they are not usable to fulfill a request of a client that requires a certain QoS.

1.1.1.3 Negotiation Process

The negotiation process is always triggered if a client sends a request to the Grid. This request is forwarded to the VRM and its root ADC. The ADC starts the initial check that verifies which resources have the potential

to fulfill the request. This comprises the basic check whether the resource requirements fit with the corresponding resource (e.g. the check fails if a 64 bit system is requested but the resource is a 32 bit system). The negotiation process is triggered in the ADCs of the resources that passed the first check (this causes a recursive negotiation process). The results of these processes are aggregated and returned to the Grid. Depending on the returned result the client can adjust requirements and retrigger the process or accept the SLA.

After the first check is passed each ADC has to create a job specific *plan* in order to estimate whether it can fulfill the SLA. This *plan* is created based on the application that has to be executed as well as application specific historical data. Both pieces of information have to be provided by the client that sent the currently negotiated request to the Grid. If an ADC accepts an SLA, the formerly created *plan* is used as a specification how the application has to be executed in order to comply with the SLA. This *plan* is divided into multiple *scheduling plans* which are executed on the nodes of the corresponding HPC System.

The main topic of this thesis is to verify whether such a predetermined schedule can be executed in the OS of an HPC node. Therefore, it is necessary to determine which constraints are implied by a scheduler that executes the *scheduling plan* as precisely as possible. The following section describes in more detail how the *plan* is modeled and created.

1.1.2 Plan Creation

For each job the Grid receives, a *plan* is created. This *plan* results from the negotiation process of the VRM and can be modeled as a program graph. Common models of a program are the Task Interaction Graph (TIG) and the Task Precedence Graph (TPG). Both are directed graphs where each node represents a task and each directed edge points from a task t_i to another task t_j. In a TPG an edge indicates that t_i has to be executed before t_j. In a TIG an edge indicates that a communication flow exists from t_i to t_j. The used graph is a combination of TPG and TIG, because the TPG information is necessary to determine a scheduling order and the communication information of the TIG is necessary to determine whether network resources has to be reserved for the job. Each node has at least four additional fields with values for an ID that identifies the HPC node that should be used for execution, a thread identifier, an absolute task execution start and end time. The time values are absolute therefore the graph has to be acyclic otherwise the task that should be executed a second time would not be executed because the start time as well as the end time would have already exceeded.

The alternative unit to measure the length of a computation phase is the number of instructions that can be executed in the phase. In the following the unit time is used, because the implementation of a counter that measures the number of executed instructions is more complex than the implementation of a timer. Moreover, the time is independent of a system, while the number of instructions is always determined based on a specific CPU architecture. Therefore, a mapping is necessary in order to convert instruction numbers between CPU architectures. The decision to use time does not restrict the results that are based on the *plan*, because instructions can be converted to time and vice versa.

Such a *plan* is created based on the instructions of the HPC application that should be executed. The *plan* starts with an initial *task* and this *task* ends (the end time of the *task* has to be set to the corresponding time) if a system call should take place. The next *task* is created if this system call execution is ended (the start time of the *task* has to be set to the corresponding time). This handling is used to create all *tasks*. The main idea behind this handling is that a system call implies that an external action is triggered which has to be completed before the application execution can proceed. While the action takes place other threads can be executed.

The VRM has the purpose to allow the processing of multiple jobs in one HPC System without reserving nodes exclusively. Therefore, a newly created *plan* has to be combined with an existing *plan* that is already processed by the system. This results in a *plan* that is not necessarily a static construct that is predetermined once and never changed. In fact the *plan* will be extended if an additional job is accepted by the system. Furthermore, the *plan* has to be adjusted in the case of a prediction error.

This *plan* which is created specifically for one HPC System is divided into multiple *scheduling plans* that are node specific. In the following the *scheduling plan* is always assumed to be given and static, since the analysis of the *scheduling plan* creation would go beyond the scope of this thesis. Furthermore, for simplification the *scheduling plan* is always modeled with relative time values. The time span between the start and the end of a *task* is termed *execution time* (used as node weight in the TPG), because this time is allocated for the purpose of executing the corresponding *task*. The time between the end of a *task* and the start of the next *task* is termed *unallocated time* (used as edge weight in the TPG). The term *unallocated time* is chosen because the time is not allocated in order to execute *scheduling plan tasks*. Therefore, the time can be used to execute arbitrary *scheduling plan* independent threads.

As described at the beginning of this section, the *plan* generation depends on the instructions of the application that should be executed. Furthermore,

the environment that should execute the *scheduling plan* could have its own constraints that restrict properties of the *scheduling plan*. This is likely because most of the systems are designed to avoid the starvation of threads and the execution of a long *task* could be interpreted as a stuck thread. In order to identify such constraints it is necessary to analyze the thread scheduler which has to process the *scheduling plan*.

2 Assessment Criteria on Scheduling Plans

Current Operating Systems (OS) that are used on High Performance Computing (HPC) nodes are using the common scheduling policies that are designed to be as fair as possible in order to maximize the usability for desktop users. The advantage is that users can execute multiple threads (as e.g. X-Server and an office application) at the same time and no thread starves. The requirements of HPC Systems are totally different, because the objective is to minimize the execution time of the HPC applications. Therefore, it is necessary to implement a new scheduler that executes a node specific *scheduling plan*.

Beneath the precise execution of the *scheduling plan* it is necessary to harmonize the *PB Scheduler* that executes the *scheduling plan* and the default scheduler of the underlying OS. This is necessary, because the OS still has to execute system threads with the default scheduler in the *unallocated times* of the *scheduling plan*. The implementation in the following sections has to show whether the switching between the default scheduler and *PB Scheduler* is always possible.

The harmonization of the *PB Scheduler* and the default scheduler depends on the degree of filling of its corresponding runqueues. In the following the structure, that contains threads that are in a state that allows the corresponding scheduler to run it, is called runqueue even if the underlying structure is not a queue. First of all the filling of a queue is considered to be binary, either filled or empty. If the *PB Scheduler* runqueue is empty a further analysis of the filling of the default scheduler runqueue is not needed, because the system runs as without the *PB Scheduler* and no harmonization is necessary. The same holds for the case where the *PB Scheduler* runqueue is filled, but the default scheduler runqueue is empty, because the default scheduler has no threads to execute there is no conflict of interest between both schedulers.

The only case where the harmonization is necessary is if the default scheduler runqueue is filled as well as the *PB Scheduler* runqueue. The *PB Scheduler* has to process the *scheduling plan* as precisely as possible in order to comply with the deadlines. Therefore, the harmonization has to start within the creation of the *scheduling plan*. Hence the stability of the OS depends on the length of the *execution times* and the *unallocated times*, because the *unallocated time* is the time that can be used by the default scheduler and the length of the *execution time* specifies how long the default scheduler is blocked by the *PB Scheduler*.

Therefore, the *task execution times* and *unallocated times* have to be set to a value that guarantees the stability of the system, which executes the

scheduling plan. In the following a *scheduling plan* with such a guarantee is termed a *safe plan*. The most important criterion is that the *scheduling plan* is created so that it is possible that the scheduler can comply with all deadlines. Furthermore, it is useful to minimize the makespan[1] of the *scheduling plan* in order to execute the job as fast as possible. Therefore, an optimal *scheduling plan* is a *safe plan* with a minimal makespan where decrementing any *unallocated time* would lead to an *unsafe plan*. Such a *scheduling plan* is termed *maximal plan*. In order to verify whether a *scheduling plan* is safe, corresponding criteria are required. Furthermore, it is necessary to define the properties of a stable or an unstable system.

2.1 Criteria of a Stable System

A fundamental assumption is that the OS without the *PB Scheduler* and a common system load would always run stable. A possible source that could cause an unstable system could be a not executed system thread.

Depending on the OS there are drivers that are not called as a kernel method, but executed by a separate worker thread. As described in Sec. 1.1.2 a new *task* in the graph is created if the action triggered by a system call is executed. When the *scheduling plan* is processed, it is possible that the corresponding system call spawns a separate driver kernel thread that has to be executed in order to provide the system state that is expected by the next *task*. Therefore, the *unallocated time* between these both *tasks* has to be at least the estimated time to execute the driver thread. Otherwise these kernel threads could starve which should be avoided in a stable system. In order to avoid the starvation of user threads it is necessary to analyze the common system load of an HPC node.

2.2 Analytical Approach of Criteria Definition

In order to define the constraints of a *safe plan* it is necessary to define an allowed domain of *scheduling plan* parameters. Plans can be restricted by the parameters: maximal/minimal *execution time*, maximal/minimal *unallocated time*.

2.2.1 Minimal Task Execution Time

There is no minimal *task execution time*. Even if it is possible that the *execution time* is zero it is more useful to remove such a *task* from the *scheduling plan*. Nevertheless there are technical constraints that avoid an arbitrary small *execution time*, because a switch to or from the execution phase is not always possible and detectable. Therefore, the *execution time*

[1] The total length of the *scheduling plan*.

should be either zero or greater than the delay that is necessary to detect such a switch.

2.2.2 Maximal Task Execution Time

In order to verify whether the OS itself has limitations of the *execution time*, an empirical analysis of this aspect is necessary. As described in Sec.1.1.2, a *task* starts and ends with a system call. Therefore a *task* represents a section of the corresponding application where only computational instructions are executed. In fact an HPC application comprises communication between the node in order to exchange results which are necessary for further computations. Therefore it is not realistic that an HPC application has *tasks* with arbitrary long execution times. Furthermore, the user threads can limit the *execution time* of a task. On a common HPC node setup there are a few administration and monitoring threads which are not part of the predetermined *scheduling plan*. There are multiple different monitoring tools, but most of them spawn a monitoring thread that has to be executed periodically in order to collect system data and send them to a central server. Based on these data the server decides whether the observed system entered an undesirable state. If these data are not sent, the node will be marked as not responding and the administrator will be informed. Beneath the formerly mentioned passive tools that aggregate and send system data, there are also active tools that try to recover the system from an undesirable state. Such a tool is a watchdog that also requires that its thread is executed periodically and if this does not happen a restart of the system is triggered. In order to classify the system as stable such threads have to be executed. The period is configured for each tool by an administrator. Therefore, the maximal task *execution time* (denoted as t_{max}) is the minimum period of these tools can be assumed to be given.

2.2.3 Minimal Unallocated Time

The minimal *unallocated time* depends on the time that is needed to execute all formerly mentioned periodical monitoring threads. Additionally there are potential kernel threads that are started at the end of a *scheduling plan task* execution. These threads are needed to fulfill I/O-tasks which are required to execute the next *scheduling plan task* properly. Therefore, the *unallocated time* has to be long enough to execute these threads. The *execution time* of the administration threads should be nearly constant (denoted with f_e) in each period, because the administration threads always have the same *task*. The time that is necessary to execute the driver kernel threads (denoted as f_i) depends on the *task* of the thread. Therefore, the analytical approach is not sufficient to determine the minimal *unallocated time* and an empirical estimation is vital.

2.2.4 Maximal Unallocated Time

The *unallocated time* is the time, where the default scheduling mechanism of the OS is used. The usage of the default scheduler should not have a temporal limitation otherwise the OS without the *PB Scheduler* modification would not run stable and one of the fundamental assumptions would not hold. Therefore, there is no maximal *unallocated time* limitation of the system. However, choosing arbitrary long *unallocated times* would violate the criterion of a minimal makespan. Therefore, the minimal *unallocated time* should always be chosen.

2.2.5 Criterion of a Safe Plan

If the empirical analysis can find a valid value for f_i and the OS limitation of the task *execution time*, the following limitations of a *scheduling plan* (as shown in Fig. 2) can be defined:

$$(\overset{n}{\underset{j=1}{\forall}} : t_j \leq t_{max}) \wedge (\overset{n-1}{\underset{j=1}{\forall}} f_j \geq f_e + f_i)$$

Figure 2: Scheduling plan with n tasks (t_i) and $n-1$ *unallocated times* (f_i).

Furthermore, the *unallocated time* depends on the precision of the *PB Scheduler* implementation. If the switch to the execution phase has a delay it is necessary to increase all *unallocated times* by an appropriate offset.

It is useful to augment the results of the analytical approach by results of an additional empirical analysis in order to determine the remaining variable values of the maximal *execution time* and the minimal *unallocated time*. Therefore, a minimal implementation of a *PB Scheduler* is needed to collect significant data.

2.2.6 Criterion of a Maximal Plan

A *maximal plan* is a *safe plan* with minimal *unallocated time* values. Therefore, the criterion of such a *scheduling plan* can be formalized based on the definition of a *safe plan*:

$$(\overset{n}{\underset{j=1}{\forall}} : t_j \leq t_{max}) \wedge (\overset{n-1}{\underset{j=1}{\forall}} f_j = f_e + f_i)$$

In the following the definition of a *maximal plan* is used less restrictive than defined before, because the *scheduling plan* is based on different prediction models and is processed by an OS which is a highly variable and complex system. Therefore, it is not useful to apply such a strict mathematical definition. Thus, a *scheduling plan* that has nearly minimal *unallocated time* values is treated as maximal.

3 Plan Based Scheduler

The previous section introduced the concept of a *plan* and how such the *plan* can be divided into multiple node specific *scheduling plans*. These *scheduling plans* should be used as precomputed schedules by an High Performance Computing (HPC) node. The purpose of this section is to verify the feasability of the *PB Scheduler* that uses the *scheduling plan* as schedule for *plan tasks* and also executes arbitrary threads in the *unallocated time* of the *scheduling plan* while complying with the deadlines of the *plan tasks*.

Therefore, the following subsection introduces the requirements of a *PB Scheduler*. Based on these requirements the subsequent subsection determines the ideal base Operating System (OS). The remaining subsections explain the scheduler of the OS and the necessary adjustments to create a *PB Scheduler*.

3.1 Requirements

The *PB Scheduler* has to schedule the threads as predefined in a *scheduling plan*. The major difference between the *PB Scheduling* approach and other schedulers is that the schedules are precomputed. This leads to the advantage that the scheduler has a basic decision logic. The disadvantage of this approach is that the schedule cannot be adjusted dynamically by the OS. Therefore, it is crucial that the *scheduling plan* is executed as accurate as possible in order to avoid exceeded deadlines. Besides the execution of *scheduling plan* threads it is also required to execute arbitrary system threads (as described in Sec. 2.2.2, e.g. administrative applications), which leads to the requirement to harmonize the default scheduler of the base OS and the *PB Scheduler*. Furthermore, it is important that the scheduled threads and the actions of the *PB Scheduler* are not preempted by another scheduler or thread. This scheduler has to be implemented into an existing OS that is representative for an OS used on HPC system nodes.

3.2 Choice of the Base Operating System

The scheduler should be implemented in an OS which is widely used on HPC Systems so that the OS specific effects are the same as on real HPC Systems. Otherwise the analysis of system specific properties in combination with the scheduler would be inconclusive. The TOP500 list of [5] ranks the fastest supercomputers[2]. The analysis of the current list showed that all listed systems use a Linux distribution as their node OS[3]. Even if different distributions are used[4] all of them are based on a version of the same kernel.

[2]This list contains only supercomputers that are registered by voluntaries. Therefore, this list is not necessarily complete.

[3]See App. A.1 for detailed information about the Operating Systems in the TOP500.

[4]See App. A.1 for a list of popular distributions.

In order to generate the most general analysis results, which are applicable to most of the distributions, the scheduler should be implemented in the current code of the main repository of the Linux kernel.

3.3 Architecture of the Linux Scheduler

The Linux scheduler (v4.13) has a basic and clear abstract design. This design does not become immediately evident by studying the kernel code, because the implementation of the design is distributed over multiple files and contains optimizations which complicate the basic understanding. Therefore, this section describes the abstract design of the scheduler.

The design of the Linux scheduler is different from the classical idea of a scheduler that has one policy (as e.g. Round-Robin or FIFO) for all threads. The Linux scheduler is designed to administrate multiple policies. Each policy is implemented in a submodule and has its own runqueue[5] and corresponding managing functions to organize the structure of the runqueue. The submodules are ordered by descending priority in a simply linked list. The scheduler iterates over the list of submodules and calls a submodule function which determines whether the currently selected module has a thread in its runqueue that should be executed. This function can also return a value that causes a restart of the scheduler loop over all submodules. This value should be returned if a lower prioritised module determined that a higher prioritized module has already a thread in its runqueue. If the function returns a thread, the iteration over the list terminates and the thread will be executed. The return value *NULL* indicates that the runqueue of the current module is empty.

The duration of a thread execution depends on the submodules with higher priority and the submodule managing this thread. If a rescheduling is triggered and the currently used submodule wants to execute the current thread longer, it is only possible if no higher prioritised submodule has a thread in its runqueue. If a submodule wants to stop the execution of its current thread it is possible that the module contains a definition of a function which is called always if the clock interrupt arises. This function can trigger a rescheduling that stops the execution of the current thread.

The list of available modules used by the the Linux kernel with disabled Symmetric Multiprocessing (SMP) mode[6] is shown in Fig. 3. The list starts with the Deadline module which is a modified implementation of the Earliest

[5] According to the Linux documentation the structure that contains runnable threads is called runqueue even if another structure than a queue is used.

[6] The SMP module list starts with the additional Stop module that schedules a CPU specific stop task. This task has the highest priority in the whole system and therefore it cannot be preempted.

Deadline First (EDF) policy. The subsequent Realtime module implements the First-In, First-Out (FIFO) as well as the Round-Robin (RR) policy. The most famous module is the Completely Fair Scheduler (CFS) module which is used as the default policy for each thread in Linux and has the goal to be as fair as possible. The last entry in the list is the Idle-Scheduler. This scheduler always guarantees a runnable thread in its runqueue[7]. Therefore, the scheduler loop described above will always terminate with a valid thread as a result.

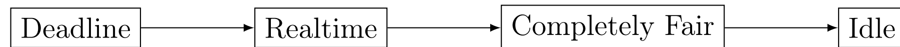


Figure 3: Order of scheduler submodules in the Linux kernel v4.13 with SMP disabled.

3.4 Implementation of the Linux Scheduler

This section first explains how threads are implemented in Linux, the subsequent section describes how the design described in the previous section is implemented in the kernel.

3.4.1 Representation of Threads in Linux

The most important object during scheduling is the thread. In order to understand the way Linux schedules threads it is useful to understand how threads are represented. A thread is represented by a pointer to a valid instance of the structure *task_struct*. This structure contains all data that are necessary to describe a thread. Contrary to other operating systems Linux represent threads with a private memory space (often termed process) and a shared memory space (often termed thread) with the same structure. Therefore, this is transparent for the scheduler.

The structure *task_struct* contains multiple fields that are not necessary to understand the general approach how the Linux scheduling implementation works. Therefore, this structure is not explained in depth and only the most important fields (in the context of scheduling) are explained.

The structure contains inter alia the thread identifier (*pid*) as well as multiple flags and priorities that are used by different scheduling modules. The most important field is the pointer *sched_class* that points to the scheduler module that is assigned to the specific thread. Furthermore, there is an integer *policy* that indicates which scheduling policy is selected for the thread. This is

[7]The returned thread does either housekeeping tasks or simply executes an assembly halt (`hlt`) instruction.

necessary because one module can provide multiple policies (as e.g. the RT module that provides the policies Round-Robin and FIFO). Therefore, the policy has to match with the scheduling module.

3.4.2 Implementation of the Architecture

The abstract scheduling functionality that loops through the submodules[8] is implemented in the procedure *__schedule* which is called by the main scheduling procedure *schedule*. As shown in Fig. 4 the implementation of the main logic is straightforward. The macro *for_each_class* loops through all submodules. Each submodule is represented by an implementation of the structure *sched_class*. Therefore, the iteration variable *class* contains a pointer to the currently selected *sched_class* implementation. Each iteration consists of a call of the function *pick_next_task* which receives the current runqueue (argument *rq*), the representation of the thread that was executed before the schedule process was triggered (argument *prev*) and some flags (argument *rf*). This is the function which determines whether the runqueue of the module contains a thread to run. If no thread is available the return value is *NULL* and if the module wants to restart the module loop the return value is the predefined pointer *RETRY_TASK*. The macro *unlikely* is just a branch prediction optimization and indicates to the compiler that the case is unlikely.

```
again:
  for_each_class(class) {
    p = class->pick_next_task(rq, prev, rf);
      if (p) {
        if (unlikely(p == RETRY_TASK))
          goto again;
        return p;
      }
  }
```

Figure 4: Main logic of the procedure *__schedule*.

One purpose of the structure *sched_class* is to work as an interface in the meaning of object-oriented programming and therefore provides a set

[8]Even if the abstract architecture suggests that the submodules are totally independent of each other and perfectly encapsulated, in fact even the general scheduling code contains submodule specific optimizations (e.g. the method *pick_next_task* contains module specific optimizations).

of functions that has to be implemented by a module. Furthermore, the structure models the simply linked list. A significantly shortened version of the structure is shown in Fig. 5. The pointer *next* points to the *sched_class* with the next lower priority and represents the link to the next list entry. The first entry of the list is defined via the macro *sched_class_highest*. The vast majority of the functions in *sched_class* are responsible to manage the runqueue. The function *task_tick* is always called by the clock interrupt handler. This function can be used to update time dependent priorities and to check whether the current thread has to be preempted. The procedure *enqueue* (or *dequeue*) should add a specific thread to (or remove it from) the runqueue. The procedure *put_prev_task* is crucial to ensure the correct scheduler state if a thread is preempted by a thread of another scheduler. If a scheduler preempts a thread of another scheduler, the method *put_prev_task* of the formerly used module is called with the preempted thread representation as an argument. Otherwise the thread representation would be lost and the runqueue of the formerly used module would enter an unexpected state[9].

```
struct sched_class {
  const struct sched_class *next;

  void (*enqueue_task) (struct rq *rq, struct task_struct *p,
          int flags);
  void (*dequeue_task) (struct rq *rq, struct task_struct *p,
          int flags);
  ...
  struct task_struct * (*pick_next_task) (struct rq *rq,
          struct task_struct *prev, struct rq_flags *rf);
  void (*put_prev_task) (struct rq *rq, struct task_struct *p);
  ...
  void (*task_tick) (struct rq *rq, struct task_struct *p,
          int queued);
  ...
};
```

Figure 5: Significantly shortened definition of the struct *sched_class* that represents the main interface of the implementation of a submodule. This extract shows only the definitions necessary to explain the implementation of the abstract design.

[9]First implementation attempts showed that if the method is not implemented, sporadic kernel panics are likely, because the former scheduler module is not correctly administrated.

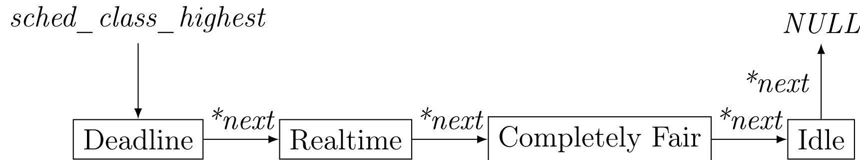


Figure 6: Technical representation of the scheduler submodule list with SMP disabled.

3.5 Architecture of the Plan Based Scheduler

In order to harmonize the *PB Scheduler* and the CFS (and Idle-Scheduler) the approach of scheduling different scheduling policies is chosen, because this approach is already implemented with the modular priority based design of the Linux scheduler which is designed via the module list. The scheduling policy as depicted in Fig. 7 of the *PB Scheduler* is as follows: The scheduler has to iterate over the *scheduling plan* list and executes the tasks exactly as long as specified in the *scheduling plan*. After the execution of a *task* the scheduler has to use the CFS (or Idle-Scheduler) in order to schedule threads which are not included in the plan. The default scheduler should only be used as long as specified in the *unallocated time* between two *tasks*. As soon as the *unallocated time* elapsed, the *PB Scheduler* should be used again.

These switches between the *task execution mode* and the *unallocated mode* of the *PB Scheduler* are termed mode switches. In order to execute the *scheduling plan* accurately it is crucial to detect necessary switches as soon as possible and to minimize the delay between the detection and the execution of a switch. This switch handling is only necessary as long as non-executed *tasks* are in the *scheduling plan*. If all entries are executed the system should use the default scheduler until a new *scheduling plan* is submitted. While no new *scheduling plan* is submitted the *PB Scheduler* is called disabled and submitting a *scheduling plan* would enable the scheduler.

The modular scheduler design of Linux is a great foundation in order to implement the formerly introduced handling, because the priority based scheduling of different scheduling policies is already implemented and can be adjusted so that the *PB Scheduler* and the already existing Linux scheduling policies are harmonized.

In order to ensure that the threads are not preempted the *PB Scheduler* has to be the module with the highest priority. Therefore, the *PB Scheduler* is the first module that is checked in the loop (described in Sec. 3.3) that determines the thread which will be executed next. If the *PB Scheduler* threads a *scheduling plan* and the current time slot is unallocated, another policy

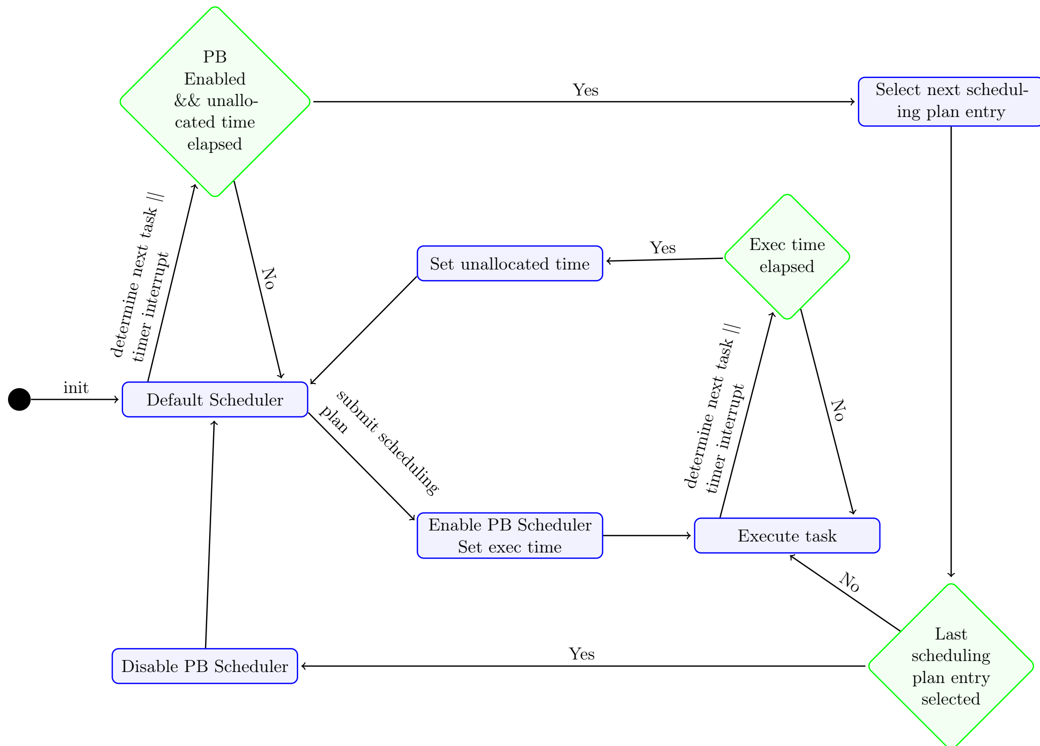


Figure 7: State Transition Diagram of the *PB Scheduling* approach on top of the scheduler (denoted as "Default Scheduler") of an arbitrary operating system.

can execute arbitrary threads. This switch can be implemented through the formerly mentioned loop. If the *PB Scheduler* module returns a thread, the remaining scheduling policies are ignored and the thread is executed, but if the module does not return a thread, the remaining modules (most important the CFS) are checked corresponding to its priority. Therefore, the first switch from the *PB Scheduler* to the CFS (or Idle-Scheduler) is just implemented through the existing architecture without adjusting other modules.

The switch from the CFS (or Idle-Scheduler) is necessary in the case that the *unallocated time* exceeded and an execution of a *scheduling plan task* has to start. This logic needs modifications in all modules[10] that are used beneath the *PB Scheduler*. If a module other than the *PB Scheduler* is used, it has to check cyclically during a thread execution whether the *unallocated time* is exceeded. If this happens the rescheduling process has to be triggered. The rescheduling leads to the execution of the thread selection loop. Therefore, the requirements of the *PB Scheduler* are implementable in the Linux archi-

[10]It is assumed that only the default scheduler CFS and the Idle Scheduler are used.

tecture without changing existing logic and the scheduler works as before if the *PB Scheduler* has no *scheduling plan* to process.

3.6 Implementation of the Plan Based Scheduler

The following sections describe an implementation approach in order to implement a *PB Scheduler*. For simplification it is assumed that the threads that have to be executed by the *PB Scheduler* are always runnable. This leads to a minimal thread management, but the implementation of a full handling would be similar to handling of other scheduling modules, because the Linux waiting queue implementation is generic and independent of the modules.

Therefore, the Sec. 3.6.1 introduces only data structures that represent the *scheduling plan* and corresponding administrative information.
The subsequent Sec. 3.6.2 describes how a method can be implemented that determines whether the *PB Scheduler* should be active or not. This method has to be used by all scheduler modules in order to determine whether they have to stop their execution and switch to the *PB Scheduler*.

The main logic of each scheduler is located in its module implementation. Therefore, the last Sec. 3.6.3 describes the implementation of a *PB Scheduler* module.

3.6.1 Enhancement of the Linux Runqueue

The common way to enhance the runqueue structure *rq* by new submodule specific information is to add a module specific struct instance. Therefore, an instance of the new structure *pb_rq* is added. As shown in Fig. 8 the plan based runqueue structure contains the *scheduling plan* and additional administrative information. The first four fields have to be set dynamically to initialize the scheduler, while the remaining fields are used internally by the *PB Scheduler* to persist its state. The array *plan* represents the *scheduling plan* that has to be executed. Each array entry is an instance of the structure *plan_entry* (also shown in Fig. 8) which consists of a time that indicates how long a task should be executed (*exec_time*) and the subsequent *unallocated time* (*uall_time*) which can be used by other less prioritized schedulers. According to the Linux scheduler time handling, the default time unit of the *PB Scheduler* is also a nanosecond.

The size of the current *scheduling plan* is specified via the variable *size*. The following definition of a runqueue has a maximal size defined via the macro *PB_MAX_PLAN_LENGTH* for simplification, but it is also possible to implement a dynamical runqueue. The variable *c_entry* is the iteration

variable that is used to loop over the *scheduling plan* and select the next *task*.

While the *scheduling plan* entries contain relative time information, the variable *exec_until* (or *uall_until*) contains the absolute time information until which timestamp the task should be executed (or the CFS/Idle-Scheduler can be used). The variable *mode* contains the value of one of three macros that indicate one scheduler mode[11]. The *disabled mode* indicates that the *PB Scheduler* already executed the current *scheduling plan* or that no *scheduling plan* is set. The remaining modes indicate whether the scheduler currently executes a *scheduling plan task* (*execution mode*) or the other scheduler modules are allowed to choose the thread which will be executed (*unallocated mode*).

```
struct plan_entry
{
  pid_t task_pid;
  u64 exec_time;
  u64 uall_time;
};

struct pb_rq
{
  struct plan_entry plan[PB_MAX_PLAN_LENGTH];
  unsigned int size;
  unsigned int current_entry;

  u64 uall_until;
  u64 exec_until;
  int mode;
};
```

Figure 8: Run queue implementation.

The *PB Scheduler* runqueue structure is initialized with the default values via the method *init_pb_rq*. This method is called by the main runqueue initialization method that initializes the main runqueue before the scheduler is used the first time. This initialization does not set a *scheduling plan*. In order to use the *PB Scheduler* it is necessary that a system call is implemented

[11]The macro *PB_DISABLED_MODE* indicates the *diabled mode*. The macro *PB_EXEC_MODE* indicates the *execution mode* and the macro *PB_UALL_MODE* indicates the *unallocated mode*.

that can be used to submit a new *scheduling plan*. This system call sets the new *plan*, the size variable and resets the other variables to its initial value.

3.6.2 Determination of the Current Scheduling Mode

The verification whether a mode switch should take place is needed multiple times. The *PB Scheduler* has to check whether a *scheduling plan task* still has to be executed and the CFS (and also the Idle-Scheduler) has to check whether its thread has to be preempted by a *scheduling plan task*. In order to avoid code duplication and to decouple the *PB Scheduler* logic from the CFS and Idle-Scheduler, the mode switch detection is moved into a separate method *determine_next_mode_pb*. This method determines the next *PB Scheduler* mode. This decision is made based on the current time and the data of the *PB Scheduler* runqueue. Therefore, this method is a vital part of the *PB Scheduler* business logic.

As shown in Fig. 9 the scheduler should be disabled if the *scheduling plan* size is exceeded. The values of *exec_until* and *uall_until* are only set by the code that implements the mode switch. The mode switch detection only reads these variables. If the scheduler is currently in the *execution mode* and the *execution time* is elapsed then a switch to the *unallocated mode* should take place. The logic of the switch from the *unallocated mode* to the *execution mode* is similar.

3.6.3 Implementation of the Plan Based Scheduler Module

Based on the assumption that all *scheduling plan* related threads are runnable when they should be executed, only a minimal thread handling is necessary. Therefore, the module implementation of *sched_class* contains multiple stub implementations in order to fulfill the interface. The implemented methods that contain business logic are *pick_next_task* and *task_tick*.

3.6.3.1 Implementation of *pick_next_task*

During the scheduling process the elementary method of the module is the *pick_next_task* method. In the *PB Scheduler* the main goal of this function (termed *pick_next_task_pb*) is to set the next mode and the corresponding variables according to the current mode and the result of *determine_next_mode_pb*. As shown in Fig. 10 the behavior of the scheduler should be unchanged if the determined mode is equal to the current mode. Therefore, *NULL* is returned if the *unallocated mode* or the *disabled mode* is set and the thread representation corresponding to the current *task* is returned if the *execution mode* is set.

If the mode changes from the *disabled mode* or *unallocated mode* to the

```
static inline int determine_next_mode_pb(u64 time,
       struct rq *rq)
{
  int mode = PB_DISABLED_MODE;
  struct pb_rq *pb = &(rq->pb);

  if (pb->c_entry < pb->size)
  {
    if (pb->mode == PB_EXEC_MODE)
    {
      mode = (pb->exec_until < time)
               ? PB_UALL_MODE
               : PB_EXEC_MODE;
    }
    else if (pb->mode == PB_UALL_MODE)
    {
      mode = (pb->uall_until < time)
               ? PB_EXEC_MODE
               : PB_UALL_MODE;
    }
  }
  return mode;
}
```

Figure 9: Method that determines the next mode.

execution mode, the variable *uall_until* is reset to zero. Subsequently the absolute *execution time* deadline *exec_until* is calculated from the sum of the relative *execution time* of the current *scheduling plan* entry and the current time. Since the execution of the *task* thread should start immediately, the corresponding thread representation has to be returned. The logic that is used if the mode switches from the *execution mode* to the *unallocated mode* is similar and the new *unallocated time* is calculated analogously. Additionally the iteration variable *c_entry* has to be incremented and checked whether the *scheduling plan* size is exceeded. If this happens the scheduler needs to be disabled, because the whole *scheduling plan* is processed. If a mode switch is executed the generic method *put_prev_task*[12] has to be executed in order to enqueue the potentially preempted thread of another scheduling module in its runqueue. This method calls the procedure *put_prev_task* of the formerly used scheduler module.

[12]The method has the same name as the module specific method, but is a generic module independent method.

3.6.3.2 Implementation of *task_tick*

If the task thread is running, it is necessary to check cyclically whether the execution time is elapsed. This is implemented through the method *task_tick_pb* which is the *PB Scheduler* implementation of the *sched_class* method *task_tick*. This function simply checks whether a mode switch is required and triggers the reschedule process by calling the method *resched_curr*. This method sets the flag *TIF_NEED_RESCHED* on the currently executed thread which causes a rescheduling as soon as possible. The same handling is added to the *task_tick* method of the CFS and the Idle-Scheduler. Therefore, the main schedule method will be called and the mode switch takes place in the method *pick_next_task_pb*.

A similar handling as in the *task_tick* method is implemented in the *pick_-next_task* method of the CFS (and the Idle-Scheduler) with the difference that instead of calling *resched_curr* the macro *RETRY_TASK* is directly returned as the result of the method. This causes a restart of the scheduling loop and the *pick_next_task* method of the *PB Scheduler* is executed and the mode switch takes place. The main goal of this implementation is to minimize the delay between the mode switch detection and the execution of the switch.

In order to implement the *PB Scheduler* as the module with the highest priority it is necessary to set the corresponding module as the first element of the scheduler module list. The first element of the list is defined by the macro *sched_class_highest* that points to the static instance of *pb_sched_class*. Furthermore, the *next* pointer has to point to the next lower module, which is the module implementation of the Deadline-Scheduler that previously had the highest priority.

```
static struct task_struct * pick_next_task_pb(struct rq *rq,
    struct task_struct *prev, struct rq_flags *rf)
{
  struct task_struct *picked = NULL;
  u64 now;
  int current_mode, next_mode;
  struct pb_rq *pb = &(rq->pb);

  now = sched_clock();

  current_mode = pb->mode;
  next_mode = determine_next_mode_pb(now, rq);

  if (current_mode == next_mode) {
    if (current_mode == PB_EXEC_MODE)
      picked = find_task_by_vpid(pb->plan[c_entry].task_pid);
  }
  else {
    if ((current_mode == PB_DISABLED_MODE ||
        current_mode == PB_UALL_MODE)
        && next_mode == PB_EXEC_MODE) {
      pb->mode = next_mode;
      pb->uall_until = 0;
      pb->exec_until = pb->plan[pb->c_entry].exec_time + now;
      picked = find_task_by_vpid(pb->plan[c_entry].task_pid);
    }
    else if (current_mode == PB_EXEC_MODE &&
        next_mode == PB_UALL_MODE){
      pb->mode = next_mode;
      pb->uall_until = pb->plan[pb->c_entry].uall_time + now;
      pb->exec_until = 0;
      pb->c_entry++;

      if (pb->c_entry >= pb->size) {
        pb->mode = PB_DISABLED_MODE;
      }
    }
    put_prev_task(rq, prev);
  }
  return picked;
}
```

Figure 10: Method *pick_next_task_pb*.

3.7 Implementation of a Plan Based Scheduler Prototype

In order to prove the concept of a *PB Scheduler* it is not necessary to implement all aspects mentioned in the former section. Therefore, in the following the differences between the full featured *PB Scheduler* and the minimal prototype scheduler are highlighted. This prototype is used to prove the feasibility of the *PB Scheduler* concept and the formerly proposed implementation approach.

3.7.1 Simplified Thread Handling

As proposed as a naive approach to implement a *scheduling plan* structure, the prototype implementation models a *scheduling plan* as an array with a maximal length that is predefined in the macro *PB_MAX_PLAN_LENGTH* but the actual size is stored in the variable *size*. This definition is used instead of a static *scheduling plan* definition because setting the *scheduling plan* dynamically simplifies the testing of the *PB Scheduler* prototype.

The prototype needs no real thread handling, only a single *proxy thread* is required, because in order to verify the feasibility of the *PB Scheduler* concept it is not significant which task the *scheduling plan* threads are executing. This *proxy thread* is represented by the pointer *proxy_task*. This pointer has to be set dynamically to a thread representation, because the pointer is not set during the initialization process of the scheduler[13]. Therefore, the entries of the *scheduling plan* array do not own a thread identifier as proposed in the former section.

In Sec. 3.6.2 the method *determine_next_mode_pb* is proposed in order to determine the current mode of the scheduler. The same logic is used in the prototype, but in the case that the scheduler is still disabled but a *proxy task* is set, the execution of a new *scheduling plan* has to be started with setting the *PB Scheduler* to the *execution mode*. Therefore, the *determine_next_mode_pb* function of the prototype also handles this case, which is not necessary in a full featured *PB Scheduler*.

3.7.2 Dynamical Plan Initialization

During the accomplishment of empirical analysis multiple *scheduling plans* have to be processed by the prototype. In order to test these *scheduling plans* as fast as possible it is necessary to avoid a recompilation process of the kernel for each *scheduling plan*. Therefore, the whole *scheduling plan* structure is designed to be dynamical (until an upper *scheduling plan* size of

[13] It is not possible to set the *proxy_task* pointer during the initialization, because the corresponding thread has to be created. During the creation of the thread, the scheduler code is used which is not completely initialized.

PB_MAX_PLAN_LENGTH is reached). Additionally there are methods to set the size of a new *scheduling plan* and to set a new entry of the *scheduling plan*. Contrary to the fully featured *PB Scheduler* the prototype does not use a system call to set a *scheduling plan*. The main idea to make the *scheduling plan* configurable is to use the Linux module to submit a new *scheduling plan*. The standard kernel implementation does not allow that a module executes any arbitrary kernel method. Therefore, the required methods and symbols are exported[14].

The implementation of a Linux module can contain an init and an exit method. The init method is executed when the module is added to the kernel (e.g. via the shell command *insmod*) and the exit method is executed when the module is removed from the kernel (e.g via the shell command *rmmod*). Therefore, a module that sets a *scheduling plan*, also has to set the *proxy task* in its init method and to reset all values to the default value in its exit method. This handling allows to test different *scheduling plans* one after another. This is done by adding a module, waiting until the *PB Scheduler* prints the debug message "PLAN DONE", removing the module and doing the same with the next module.

Furthermore, the module can be generated automatically for each *scheduling plan* definition, because only the *scheduling plan* definition part differs from module to module. See App. A.3.1 for a full example module and App. A.3 for the explanation, how the modules can be generated. The disadvantage of this approach is that the module has the requirement that the header files of the current kernel version are installed. Therefore, it is necessary to build them and install them in the test environment which is a time-consuming task.

3.7.3 Kernel Thread Execution Time Measurement

The previous kernel changes were made to demonstrate the feasibility of implementing a *PB Scheduler* and to determine a maximal *task execution time*. Furthermore, it is necessary to determine a minimal *unallocated time* value that consists of the execution time of threads that depend on user specific configurations and the execution time of kernel threads that have to be executed to prepare the execution of the next task. In order to measure the kernel thread execution the kernel has to be extended.

The main idea of the following extension is to check what kind of thread is picked to be executed next (called *next*) and what kind of thread is currently executed (called *prev*). Therefore, each thread can be either a kernel or

[14] The macro *EXPORT_SYMBOL* can be used in the kernel in order to make a symbol accessible for modules.

non-kernel thread. If both threads are non-kernel threads, then no behavior change is needed. If the previous thread is a non kernel thread and the next is a kernel thread then the measurement of the kernel thread time should start because the kernel thread will be executed immediately. If both threads are kernel threads then no behavior change is needed because it is assumed that a previous case already started the measurement. If the previous thread is a kernel thread and next thread is a non-kernel thread, the measurement has to stop. Subsequently the time of the previous measurement has to be added to the whole measurement result. It is not necessary to embed this logic in the *PB Scheduler*, but since this code base is well known, extending its structures is easy. Therefore, the *PB runqueue* is extended as shown in Fig. 11 by one integer *measure_k* which is treated as a boolean and indicates whether the kernel thread execution time measurement is enabled or disabled. The remaining variables are needed to save times. The variable *start* saves the time the measurement is started. *ktime* contains the accumulated kernel thread execution time while *kstart* saves the time at which a kernel thread starts to execute.

```
struct pb_rq {
...
  int measure_k;
  u64 kstart;
  u64 ktime;
  u64 start;
};
```

Figure 11: Extended PB runqueue structure.

The measurement functionality is implemented in the main scheduling method *__schedule*, because this method is always called to execute the scheduler and therefore has up-to-date information about the formerly executed thread as well as the next thread. The pointer *prev* points to the *task_struct* that represents the formerly executed thread and *next* points to the *task_struct* that represents the thread that will be executed. At the end of *__schedule* the *next* thread will be executed immediately. In order to check whether a thread is a kernel thread it is necessary to check whether the thread has the flag *PF_KTHREAD*. The idle thread is also a kernel thread and it would be necessary to remove the idle time from the kernel thread execution time. Therefore, the method *is_kthread* also checks that the thread is no idle thread (via checking the flag *PF_IDLE*). A new measurement phase starts with setting the variable *kstart* to the current time if the execution of a kernel thread starts and the previous thread was a non-kernel thread. If the next thread is a non-kernel thread, but the previous thread was a kernel thread, the

difference between *kstart* and the current time is added to the measurement result in the variable *ktime* and *kstart* is reset.

```
...
  next = pick_next_task(rq, prev, &rf);

  if (prev != next &&
      rq->pb.measure_k == PB_MEASURE_K_ON &&
      is_kthread(prev->flags) != is_kthread(next->flags))
  {
    if (is_kthread(prev->flags) &&
        rq->pb.kstart > 0)
    {
      rq->pb.ktime += sched_clock() - rq->pb.kstart;
      rq->pb.kstart = 0;
    }
    else if (is_kthread(next->flags))
    {
      rq->pb.kstart = sched_clock();
    }
  }
...
```

Figure 12: Small extract of the *__schedule* method that contains the kernel thread execution time measurement logic.

3.7.4 Results

The first small tests showed that the prototype implementation of the *PB Scheduler* as described before works as expected, on hardware as well as in a virtual environment[15]. The integration of the adjusted kernel into a Lubuntu distribution is easy. As required the whole system works as before if no *scheduling plan* is submitted. If a *plan* is submitted and the *execution time* begins only the corresponding thread is executed. Therefore, the whole Window Manager[16] (in the case of QEMU the terminal) freezes during the *scheduling plan execution times* and is only available during the *unallocated times* and if the *scheduling plan* is processed.

In order to verify whether the switches from the *execution mode* to the *unallocated mode* work precisely a test series is executed. During the test

[15]See App. A.2 for more information.
[16]Further tests are executed with a deactivated X environment.

a *scheduling plan* is submitted that consists of 100 tasks with an execution time and an *unallocated time* of $3s$. The first test series runs in a system with a low system load and the second series runs in a system with high load[17]. As depicted in Fig. 13 the test series with a low system load has a higher mean deviation of the *unallocated time* than the test results with a high system load. This difference suggests that the switch from the Idle-Schedulder to the *PB Scheduler* has to be improved in order to reduce the deviation. The deviation is in the positive value range therefore the switch from the *unallocated mode* to the *execution mode* takes place after the start deadline of the next task already has exceeded. This has to be avoided, because executing the task one millisecond shorter than defined could cause serious issues because the system call that should complete the *task* execution is not called. Otherwise executing a task thread longer than defined should not cause problems, because the execution of the system call changes the corresponding thread state and the thread waits for an event and the thread is not runnable. Therefore, a reschedule and a switch would take place in a fully featured scheduler.

In order to move the values from the positive range to the negative range it is useful to subtract an offset[18] from the start deadline in order to move the deviation into the negative area. As shown in Fig. 13 this optimization in combination with a high system load leads to the desired result. The deviation of the *execution time* is negligible, but the value will be higher in practice, because the *proxy thread* is a kernel thread that has nearly no business logic. Therefore, the thread always triggers a rescheduling and the scheduler can verify whether a switch is necessary. In practice a user thread will be interrupted and the timer interrupt method *task_tick* has to verify whether a switch is necessary. Therefore, the handling will be similar to the handling of the switch from the *unallocated mode* to the *execution mode.*

This test series showed that the *PB Scheduler* prototype is able to comply with the execution time deadline and also with the *unallocated time* deadline, even if optimizations are still necessary. Therefore, the most important assessment criterion defined in Sec. 2 is met. The scheduler is able to comply with all deadlines if the *execution time* and *unallocated time* values are as specified by the *plan* constraints.

[17]Triggered by the stress testing tool *stress*.

[18]In order to reduce deviations the kernel is compiled with the 1000 Hz option. Therefore, in the period of $1000000ns$ timer interrupts should occur. It is useful to choose a multiple of the period. Therefore, 2500000 is used as offset.

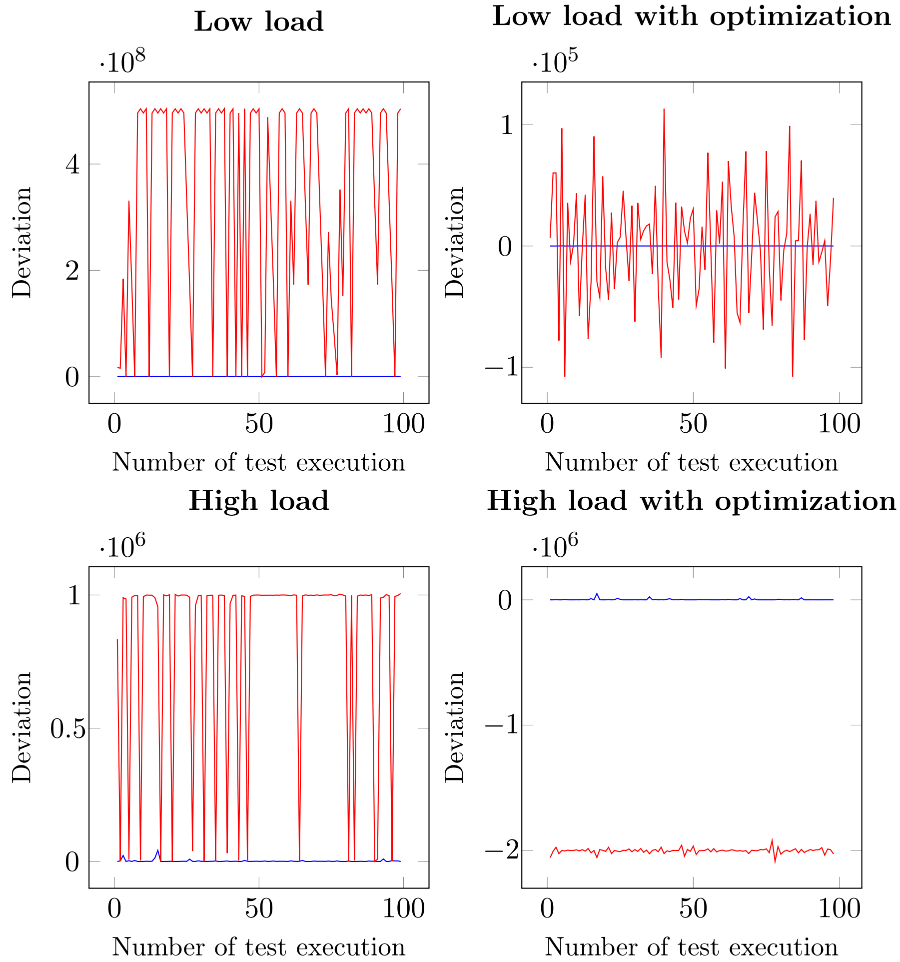


Figure 13: Measurement of the deviations of the *execution time* and *unallocated time* value. The red graph represents the deviation that occurs during the switch from the *unallocated mode* to the *execution mode*. The blue graph represents the deviation that occurs during the switch from the *execution mode* to the *unallocated mode*. Negative values indicate that the switch takes place before the corresponding deadline is exceeded and positive values indicate that the switch takes place after the corresponding deadline is exceeded.

4 Empirical Analysis

The previous section proved the feasibility of the *PB Scheduler* concept. Based on this knowledge and the prototype implementation described in Sec. 3.7 this section analyses Operating System (OS) dependent constraints of the *scheduling plan* and therefore completes the results of Sec. 2.2.

4.1 Analysis of the Maximal Task Execution Time

The analytical approach of Sec. 2.2.2 is not sufficient to determine OS specific limitations of the *task execution time*. However, it determines an administrator defined upper boundary which is the minimum of the periods of user threads in which these threads have to be executed. Therefore, the purpose of this analysis is to clarify whether an OS dependent limitation exists that decreases the formerly found upper boundary of *task execution times*.

The first test with a task that should be executed for one minute failed, because the *proxy thread* is a kernel thread and the internal Linux watchdog (not to be confused with the software watchdog daemon) softlockup detector causes a kernel panic. This handling is made to detect hanging and defective driver or kernel code. Therefore, a non-prototype scheduler would not have this problem. After disabling the watchdog softlockup handling[19] a *task* with an *execution time* of one minute executed successfully and the system behaved after the execution as before.

In order to create a more realistic test scenario a stress test tool runs before, during and after the execution of the *task*. The system works as before the execution and does not cause an unstable state in the OS. The same test results were reproducible with the same test setting for *tasks* executing twenty minutes and sixty minutes[20].

The results of the tests showed that even if the execution time is restricted by the OS, the upper boundary of the *execution time* is greater than sixty minutes, which is assumed to be an unrealistic long time for a *task*. Therefore, it is assumed that the OS has no limitations concerning the execution time of a task.

4.2 Analysis of the Minimal Unallocated Time

An *unallocated time* slot in a *plan* is always created if the corresponding High Performance Computing (HPC) application wants to communicate via a system call (as described in Sec. 1.1.2), because the start of a system call is

[19] Disabled during runtime via the shell command *sysctl kernel.soft_watchdog=0*

[20] All tests are repeated four times.

the end of a *task* execution and the end of the system call action is the start of the next *task* execution. In order to make sure that the corresponding kernel thread that realizes the communication already ran if the next *task* started, it is necessary to determine the required execution time of the kernel thread. This value can be estimated by analyzing the needed kernel thread execution time of a common representative HPC application. In order to determine the time used by a kernel thread, the kernel is extended as described in Sec. 3.7.3. Another approach is to determine the execution time of an application and the time executed in user mode. The difference between these values is also the used kernel time.

The measured applications are a distributed version of a prime number generator that determines all prime numbers until an upper boundary and a distributed matrix multiplication. Both applications use MPI for message passing. Each application ran with two or four threads and four different input values. The input value of the prime number generation is the upper boundary of the greatest prime number that has to be generated. The matrix multiplication receives the size of the quadratic matrices that have to be multiplied.

The Fig. 14 shows the measurement results of the prime number generation application. The results of the matrix multiplication application are depicted in Fig. 15. The great outlier values that lead to the great error are not produced by the HPC application which is executed, but by other threads (e.g. a cron job)[21]. Therefore, the most interesting value is the mean value which should be smaller than the plotted value because the outlier leads to a higher mean. Even with this higher mean value an upper boundary of the percentage is always 0.5% and if it is assumable that high input values are used a percentage of 0.2% would lead to a *safe plan*. Therefore, an *unallocated time* that consists of 0.5% (or 0.2%) of the whole runtime plus the time that is required by the administrative threads should lead to a *safe plan* and therefore a stable system.

4.3 Results

Based on the results of the former analysis as well as the results of Sec. 2.2 it is possible to construct a *maximal plan*. For this purpose the variables of the *maximal plan* definition have to be analyzed:

$$(\underset{j=1}{\overset{n}{\forall}} : t_j \leq t_{max}) \land (\underset{j=1}{\overset{n-1}{\forall}} f_j = f_e + f_i)$$

As shown by the analytical approach the only Linux specific thread execution time restriction is that the time should be greater than the period between

[21] This analysis is discussed in detail in App. A.4.3.

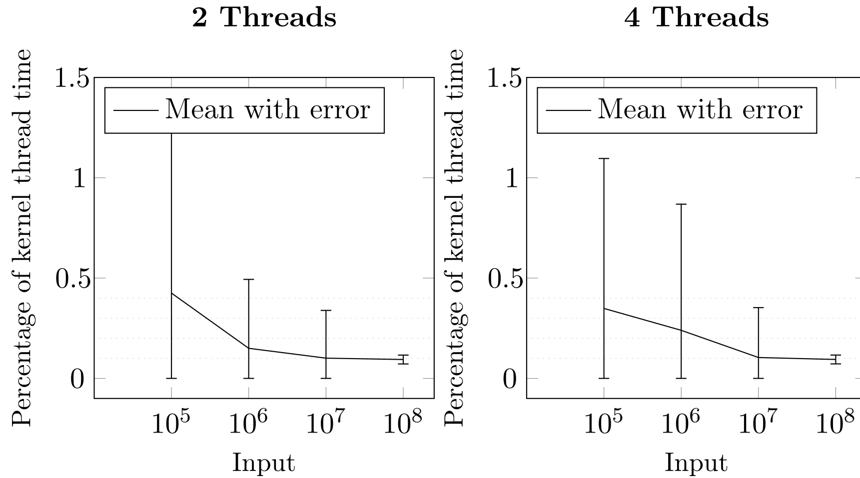


Figure 14: Results of measuring the kernel thread time of the prime number application. The x axis shows the input and the y axis the percentage of kernel thread time in the whole runtime. Since each measurement ran 100 times the error is also shown as a bar.

two timer interrupts that determine whether a mode switch should take place. Furthermore, the *execution time* is restricted by the period of execution of administration and monitoring tools. Thus, the maximal *task execution time* t_{max} of the *scheduling plan* depends on the configuration of the system that is assumed to be given.

The kernel thread time of the *unallocated time* analyzed by former test is approximately 0.2% of the whole execution time but the test also showed that this value is variable. Therefore, there is no general purpose formula and the percentage that affects f_i has to be determined by the administrator who also determines f_e, because the *unallocated times* depends on the configuration of the system.

Furthermore, it is very likely that the kernel execution time f_i is much smaller than the execution time of administrative threads f_e. It is assumed that the administrative threads are executed in each *unallocated time*, therefore the kernel execution time becomes insignificant compared to the execution time, because the kernel execution times are very small and f_e consists of the execution time of multiple threads which aggregate results and send them via the network, which is a costly task. If this assumption holds the only constraints of a *scheduling plan* is based on configurations of the administrator. Therefore, it would be possible to create a *maximal plan* by choosing the maximal *execution time* as upper boundary of the *execution time* and choosing the execution time of all tools in a period as *unallocated time*.

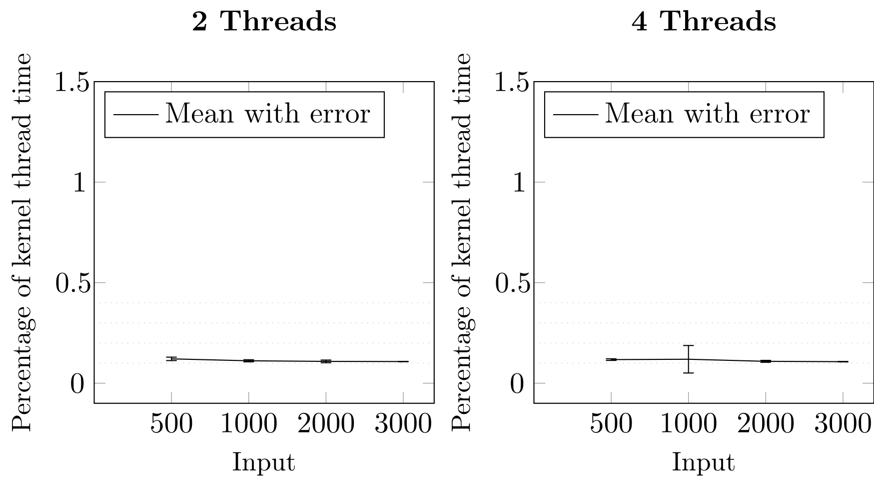


Figure 15: Results of measuring the kernel thread time of the matrix multiplication application. The x axis shows the input and the y axis the percentage of kernel thread time in the whole runtime. Since each measurement ran 100 times the error is also shown as a bar.

5 Conclusion

The main objective of this thesis is to analyze the feasibility of a *PB Scheduler* in the de facto standard High Performance Computing (HPC) node operating system Linux. Therefore, it is necessary to create a scheduler that fulfills the contrary requirements of the *PB Scheduler* and the CFS. Instead of creating one scheduler policy that fulfills these requirements, the developed scheduler (introduced in Sec. 3.5) switches between the *PB Scheduler* and the CFS. This can be considered to be a scheduler of scheduler policies. Therefore, the concept of *PB Scheduling* is proven to be applicable in a real Operating System (OS). At the beginning of this work it was not clear whether OS depending limitation would make it impossible to implement a scheduler that fulfills the requirements of the *PB Scheduler* as well as the requirements of the CFS. The prototype implementation of the scheduler also showed that the modular Linux scheduler architecture is very suitable for a *PB Scheduler* implementation.

The second objective of this thesis is to analyze whether OS and system dependent constraints exist which restrict the structure of *scheduling plans* that can be executed with the *PB Scheduler*. At the beginning of this work the extreme values of the *execution time* and *unallocated time* value were identified as parameters that could be restrictive factors. The analytical approach already showed that neither the minimal *execution time* nor the maximal *unallocated time* are limited. The later empirical analysis showed that the maximal *execution time* is only limited by the periodical administration threads which have to be executed in a predefined interval. The period of this interval is the upper boundary of the *execution time*.

The minimal *unallocated time* is also restricted by a lower boundary which depends on the execution time of the administration and monitoring threads which are configured by the administrator. There are also other kernel threads which have to be executed but the execution time is assumed to be negligible because it is likely that this time is much smaller than the execution time of the monitoring threads.

Therefore, the structure of the *scheduling plan* is hardly restricted and even the only restrictions of the *scheduling plan* are configurable by the administrator. Thus, it is possible to create a *maximal plan* that guarantees the stability of the system during its processing and has a minimal makespan. Furthermore, the Sec. 3.7.4 showed that it is possible that a *PB Scheduler* complies with all deadlines. Therefore, all assessment criteria that are defined in 2 are fulfilled.

6 Future Work

This thesis is a basic analysis of the *PB Scheduler* approach and its impact on the *scheduling plan* that has to be executed. This forms a foundation for many possible research topics and implementations. In the following a non-exhaustive list of possible enhancements and topics is presented.

6.1 Design of a Scheduler for SMP Systems

One assumption that simplifies the whole thesis is that the used node has only one core. In fact in a modern High Performance Computing System (HPC System) each node has multiple cores. In order to maximize the performance of each node it is necessary to be able to use all cores. Therefore, it can be useful to analyze whether the harmonization of the CFS and *PB Scheduler* can be adjusted on an Symmetric Multiprocessing (SMP) node.

A naive approach is to reserve one core for threads with the policy CFS and the other cores are reserved for threads that are part of the *scheduling plan*. Such an approach could minimize restrictions of the *unallocated times*, but it would be necessary to analyze whether other resources have to be taken into account, because of multiple instances which could access them. Furthermore, the approach to reserve a whole core for the CFS could be too restrictive.

6.2 Implementation of a Fully Featured Scheduler

The implementation of Sec. 3.6 only proves that the concept of a *PB Scheduler* works, but it is not a complete scheduler, because several features are not implemented. If the implementation of the *PB Scheduler* should become a part of the Linux kernel it is necessary that the code complies with the Linux community standards as the coding style guide.

6.2.1 Enhancement of the Scheduling Functionality

In order to use the *PB Scheduler* a fully featured version has to be implemented as described in Sec. 3.5. Thus, it is also necessary to implement a thread handling that allows to organize and execute arbitrary *scheduling plan* threads. Furthermore, the scheduler has to handle all possible thread states in an appropriate way. In practice it is possible that the *task* that has to be executed still waits for an event. The prototype implementation simply ignores these thread states.

6.2.2 Implementation of the User Interface

In Sec. 3.7 the kernel is adjusted so that a new *scheduling plan* can be submitted via a Linux kernel module. This solution is not usable in a

productive scenario, because important scheduling structures are exported for that solution and this would empower Linux modules to modify the scheduling process which is not intended. Therefore, this solution is not recommended and would not be accepted by the Linux community. Furthermore, the common user interface to scheduling modules are system calls. Therefore, it is useful to implement a system call for that purpose.

6.2.3 Design and Implementation of an Error Handling

This thesis always considers that the *scheduling plan* will be executed correctly and all deadlines will be met, but the *scheduling plan* is based on different prediction models that can create inaccurate deadlines. Therefore, it is vital to design and implement an error handling that defines how the system reacts if a deadline is exceeded. If one deadline is exceeded it is reasonable to ensure that the subsequent deadlines are still met and the deadline of the whole job will be met.

Depending on the severity of the problem there are multiple possible solutions. The easiest solution is to ignore the problem, proceed with the execution and assume that the deviation from the expected deadline to the actual execution time is so small that another prediction deviation will counterbalance the other deviation. Depending on the solution of Sec. 6.1 and on whether the subsequent *task* is a *task* of another thread it is possible that an idle CPU can be used to execute the next *task*, but this could lead to a greater delay, because switching threads from one CPU to another is expensive since the cache of the target CPU has to be filled with the required data which is costly. If the deviation of the deadline is high it can be necessary to escalate the problem to the next higher instance which is the RMS. The RMS can try to solve the problem or it escalates to the next higher instance which is the ADC. It is possible that the best possible result is that a delay has to be accepted by the VRM while other jobs that run on the same system are still in time. The worst case would be that the delay in one job would cause delays in other job executions. This domain is complex and therefore still requires much research.

A Appendix

A.1 TOP500 Statistics

Operating System Family	Count	System Share (%)	Cores
Linux	500	100	69,004,640

Table 1: Statistic of the Operating System Families in the TOP500 (see [5]).

Operating System	Count	System Share (%)	Cores
Linux	267	53.4	17,942,148
CentOS	109	21.8	26,702,802
Cray Linux Environment	46	9.2	5,976,520
SUSE Linux Enterprise Server 11	16	3.2	1,166,516
bullx SCS	11	2.2	749,752
TOSS	10	2.0	496,584
RHEL 7.2	5	1.0	196,580
RHEL 7.3	5	1.0	92,448
Scientific Linux	4	0.8	123,128
Bullx Linux	3	0.6	204,000
Ubuntu Linux	3	0.6	142,344
Redhat Enterprise Linux 6.5	2	0.4	105,216
Ubuntu 14.04	2	0.4	72,224
SLES12 SP2	2	0.4	89,856
SUSE Linux Enterprise Server 12 SP1	2	0.4	280,728
Kylin Linux	2	0.4	3,294,720
RHEL 6.8	2	0.4	46,336
Redhat Enterprise Linux 6.4	2	0.4	68,906
Redhat Enterprise Linux 6	2	0.4	295,656
SUSE Linux	1	0.2	153,216
Sunway RaiseOS 2.0.5	1	0.2	10,649,600
Redhat Enterprise Linux 7.2	1	0.2	31,968
RHEL 6.2	1	0.2	46,208
bullx SUperCOmputer Suite A.E.2.1	1	0.2	77,184

Table 2: Statistic of the Operating Systems in the TOP500 ordered by the number of usages in HPC systems (generated by the TOP500 statistics tool, see [5]).

A.2 Development and Test Infrastructure

The development of OS kernel features is less suitable for unit tests. Therefore, a development environment that allows to test the features as fast as possible is necessary. During the main development phase a virtual solution is used. In the later testing process a real Linux distribution is used in order to use an environment that is similar to a real HPC node.

A.2.1 Virtualization

In order to minimize the time of a development cycle the virtualization tool QEMU is used in combination with an image containing a Debian root file system[22]. Therefore, the only variable input of QEMU is the compiled kernel. Contrary to the realistic setting with a common Linux distribution the system load of the QEMU system is very low, because the system configuration is minimal and has no X environment or cron jobs. In order to execute the tests in a steady system state, the stress testing tool *stress* is used before the tests to have a more realistic state of the runqueue of the CFS.

The QEMU environment runs on the following hardware:

Component Name	Component Description
Processor	4 x Intel(R) Core(TM) i5-2520M CPU @ 2.5 GHz
Memory	4GB DDR3 @ 1600 MHz
OS	Lubuntu 16.10

Table 3: Hardware that is used to execute QEMU.

A.2.2 Hardware

In order to test in a more realistic setting the modified kernel is built into debian package files. These files are used to install the modified kernel into a Lubuntu. The installation and testing is much more time-consuming than the virtual solution, but the test results are more representative. During the real test executions the whole X environment is disabled.

Lubuntu runs on the following hardware:

Component Name	Component Description
Processor	4 x AMD A8-6500 APU with Radeon(tm) HD Graphics @ 3.5 GHz
Memory	4GB DDR3 @ 1333 MHz
OS	Lubuntu 16.10

Table 4: Hardware that is used to execute the adjusted kernel.

A.3 Linux Module Generation

As mentioned in 3.7.2 linux modules are used to trigger the execution of a certain *scheduling plan*. In order to avoid the copy and paste task to create a module for each *scheduling plan*, a *scheduling plan* is modeled as a CSV file with maximal 100 rows where each line contains two values. The first value

[22]The root file system is created with the tool *debootstrap* and has Debian version 8 (Jessie).

indicates the *task execution time* and the seconds one the *unallocated time* after the execution.
After the kernel is compiled, a perl script converts all *scheduling plans* in a certain directory to Linux modules and creates one Makefile to build all module. After all modules are compiled, all corresponding modules files (with the extension *.ko*) and the *scheduling plan* modules are copied into the Debian image. If the image booted with the current kernel, all modules can be tested with a perl script, that also aggregates and evaluates all results using the module files.

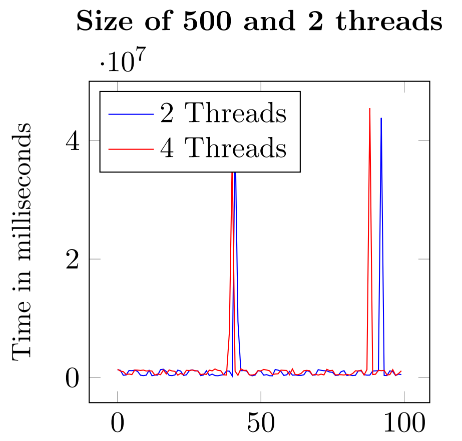
A.3.1 Linux Module to set a Scheduling Plan

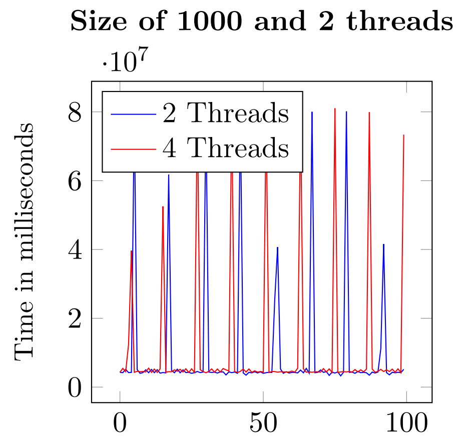
```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/kthread.h>
#include <../kernel/sched/sched.h>


MODULE_LICENSE("GPL");

static int proxy_thread_func(void *data)
{
        unsigned int c = 0;
        while (!kthread_should_stop()) {
                int a = 0;
                set_current_state(TASK_INTERRUPTIBLE);
                for (;a < 200000; a++){}
                schedule();
                c++;
        }
        return 0;
}

static void init_rq(struct pb_rq *pb_rq)
{
    set_pb_plan_size(pb_rq, 2);
    // execute time of 20min, unallocated time of 2s
    set_pb_plan_entry(pb_rq, 0, 1200000000000, 2000000000);
    // execute time of 1s, no unallocated time (last entry)
    set_pb_plan_entry(pb_rq, 1, 1000000000, 0);
}

static int __init pb_client_init(void)
```

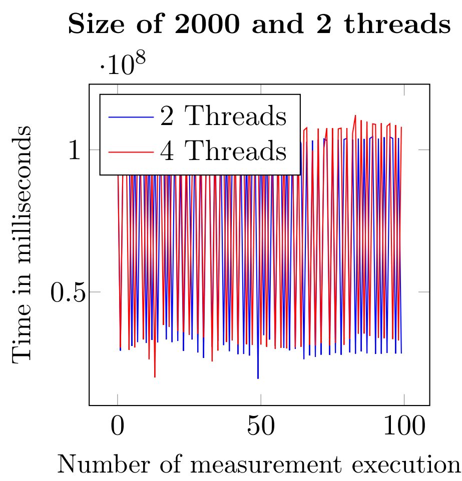
```
{
    struct task_struct *proxy_task;
    struct rq *rq;

    proxy_task = kthread_create(proxy_thread_func, NULL,
      "PB proxy thread");
    proxy_task->sched_class = &pb_sched_class;

    rq = this_rq();
    init_rq(&rq->pb);

    rq->pb.proxy_task = proxy_task;

    return 0;
}

static void __exit pb_client_cleanup(void)
{
    struct rq *rq;
    rq = this_rq();

    // set pb_rq back to initial values
    init_pb_rq(&rq->pb);

    printk(KERN_DEBUG "Cleaning up module.\n");
}

module_init(pb_client_init);
module_exit(pb_client_cleanup);
```

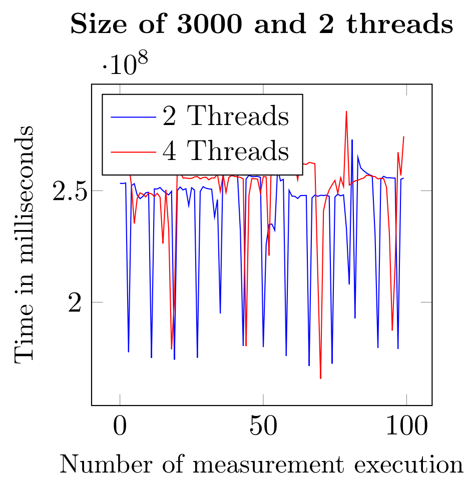
A.4 Measurement Results

A.4.1 Matrix Multiplication Results

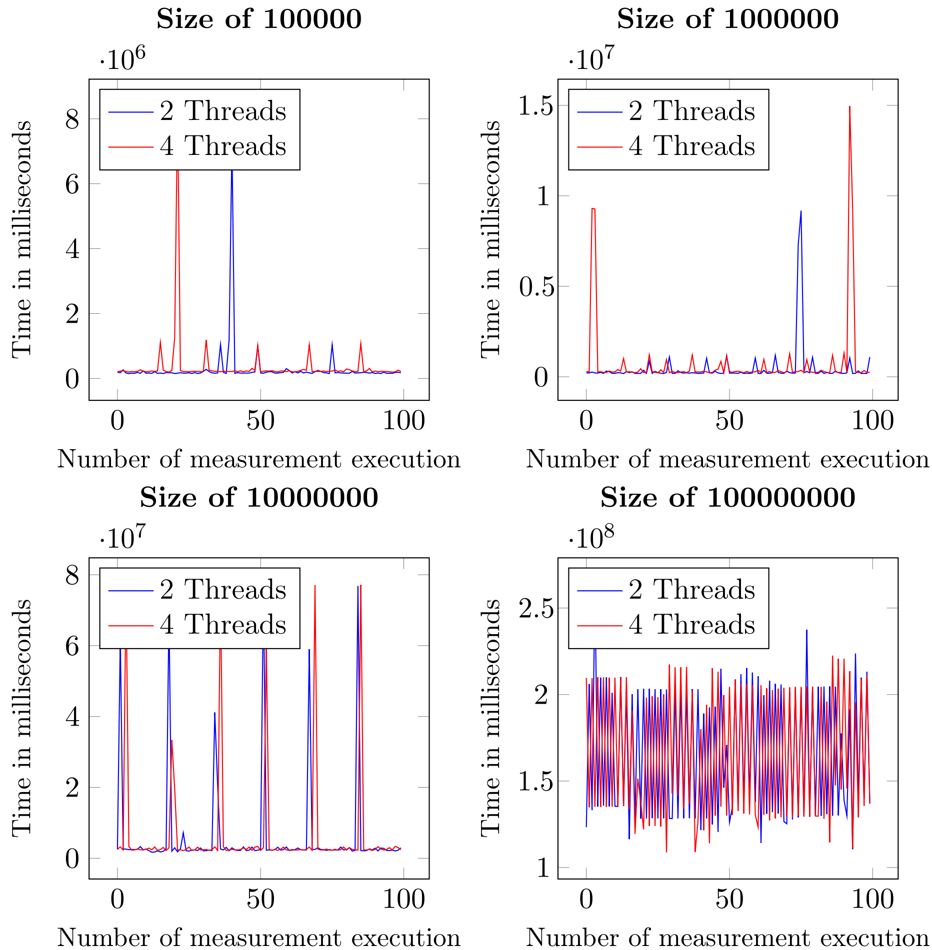








A.4.2 Prime Number Generation Results



A.4.3 Result Analysis

The measurement results contain huge error values. A great example of a dataset containing such outliers are the results of the measurement of the prime number generator with the input value of 100000. The maximal percentage is 10.158 while the mean is 0.425 which is near to the minimum of 0.220. Therefore, it was necessary to analyze the reason for those outlier values. The following analysis has the focus on the dataset of the matrix multiplication with a matrix size of 1000 and two threads. The plot of the non-aggregated 100 measurement results in Fig. 16 already shows a suspicious development of the kernel runtime over the whole measurement process. The measurement executions are triggered by a script and therefore the time that elapsed between two executions should be nearly constant. The plot shows that the kernel runtime value always elevates after a fixed time period elapsed. Since the single executions of the measurement are independent of each other

the time dependent behavior has to be rooted in the test system. The analysis of the raw measurement results containing the system timestamps shows that time that elapsed between two extremly elevated values is on average 120.676 seconds. This suggests that at least one cron job with a period of two minutes exists that is also executed on the test system. The other datasets have similar outlier values that can be reduced to other threads that are executed periodically on the test system.

The results of this analysis are also applicable to the other results because the same periodical behavior of the runtime values appeared in them even if they are not as distinct as in the analyzed result set.

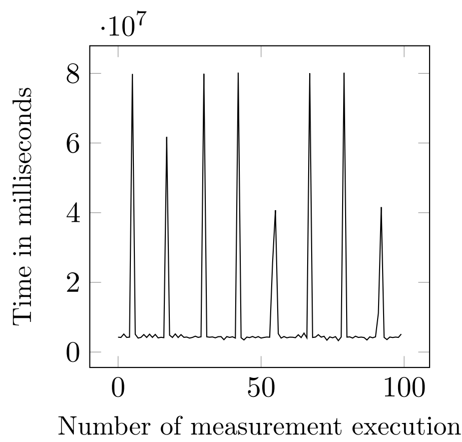


Figure 16: Results of the matrix multiplication with input 1000 and two threads.

References

[1] I. Foster and C. Kesselman, Eds., *The grid: Blueprint for a new computing infrastructure*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, ISBN: 1-55860-475-8.

[2] H. Bal, C. Laat, S. Haridi, K. Jeffery, J. Labarta, D. Laforenza, P. Maccallum, J. Masso, L. Matyska, and T. Priol, "Next generation grid (s) european grid research 2005–2010," Jan. 2003.

[3] L. O. Burchard, M. Hovestadt, O. Kao, A. Keller, and B. Linnert, "The virtual resource manager: An architecture for sla-aware resource management," in *IEEE International Symposium on Cluster Computing and the Grid, 2004. CCGrid 2004.*, Apr. 2004, pp. 126–133. DOI: `10.1109/CCGrid.2004.1336558`.

[4] M. Hovestadt, O. Kao, A. Keller, and A. Streit, "Scheduling in hpc resource management systems: Queuing vs. planning," in *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 1–20, ISBN: 978-3-540-39727-4.

[5] E. Strohmaier, J. Dongarra, H. Simon, M. Meuer, and H. Meuer. (2017). Top 500 list of the 500 fastest supercomputers, [Online]. Available: `https://www.top500.org/list/2017/11/` (visited on 11/29/2017).

List of Figures

1 Architecture of an AD. 4
2 Scheduling plan with n tasks (t_i) and $n-1$ *unallocated times* (f_i). 11
3 Order of scheduler submodules in the Linux kernel v4.13 with SMP disabled. 15
4 Main logic of the procedure *__schedule*. 16
5 Significantly shortened definition of the struct *sched_class* that represents the main interface of the implementation of a submodule. This extract shows only the definitions necessary to explain the implementation of the abstract design. 17
6 Technical representation of the scheduler submodule list with SMP disabled. 18
7 State Transition Diagram of the *PB Scheduling* approach on top of the scheduler (denoted as "Default Scheduler") of an arbitrary operating system. 19
8 Run queue implementation. 21
9 Method that determines the next mode. 23
10 Method *pick_next_task_pb*. 25
11 Extended PB runqueue structure. 28
12 Small extract of the *__schedule* method that contains the kernel thread execution time measurement logic. 29
13 Measurement of the deviations of the *execution time* and *unallocated time* value. The red graph represents the deviation that occurs during the switch from the *unallocated mode* to the *execution mode*. The blue graph represents the deviation that occurs during the switch from the *execution mode* to the *unallocated mode*. Negative values indicate that the switch takes place before the corresponding deadline is exceeded and positive values indicate that the switch takes place after the corresponding deadline is exceeded. 31
14 Results of measuring the kernel thread time of the prime number application. The x axis shows the input and the y axis the percentage of kernel thread time in the whole runtime. Since each measurement ran 100 times the error is also shown as a bar. 34
15 Results of measuring the kernel thread time of the matrix multiplication application. The x axis shows the input and the y axis the percentage of kernel thread time in the whole runtime. Since each measurement ran 100 times the error is also shown as a bar. 35
16 Results of the matrix multiplication with input 1000 and two threads. 45