Virtual Machine Scheduling in Dedicated Computing Clusters

Dissertation
for attaining the PhD degree
of Natural Sciences

submitted to the Faculty of Computer Science and Mathematics
of the Johann Wolfgang Goethe University
in Frankfurt am Main, Germany

by Stefan Boettger
born in Torgau, Germany

Frankfurt 2012
(D30)

accepted by the Faculty of Computer Science and Mathematics of the

Johann Wolfgang Goethe University as a dissertation.

Dean:	Prof. Dr. Thorsten Theobald
Expert assessor:	Prof. Dr. Udo Kebschull
	Prof. Dott. Ing. Roberto V. Zicari
Date of disputation:	

Time-critical applications process a continuous stream of input data and have to meet specific timing constraints. A common approach to ensure that such an application satisfies its constraints is over-provisioning: The application is deployed in a dedicated cluster environment with enough processing power to achieve the target performance for every specified data input rate. This approach comes with a drawback: At times of decreased data input rates, the cluster resources are not fully utilized. A typical use case is the HLT-Chain application that processes physics data at runtime of the ALICE experiment at CERN. From a perspective of cost and efficiency it is desirable to exploit temporarily unused cluster resources. Existing approaches aim for that goal by running additional applications. These approaches, however, *a*) lack in flexibility to dynamically grant the time-critical application the resources it needs, *b*) are insufficient for isolating the time-critical application from harmful side-effects introduced by additional applications or *c*) are not general because application-specific interfaces are used. In this thesis, a software framework is presented that allows to exploit unused resources in a dedicated cluster without harming a time-critical application. Additional applications are hosted in Virtual Machines (VMs) and unused cluster resources are allocated to these VMs at runtime. In order to avoid resource bottlenecks, the resource usage of VMs is dynamically modified according to the needs of the time-critical application. For this purpose, a number of previously not combined methods is used. On a global level, appropriate VM manipulations like hot migration, suspend/resume and start/stop are determined by an informed search heuristic and applied at runtime. Locally on cluster nodes, a feedback-controlled adaption of VM resource usage is carried out in a decentralized manner. The employment of this framework allows to increase a cluster's usage by running additional applications, while at the same time preventing negative impact towards a time-critical application. This capability of the framework is shown for the HLT-Chain application: In an empirical evaluation the cluster CPU usage is increased from 49% to 79%, additional results are computed and no negative effect towards the HLT-Chain application are observed.

Acknowledgements

For inspiration and guidance on the road to this thesis I thank my professor Mr. Kebschull. Both valuable feedback and daily life companionship from my colleagues Timo Breitner and Jochen Ulrich also helped me in mastering this task. Special thanks goes to my colleague Camilo Lara whom I owe much for professional advice and fruitful discussions. My parents I thank for all their unconditional trust and support. Heinke, thank you for your understanding and sincerity.

Contents

List of Figures

List of Tables

1. Introduction

Today, computer clusters play an important role in scientific and commercial computing. A cluster is a confederation of interconnected computing devices ("nodes"). It represents a single computer system and is typically deployed in order to make use of the combined processing power of its constituting nodes. Most commonly, a cluster consists of commodity hardware nodes which are loosely connected by a Local Area Network (LAN) and spatially located within a single computing center. Clusters have a wide range of applicability that involves scientific simulations (climate research, fluid dynamics), modeling (computer aided design, solid modeling) and image processing (medical imaging, oil reservoir imaging). Computing tasks ("applications") which run in a cluster can be distinguished by timing constraints that apply to their desired processing performance:

- A best effort application processes readily available, persistently stored input data and has no explicit timing constraint. Such an application can be re-run if a failure has occurred without suffering from degraded result or service quality. Simulations and modeling applications are typical examples.
- A time-critical application processes a stream of input data and has to satisfy a timing constraint concerning processing performance. Typical timing constraints are: Maximum response time per user request, number of data units processable per second or number of customers servable per day. If a timing constraint is not satisfied, the service or result quality is degraded. Examples are web based applications (web servers, online shops), applications for stock trading or applications that process data in high-energy physics experiments.

A typical problem in the context of time-critical applications is to ensure that an application's timing constraint is always satisfied. A widely accepted approach to this problem is "over-provisioning". In simple terms this means buying more computing power than needed in the worst expectable case. This approach is adopted by hardcore gamers[1] that buy a desktop computer according to the criterion that next year's version of their favourite game has to run smoothly. In cluster computing, this means to tailor the cluster environment explicitely to the application, i.e. to design the cluster in such a manner that a specific maximum application performance will be achieved. Such a specifically tailored cluster environment is called a "dedicated cluster". The stream of input data for an application in a dedicated cluster is not necessarily constant, i.e. both frequency and size of to be processed data units may vary over time.[2] For instance, a web server hosting a sports ticketing software will experience increased user request rates before a tournament and an instant decline thereafter. Such a variation in the input data rate translates to a variation in the resource usage of the dedicated cluster. Running a cluster is cost-intensive and a temporarily decreased resource usage means wasting precious computing power. Therefore, a nearly optimal resource usage is desirable. The question arises as to how unused resources in a dedicated cluster can be exploited without affecting the time-critical application.

[1]Hardcore gamer is a colloquial term for a person who spends significant time and money on playing specific computer games such as 3D shooters.

[2]For ease of understanding, the size of to be processed data units is assumed to be constant. The metric indicating the amount of data to be processed within a specific time frame is called "data rate".

A use case which belongs to this category and inspired this thesis is the cluster based processing of detector data during a particle physics experiment: Conseil Européen pour la Recherche Nucléaire (CERN) is a research facility that operates a particle accelerator and collider, the Large Hadron Collider (LHC) [67]. The LHC is an underground-built ring of 27 kilometers in circumference. It accelerates protons and heavy ions to 99.9999991% of the velocity of light and brings these particles to collision at specific interaction points in the ring. A Large Ion Collider Experiment (ALICE) [1] is an experiment located at one of these interaction points and is optimized for studying heavy ion collisions. Its main purpose is to explore the Quark-Gluon-Plasma (QGP) that is supposed to be forming at high energy density and temperature conditions. The ALICE experiment consists of multiple detectors and processing elements that gather and process collision data. One of these processing elements is the ALICE High Level Trigger (HLT). Basically, the HLT is a dedicated cluster that runs a time-critical application, the HLT-Chain [104]. The HLT-Chain is a distributed and hierarchical application for evaluation, selection and compression of collision data at runtime of the experiment. According to specification, the HLT-Chain has to be capable of processing a maximum input data rate of 25 GByte/s. The cluster hosting this application ("HLT-Cluster") comprises of about 200 commodity hardware nodes and 25 infrastructure servers. The resource usage in the HLT-Cluster is varying: There are alternating run / no-run phases for the ALICE experiment. While collision data has to be processed during a run phase, no processing occurs during a no-run phase.[3] A run phase can last from minutes to hours, a no-run phase can last from minutes to months. The data rate can be different for separate run phases due to a modified filling scheme of the LHC and a modified trigger scheme of the ALICE experiment. During a single run phase, the data rate also changes due to decreasing beam quality ("luminosity"). Additionally, the HLT-Chain application may be reconfigured during a run phase. This leads to a spatial reallocation of HLT-Chain processes in the HLT-Cluster and consequently to a changed resource usage on the concerned cluster nodes. Hence, there is variation in the resource usage of the HLT-Cluster and temporarily unused resources exist.

Unused resources in a dedicated cluster can be exploited by running additional applications. While this can lead to an increase in cluster resource usage and result output, it also poses several challenges:

Software Incompatibility: An additional application requires a suitable Operating System (OS) configuration. If the application was compiled against a specific kernel type that is not compatible with the cluster configuration (Windows application / Linux cluster, 64bit application / 32bit OS), considerable administrative overhead is required to assure the running of such an application in the cluster.[4] Incompatibilities may also occur on another level: Different versions of shared libraries might be required by the time-critical and additional applications. Such a version conflict may prevent an application from being started or cause it to fail at runtime. Since this is inacceptable for the time-critical application, special care has to be taken when trying to install an additional application in dedicated cluster.

Security Issues: During installation of an additional application, new software packages are merged into a dedicated environment. This may introduce security risks which can be exploited by attackers. This not only concerns intentionally designed malicious code, but any sufficiently complex software with application and user interfaces. Exploits may compromise the dedicated cluster setup and lead to disclosure of confidential data, degraded application performance or even a break in the functionality

[3]The cluster is used for occasional testing and calibration during a no-run phase.

[4]For instance, this could be achieved by having a dual-boot node configuration, enabling to switch the running OS upon reboot of a node.

of the time-critical application.

Software Bugs: An additional application may suffer from software bugs. Infamous bugs are programming errors like memory leaks, synchronization faults (deadlocks, race-conditions) and unintendedly created infinite loops. Common to bugs is to cause an application to behave differently than expected or specified. Some bugs only emerge occasionally or with subtle effects, e.g. a faulty calendar mapping in a leap year or a periodically occurring minor computation error. Other bugs are more severe: An application may crash or even affect the OS environment as a whole. For example, the memory leak of an application may consume a great share of a node's memory, which eventually affects the performance of other running applications.[5] The risk of introducing software bugs needs to be taken into account when running an additional application.

Competition for Resources: In a dedicated cluster, all resources (Central Processing Unit (CPU), network, memory etc.) are at sole disposal of the time-critical application and dimensioned such that the timing constraint can be satisfied. By running an additional application, resources will be shared among the additional application and the time-critical application. If a certain resource is fully utilized (e.g., the CPU on a node), there will be competition for this resource between these applications. This may affect the performance of the time-critical application. Two scenarios exemplify this effect:

- Given that the input data rate for the time-critical application is below the expectable maximum data rate. Then there probably are unused resources, e.g. idle CPU cycles on a node. If an additional application is now started on this node, CPU shares are attributed to this application in a fair manner, i.e. the CPU usage is split between the time-critical and the additional application. This might cause the additional application to receive more CPU cycles than were idle before, hence the additional application will steal CPU cycles from the time-critical application.[6] Such stealing may affect the time-critical application by decreasing its performance. Therefore, the question is how to give an additional application only that share of resources that is unlikely to decrease the performance of the time-critical application.
- Given that an additional application is already running in the cluster and only uses otherwise idle CPU cycles. Now the input data rate for the time-critical application is suddenly increasing. Then the amount of resources needed by the time-critical application also rises, i.e. the amount of resources safely usable by the additional application decreases. If the additional application does not release a share of its used resources, the time-critical application might suffer from a decreasing performance. The question is how to dynamically adjust the resource share given to an additional application according to the changing needs of a time-critical application.

These challenges not only apply to the CPU usage on nodes, but are valid also for other resources, e.g. the network. Additionally, there may be more than one additional application. This eventually gives rise to the question as to how the competition for resources between a time-critical application and multiple additional applications can be coordinated such that the time-critical application still meets its timing constraint.

[5]A memory leak enforces the OS to heavily make use of swap space or running the out-of-memory emergency routine. This decreases the performance of running applications.

[6]The fair-share scheduling policy is the default setting for today's Linux and Windows OS process schedulers. Both may be influenced by manually setting a process priority. This setting, nevertheless, is not sufficient for ensuring that a process does not use more than a certain desired ratio of CPU cycles. For more information, please refer to Linux documentation [71].

This thesis presents the conceptual approach and implemention of a framework which addresses these challenges and allows to run additional applications in a dedicated cluster with a time-critical application. By using platform virtualization for additional applications, a sufficient isolation between the time-critical application and additional applications is provided. Thus, security issues are avoided and the effects of software bugs towards the time-critical application are eliminated. Platform virtualization also broadens the scope of runnable additional applications by providing a specifically configured environment for each application. This method renders obsolete software incompatibilities and library version conflicts.
The core challenge of this thesis is how to run additional applications without causing resource competition that leads to a violation of the timing constraint of the time-critical application. In this thesis, a concept is developed that provides a dynamic resource allocation for every Virtual Machine (VM) hosting an additional application: Temporary resource bottlenecks are anticipated and counteracted by allowing VMs only to use resources for a limited time frame if the total usage of a resource exceeds a certain value. The share of resources used by VMs is modified at runtime. The dynamic resource allocation for VMs according to the needs of a time-critical application is realized on two levels: A Global Scheduler residing on a management node decides on using virtualization product features like VM hot migration, suspend/resume and start/stop in order to prevent or resolve resource bottlenecks. A Local Controller residing on each worker node uses a feedback controller to dynamically adjust the share of resources like CPU and network usable by local VMs.
The presented approach is distinctive and novel by combining two features. First, the framework relies on application-agnostic, OS provided metrics only and thereby is not tied to the utilization for a specific time-critical application. Second, it combines multiple methods for adapting the resource usage of additional applications: A developed search heuristic chooses between different coarse-grained, globally initiated VM state transitions to adapt the resource usage at runtime. In parallel, a fine-grained, decentralized adaption of VM resource usage is carried out. The feasibility of the approach is shown for the HLT-Chain application. However, the applicability of the framework is not tied to HLT-Chain context. Configuration items (policies) will be presented that allow to customize the framework towards dedicated clusters running other time-critical applications. By employing the presented framework it is possible to exploit unused resources and to increase the result output of dedicated clusters by running additional applications without affecting the time-critical application. This framework, though explicitely designed for dedicated clusters with a time-critical application, is not restricted to employment in such environments. The conceptual approach and implementation is feasible also for exploiting unused resources and increasing the result output of clusters running multiple time-critical applications or applications without timing constraints.

The thesis is structured as follows:
Chapter 2 states the goal of this work. Chapter 3 discusses the context of this thesis and portrays methods used for dynamic resource allocation in clusters. Furthermore, a review of competing approaches is given. In Chapter 4 the conceptual work of this thesis is described: The basic terminology is introduced, conceptual decisions are explained and the functionality of the decision-making infrastructure is laid out in detail. Chapter 5 gives insight into implementational details and user interfaces. Empirically retrieved results are presented in Chapter 6. These include performance tests with simulated workload and tests for the HLT-Chain application. In Chapter 7 the particular approach and the achievements of this thesis are summarized. Additionally, an outlook for future work is presented. Finally, the Appendix details several marginal aspects and provides a description of the used abbreviations.

2. Thesis Goal

Over-provisioning is a paradigm for dimensioning a computing facility according to a required peak performance. For a cluster that runs a time-critical application, this requires to design the cluster in a manner that the timing constraint of the application will be satisfied for every specified data input. Such a cluster explicitely tailored to the needs of a specific application is called a dedicated cluster. If the input data rate for the time-critical application is varying over time, then the dedicated cluster is likely to be underutilized at times of low data rates. This thesis aims at exploiting unused resources in a dedicated cluster by running additional applications. For this purpose, a software framework needs to be developed that realizes this task. The following conditions constrain the behaviour of this framework:

1. The proper functionality of the time-critical application must not be affected by additional applications.
2. The framework cannot control the time-critical application. That is, the time-critical application is a black-box and the framework can only passively react to the behavior of this application.[1]
3. The framework improves the efficiency of the cluster such that it increases *a*) the cluster usage[2] and *b*) the number of computed results per time.

The provided framework has to fulfil the stated goal for the HLT-Chain application. Throughout this thesis, the term Main Application (MainApp) will be used to refer to the time-critical application, for example the HLT-Chain. An additional application that exploits unused resources via the provided framework is called a Secondary Application (SecApp).

[1]This excludes contrasting approaches that are not valid for the HLT-Chain use case, e.g. steering the time-critical application using its own proprietary Application Programming Interface (API) or using hypervisor commands for e.g. migrating virtualized components of the time-critical application.

[2]The term cluster usage refers to the share of CPU cycles used in a cluster averaged over time and nodes.

3. Context and Related Work

The goal introduced in Chapter 2 explicitely demands to run additional applications in a dedicated cluster. The underlying general question is that of how to give resource consumers, e.g. additional applications, access to cluster resources such that certain criteria are met. This is called a "resource allocation problem". For a time-bound resource allocation problem, which involves both the question of where (space dimension) and when (time dimension) to allocate a resource consumer, the term "scheduling problem" is commonly used. Both terms "resource allocation problem" and "scheduling problem" are used interchangeably in this thesis.

Scheduling is a very general concept and encompasses a large range of problems, use cases and approaches. This range spans from hardware scheduling, Operating System (OS) process scheduling over cluster job scheduling to capacity planning in manufacturing. No exhaustive classification[1] is given, instead it is focused on resource allocation problems arising in the thesis' context: Multiple, potentially distributed applications request access to resources in a computing cluster. Since an application is made up of at least one OS process, two levels of scheduling exist: Global scheduling deals with the problem of where and when to allocate (run) a process in the cluster. Local Scheduling is a task of the OS process scheduler on the node to which the process was allocated to.[2] The OS process scheduler usually is part of the OS kernel and follows the time-sharing paradigm: Local processes are given time-slices of the CPU, the length of the time-slice is repeatedly recalculated according to a predefined scheduling policy (e.g. fair-share according to priorities in Linux kernel 2.6 [71]). This thesis is concerned with scheduling problems on the global level. Nevertheless, the thesis goal may require the modification of the amount of resources given to an application's processes locally on a node. This, however, will not be achieved by patching or replacing the policies employed by a native OS process scheduler. Such an approach would limit the generality of a to-be-developed framework and put into question the usability of the framework in dedicated clusters. For instance, in the HLT-Chain use case such a modification is not permitted. A dicussion of OS process scheduling techniques is therefore omitted in this chapter.
When scheduling applications, a common strategy is to allocate the application's processes to cluster nodes once and leave them running until they finish their computing tasks. In certain cases, this approach is not sufficient: For example, if a low-prioritized application fully utilizes the CPU resources of a node and a high-prioritized application requests to be run, then it might be appropriate to stop the low-prioritized application in favour of starting the high-prioritized application. In this thesis, the phrase "dynamic resource allocation" will be used when scheduling of (distributed) applications not only involves an initial decision on when and where to run an application's processes, but also involves the usage of additional methods ("primitives") to modify the access to resources for already running applications. As motivated earlier in the introduction, a framework that strives to meet the thesis goal needs to possess the capability of "dynamic resource allocation": If the resource needs of a time-critical application change due to a varying data input rate, the resource usage of already running

[1] For a comprehensive overview on scheduling, please refer to Pinedo [87].

[2] The distinction between global and local scheduling was introduced by Casavant and Kuhl [20]. In this paper, a brief but well founded taxonomy for scheduling in distributed systems can be found.

additional applications might need modification such that no or only limited competition for resources occurs. This might be necessary for avoiding the violation of the timing constraint of the time-critical application.
In research, multiple approaches to dynamic resource allocation have been proposed and different aspects have been prioritized. Existing research work differs in pursued goals, methods of data acquisition, applied decision logic and proposed modification primitives. In the first section of this chapter, these goals and methods are briefly discussed to allow the embedding of this thesis in the current state-of-the-art research. After having introduced the context in which this thesis is situated, several significant approaches are discussed in the second section of the chapter.

3.1. Research Overview

This section gives an overview on how researchers propose to automate dynamic resource allocation in a cluster environment. The overview especially focuses on the usage of platform virtualization to host applications.[3] This technology is considered an enabling technology towards the goal of this thesis. The reasons herefore are discussed later in Section 4.3.2. Therefore, approaches for both native and virtualized applications are considered.

Computing Task	Schedulable Constituent	Finished Computation
Application	OS process	Result
Job	OS process	Finished job
Application	VM	Result

Table 3.1.: Differing terms used for scheduling of generic applications, jobs and VMs.

For the following overview, the terminology needs to be split to avoid confusion. The term job is a commonly used abstraction for a schedulable computing task in distributed environments. A job in its most simple representation is a specific application that consists of at least one OS process that requires to be scheduled to a physical node in a cluster. A job has an (often unknown) duration during which it consumes resources.[4] Once the job finishes, a useful result has been computed. As an example, a job can be the simple calculation of all prime numbers between 1000 and 2000, but also the weather forecast for the next weekend using a sophisticated atmospheric model. A job therefore is the computation of a single "result" by a specific application. With the introduction of the job term, there are now competing terminologies for the same concept. This is illustrated in Table 3.1. Both the first and the second row represent an application that consists of schedulable OS processes which upon finished computation yield a result. In contrast to the concept represented by rows one and two in this table, the third row represents virtualized applications. These also consist of OS processes, yet these processes are wrapped inside of VMs (Virtual Machines) with each having a dedicated OS environment. Distributed virtualized applications therefore consist of a set of VMs that need to be scheduled to physical nodes.[5] In this chapter, the term job will be used only for describing existing products that label their own approach as job scheduling, otherwise the term application is chosen. The term process will be used to denote the schedulable constituents of a non-virtualized application/job, and

[3]Details on virtualization technologies and products are given in Section A.4.
[4]For a comprehensive overview on the job nomenclature, its derivates like parallel jobs, job-shops and flow-shops and the according mathematical models, please refer to Pinedo [87]. For a more practical approach to "parallel machine models with parallel jobs" (i.e. scheduling of distributed applications in clusters), please refer to [40, 41].
[5]More precisely, a VM on a physical node is served by a local hypervisor that realizes the virtualization semantics.

the term VM is used to denote the schedulable constituents of a virtualized application.

The task of dynamic resource allocation involves decisions on *a*) which physical nodes are eligible to run constituents of a distributed application and *b*) to which extent local resources can be utilized. In this context multiple goals and methods have been examined in research. These are now described along the following dimensions:

1. What goals do researchers and products pursue?
2. What are the basic primitives proposed for changing the resource allocation?
3. What kind of information is incorporated to make a resource allocation decision?
4. How do researchers decide when and how to modify resource allocation?

3.1.1. Goals in Current Research

Three basic directions have been identified for the current research concerning dynamic resource allocation in clusters: First, researchers and industry focus on formalizing and satisfying runtime constraints for single applications. Second, it is researched how clusters can be exploited most efficiently concerning number of serviced customers, energy consumption and costs. The third direction is that of how to provide cluster resources to a multitude of users and applications in a uniform and abstracted manner.

Application Runtime Constraints Applications typically possess specific run conditions under which they are considered fully functional, or vice versa: there are conditions which may keep an application from fulfilling its purpose. Examples are real-time video conferencing or multi-tier web servers. Users only tolerate a certain delay, a non-tolerated delay makes the application unusable from the user's point of view. The term Quality of Service (QoS) [57], for instance, refers to constraints for traffic in telecommunication networks. Such constraints have always been an issue but have received additional attention in research over the last years: The shared and concurrent usage of computing clusters by multiple applications and users led to the need *a*) to formalize runtime constraints for applications and *b*) to find ways to make sure these constraints are met at runtime.
The first point has gained massive focus by industry in order to provide accountability. In analogy to the QoS terminology for telecommunications, the Service-Level Agreement (SLA) concept has been proposed in Information Technology Infrastructure Library (ITIL) [58]. A SLA is an agreement between a service provider and a customer and describes a service, its targets and defines responsibilities. An SLA can be used to describe the specific service "to host an application in a cluster". Part of the SLA is the Service-Level Targets (SLT) that describe the desired qualities of a service quantitatively. For this purpose the SLT consist of multiple Key Performance Indicators (KPI). An example KPI is "maximum response time of a web server per user request". The SLT therefore state under which circumstances an application is considered to be fully functional. While such targets enable legal treatment of software run conditions, they do little to describe how these goals can be met by a service provider. In ITIL, the term Operational-Level Agreement (OLA) is proposed for guidelines and agreements used within the Information Technology (IT) department of the service provider. Such OLAs comprise of low-level technical goals that when pursued and achieved will lead to meeting the SLT. For instance, if for a web server the maximum response time per user request is specified to be $5s$, a potential low-level goal to be pursued by cluster operators is to keep the CPU utilization under 80% on the physical nodes hosting the web server. SLT are not necessarily specified

using technical, IT related terms but can also be composed using some abstract KPI like customer satisfaction. As of today, there is no agreed-upon general technique to find and specify OLAs for given SLT. Some reseachers try to model the dependency between high-level goals (SLT) like response time, customer satisfaction or mean transaction rate and low-level goals (OLAs) like CPU usage mathematically [111, 133], using feedback control[6] [134], or based on historical experience and feasibility [52]. Others [46, 126, 134, 26, 8, 54] only monitor low-level properties like CPU usage and memory consumption of a physical machine. If these properties of a physical machine do meet certain criteria (e.g. average CPU usage over last 5 minutes < 80%), hosted applications are considered to perform correctly. Such an approach is criticized by researchers [14, 115, 84, 132] for being too general. These propose to consider other application-specific metrics, like errors in logfiles, when trying to meet SLT requirements.

Consolidation and Efficiency While the research work mentioned in the last paragraph deals with satisfying constraints and requirements of a single application, there is also the perspective of the service provider, e.g. the IT department running and providing the cluster environment for a multitude of applications. While any service provider wants to make sure that each customer is satisfied with the delivered service, he also aims at minimizing its own total costs or increasing revenue. This can be accomplished by e.g. reducing the number of running needed physical machines, reduced power consumption or maximized resource exploitation. Again ITIL delivers a term to describe this aspect: Such goals are subsummed under the term Business-Level Objective (BLO). Researchers often cover both application SLT and provider BLOs and try to find a trade-off between these potentially contradicting objectives. Some researchers specifically focus on server consolidation, trying to minimize the number of physical machines needed to serve applications [46, 14, 116, 102, 65, 135]. This typically is tried by putting VMs or applications as densily as possible on physical machines. Others explicitely claim to gain lower power consumption [118, 55, 89]. In contrast to cutting cost by using less servers or consuming less power, classical job schedulers like Load Sharing Facility (LSF) [88], Oracle Grid Engine (OGE) [81], Torque [108] or Maui [61] rather aim at improving the efficiency of a cluster by improving aggregated performance metrics: For instance the "throughput" of jobs describes the number of finished jobs over time, "walltime" (also: "turnaround") is the average time a job takes until its computation is finished and "queuetime" is the average time a job has to wait until being given computing resources. Another goal often pursued is to use existing CPU resources as optimally as possible, i.e. to strive for a high cluster CPU utilization. Such problems like reducing power consumption, computing more results over time, increasing exploitation of available resources but also offering satisfactory service to many applications at the same time are accepted goals pursued by many researchers.

Encapsulation and Interfacing Applications running in distributed environments often have specific runtime constraints, e.g. SLT. Sometimes applications are distributed, have a multi-tier architecture or base on parallel code (e.g. Message Passing Interface (MPI) applications). Between the constituents of a distributed application (processes/VMs) there may be interdependencies. The number of constituents may be variable for a single application.[7] Such properties and other relevant

[6]Control theory describes how a dynamical system, i.e. a system whose state evolves over time, reacts to input values. A closed-loop feedback controller uses sensor data, i.e. a system's output values, to continually adjust the input values for the system. A common goal of using a feedback controller is to make a system reach a specific target state. For more information, please refer to Franklin et al.[42].

[7]In job scheduling the options of moldability and malleability exist. While the former means that the number of processes can freely be chosen upon starting of a job, the latter even allows for manipulating the number of processes for a job at

meta-information, e.g. the amount of memory an eligible physical node has to possess, the number of CPU cores needed by processes/VMs or which OS environment is required, need to be specified and taken into account when making decisons on when and where to run applications. In job scheduling such meta-information is manually coded in job description files that are made available to a decision instance, the job scheduler. For a user that wants to run a job, an interface for submitting a job along with the description file exists. Upon submittance, the job scheduler takes full responsibility to schedule the job according to the requirements stated in the description file. In general, after submitting the job no further possibilities exist for the user to influence the scheduling of the job.[8] In grid computing this approach is extended to multi-cluster environments with intermediate decision instances (resource brokers) that decide in which local cluster the job is executed. Job scheduling with all its existing products (LSF, OGE, Torque, Maui, Condor) is a mature, highly configurable and fully established technology for running distributed applications in clusters. This still is different for the scheduling of virtualized applications, but with the dawn of cloud computing[9] now draws a lot of attention. Scheduling of virtualized applications is still an evolving technology and the question of how to interface and instrument virtualized environments and applications is still under debate. While for plain job scheduling only a simple description file is needed and appropriate physical resources have to be found before a job is run, running a virtualized application requires the existence of properly configured VMs and environments. This means that not only meta-information on the application is needed, but also a description of the VMs, a configured virtual environment and properly set up VMs are required. Research and product development has heavily focused on how to make distributed resources available to arbitrary applications in a virtualized manner. This spans:

- Tools like rPath [93] and Cloudzoom [28] for obtaining, creating and configuring specific VM images ("appliance").
- Unified, abstracted interfaces and APIs like libvirt [70] for managing virtual environments independently from the underlying virtualization technology and product.
- Cloud platforms providing standardized interfaces to create, operate and manage arbitrary virtualized environments (Amazon EC2 [4], Amazon S3 [5], OpenStack [86] and Oracle Management API [80]). These solutions offer APIs or a Graphical User Interface (GUI) for setting up and managing a complete virtual infrastructure including VMs, storage and Virtual Local Area Network (VLAN). However, management of the environment is still delegated to the user of these APIs. Basic automated resource allocation like initial allocation of VMs to physical nodes is provided using a round-robin, first-fit algorithm[10] for Amazon EC2.
- OpenNebula [101], Enomalism [37] and Eucalyptus [38] are software frameworks that instrument the above mentioned or proprietary APIs and establish a convenience layer on top of these APIs for easening the interaction with cloud environments. For instance they incorporate applicance building and VM life-cycle management, add enhanced user security (Authentication, Authorization), monitoring or high availability options. OpenNebula allows to define sets of VMs that form functional units, thereby starting either all or none of the respective VMs. Auto-

runtime.

[8]Job schedulers like Condor[72] also allow for pausing a job. All job schedulers allow to cancel a submitted job.

[9]An overview on cloud computing and the Everything as a Service (EaaS) stack can be found in Section A.3.

[10]Round-robin is a basic scheduling algorithm that assigns a resource to multiple resource consumers in a circular order, thereby implicitely giving each consumer the same priority. Each resource consumer receives the same portion of the resource. First-fit is a scheduling algorithm that assigns the first appropriate resource (found among multiple resources) to a resource consumer. A round-robin, first-fit approach in the respective context means that VMs are processed in a circular order and the first eligible node in the cluster is chosen as a host for a VM.

mated resource allocation for VMs does not go beyond choosing an inital placement based on round-robin, first-fit algorithms. However, the need to make resource allocation for VMs more adaptive and flexible is acknowledged, offering to plug-in external VM scheduling algorithms.

The need to treat a set of VMs hosting a single distributed application as a functional unit and to automatize resource allocation has been addressed first by Chase et al. [22] and Irwin et al. [49]. While Chase et al. only aim at modifying a set of VMs as a whole, Irwin et al. wrap a set of VMs with additional meta-information, calling it a lease. Such a lease belongs to a user, has a life-cycle and associated timing constraints. Researchers build on this early work and acknowledge the need to treat sets of VMs as functional units, representing a single virtualized application. A lot of research effort has since gone into the evaluation of methods how to dynamically manipulate resource allocation in distributed environments, especially for virtualized applications. Several approaches are discussed in the next section.

3.1.2. Resource Allocation Primitives

The basic operations ("primitives") for resource allocation in job scheduling are starting and stopping of jobs on physical nodes. While preempting processes, i.e. pausing single processes for a limited time frame, is an important operation used by process schedulers in operating systems, this is less common in distributed computing where gang scheduling prevails. Some job schedulers like Condor [72] offer the option to preempt jobs, but strictly limit its usage to jobs with single, non-communicating, Input/Output (I/O) restricted processes [31]. VMs offer more flexible methods to manipulate an application's resource allocation. This actually means that beside of initially allocating and running VMs on physical nodes until their computing tasks are finished, additional modifications at runtime are possible. Researchers currently focus on using such modifications in order to meet goals like server consolidation, increased resource usage or SLT compliance. The following primitives have been proposed:

Run-to-Completion The basic paradigm to run an application is run-to-completion. In its initial context this means a job/application is allocated to resources, put in running state and kept running until its computing task completes. This approach is adopted by OpenNebula for VMs and Maui, OGE and LSF for native applications. Such a basic strategy lacks flexibility for adapting to changing resource availability.

Stopping and Restarting Stopping an application and restarting it at a later point in time is a very basic way to modify the resource usage in a cluster. For their In-Vigo system Xu et al. [130] propose stopping of VMs and restarting these on different physical machines. They regard this as a mechanism to cope with QoS violations, specifically deadline violations. Unless partial results are saved prior to stopping a VM or job, this method has the drawback that unsaved computational state is lost and computations need to be repeated upon restart.

Preemption This downside can be avoided if preemption is used instead of stopping/restarting jobs or applications. Preemption basically means that a computing task is temporarily stopped from consuming resources. Its computational state is preserved until the task is resumed at a later stage. The base technology for preempting distributed applications is called checkpointing. Checkpointing saves the state of an application to disk such that it can be continued at this very checkpoint at a later point in time. However, checkpointing does not necessarily stop an application from running and consuming resources. Commonly this technology is used to save specific computational milestones that later can be rolled back to, e.g. in case of failures. Still, this base technology can also be used to manipulate the resource consumption of a running application, via checkpointing and successively stopping this application. There are multiple techniques to implement checkpointing for native, i.e. non-virtualized applications: user-, kernel- and application-level checkpointing.[11] Each of these have specific drawbacks, which eventually prevent the wide adoption of this technology in job scheduling. The most important ones are synchronization issues for distributed applications and the difficulty to guarantee that a later continuation occurs in a comparable, i.e. non-differing OS and kernel environment [9, 19]. Therefore, preemption for applications is applied by job schedulers only with strict limitations [31, 88] or if applications make use of specific checkpointing libraries [72], hence severely limiting the scope of runnable applications. For VMs, preemption ("suspend/resume") is a natively supported operation[12] which causes few problems because not only the application state is saved, but also its whole operating system environment including the state of I/O transactions and inter-process communication. Nevertheless, a problem that also exists for VM preemption is the synchronization between multiple to-be-suspended VMs, e.g. hosting an MPI application. For this problem, Walters et al. [121] propose a VM-aware MPI library which allows the parallel suspending of multiple VMs without breaking the application's functionality. Agarwal et al. [2] introduce a distributed checkpointing library and Emeneker et al. [36] propose to synchronize clocks before preempting a set of VMs. Despite of its native support by all virtualization products, VM preemption is not often proposed in research concerning dynamic allocation of resources for VMs. A possible explanation is that the overhead for the suspend/resume operations is still not sufficiently accounted for and coped with in current research. Also, suspending an interactive job or application will break its execution semantics, thereby only allowing best effort jobs to be suspended. Zhao et al. [132] mention the usage of the suspend/resume feature but do not specifically propose when and how to make use of it. Sotomayor et al. [100] discuss the overhead involved with suspending and resuming of VMs. Walters et al. [120] propose to use preemption of lower prioritized VMs in favour of other VMs hosting interactive applications.

Migration Migration is an operation that changes the allocation of a resource consumer to a certain resource. For native applications this means moving a process from one physical node to another. For virtualized applications this means moving a VM accordingly. The purpose of using this operation depends on the goal pursued by the researcher. For instance, it can be used to free resources on a specific physical node, but also for improving application performance by moving a process or VM to a physical node with more computing power. A distinction between cold, warm and hot migration can be made. Cold migration basically refers to the fact that a process or VM that previously had been running on a certain physical node was stopped and restarted on a different physical node. Accordingly warm migration means that a suspended single process or VM is resumed on a different physical node. The most interesting feature is hot migration. Using this operation, a process or VM can me moved from one physical node to another at runtime of the process or VM. In most cases, the migration act is transparent towards users or other processes communicating with the migrated process or VM [75, 27, 119]. Hot migration is a powerful tool for achieving different goals like server consolidation or application SLT compliance. Cold migration of jobs is a basic feature that job

[11] User-level checkpointing requires an application to be compiled/linked with checkpointing library support. Application-level checkpointing requires an application to self-provide the mechanisms for saving its state. Kernel-level checkpointing makes use of the preemption techniques of the OS process scheduler, but requires additional kernel patches or modules. For more information, please refer to [19].

[12] The naming of this feature differs for the specific VM products. See Table A.2 for nomenclature details. In this thesis checkpointing to disk, stopping and later continuation of a VM is referred to as suspend/resume.

schedulers make use of. Warm migration requires previous checkpointing and therefore is only used by job schedulers that utilize the checkpointing feature (Condor, LSF). Hot migration is not used in job scheduling. Cold and warm migration is supported by all virtualization platforms (see Section A.4 for an overview on virtualization platforms). Hot migration is also offered by most vendors, but sometimes (VMware ESX [123]) fully supported only in commercial distributions or with respective licenses. The migration feature is subject to current extensive studies. Xu et al. [130] propose cold migration. They suggest to stop VMs which violate a certain deadline three times in a row and to restart it somewhere else. They do not make remarks on which physical machines and when these VMs should restarted. Zhao et al. [132] also try to meet application deadlines by using warm migration. Sotomayor et al. [100] mention the benefit of warm migration for meeting time constraints of virtualized applications. Hot migration of VMs is focused on by a lot of researchers, e.g. for resolving hotspots [114, 126, 8], load balancing [94], consolidation [54, 55, 118, 102, 65, 47, 89] and SLT compliance [26, 98]. Others examine general properties and benefits of VM hot migration [111, 23]. The overhead of hot migration for VMs machines is explicitely tackled by Akoush et al. [3] and Voorsluys et al. [119]. From now on, the single term "migration" will be used to refer to VM hot migration.

Global Resizing A method for scaling an application at runtime, e.g. in order to meet SLT, is to modify the number of its processing components. In this thesis this approach is called "global resizing". PCM [73] and Dynaco [17] are initiatives that offer an API to MPI applications by which these can dynamically spawn new processes on new physical nodes if resources are available. Accordingly, processes can also be eliminated at runtime without affecting the functionality of the application.[13] APM [56] offers such a dynamical adaption in Symmetric Multiprocessing (SMP) environments.[14] Applications have to support this feature explicitely by either instrumenting a specific API or self-providing the methods needed for such a functionality like dynamic discovery, registration and integration of new processes.
A similiar approach called clustering is used by application servers like JBoss [60] and Websphere [122]. Applications hosted in such runtime environments are able to utilize additional application server instances such that application performance can be increased. Again, such applications need to make use of APIs and programming constructs (e.g. Enterprise Java Beans) provided by application servers in order to benefit from this feature. Common job schedulers are also able to incorporate new physical nodes at runtime. However, these are then only available to newly started jobs. Already running ones cannot profit from this modification. In the context of virtualization, adding (and removing) VMs to (from) a set of VMs hosting a specific application is no technical problem. However, in this case the same requirement exists: The hosted application has to support this up/down-scaling. Chase et al. [22] propose to modify the number of VMs dedicated to a specific application. In their work, VMs host a batch scheduling client (LSF) and the number of clients, i.e. the number of VMs, is dynamically adapted based on a metric provided by the job scheduling server (LSF).

Local Resizing Contrary to global resizing, which means modifying the number of processing units for a distributed application, local resizing modifies the share of resources given to a single process or VM on a node. A typical example is the "nice" command in Linux used for manipulating

[13]This means the application will not crash. It may, however, have an impact to the performance, i.e. the SLT compliance of the application.
[14]SMP refers to a paradigm in computer architecture where multiple processing units equally access a common shared memory. A typical example is a CPU with multiple cores which share a common memory space.

the scheduling priority of a process at runtime. Such a change of scheduling priority may affect the share of CPU cycles given to that process. A comparable command is "chrt" which allows a runtime change of scheduling policy and real-time priority the process scheduler applies to a specific OS process.[15] The command called "ionice" can be used to adapt the I/O scheduling priority of a process at runtime. Other Linux tools exist (iptables, niceload) that can be used to manipulate the amount of locally consumed resources of a single process. Common job schedulers (Maui, Condor, LSF, OGE, Torque) do not use such methods for modifying the resource share of running jobs. The virtualization platform XEN [11] offers runtime adaption of memory size, number of virtual CPU cores and maximum share of CPU cycles given to single VMs. VMware ESX offers runtime adaption of VM memory size and number of virtual CPU cores. A runtime modification of the number of CPU cores attributed to a VM is proposed in several works [89, 133, 94, 64]. Padala et al. [84] use the "cap" command to manipulate the CPU share utilized by XEN VMs. They also change the disk I/O bandwidth available to a running VM using a modified XEN I/O-driver.

3.1.3. Data Used for Decision Making

Different primitives have been presented that researchers propose for dynamic resource allocation However, it has not been mentioned yet how researchers make use of these primitives in order to achieve their goals. This question is split in two parts. First it is examined what data is used by researchers to make decisions on dynamic resource allocation. Second there is the question how this information is incorporated and which algorithms and techniques are proposed for making resource allocation decisions. The first aspect, the type of data used for decision making will be discussed next.

Static Data Available resources in the cluster and requirements of jobs/VMs need to be considered when making resource allocation decisions. There are several "static" properties of a distributed environment which commonly do not change in the short run, like the number of physical machines available, their hard- and software specification (Random Access Memory (RAM), CPU, Instruction Set Architecture (ISA), OS) and network topology. For a job, the description file offers the possibility to specify e.g. the memory and amount of CPU cores needed by a job's processes, the swap space a physical node has to possess and which OS environment is required to run the job's processes. Accordingly, a VM also has specific configuration properties like its number of virtual CPU cores, amount of RAM and desired network configuration. A resource allocation decision has to match such requirements with available cluster resources. For properties like number of CPU cores and amount of memory a cumulative match-making is sensible, taking into account the amount of resources already attributed to other jobs or VMs. The XEN virtualization platform is an exception because it explicitely allows over-provisioning of CPU cores and RAM ("memory ballooning"). Most scheduling approaches take into account such static information. Regarding network topology common job schedulers assume a homogeneous network such that full bisectional bandwidth is given for every node-to-node communication. However, this assumption does not always hold in real life environments and may lead to decreased performance for distributed jobs. A network topology-aware scheduling algorithm is therefore proposed by Pascual et al. [85] using locality-aware policies. Meng et al. [74] consider network topology in the virtualization context.

[15]The "nice" setting affects the length of a timeslice given to a process in the SCHED_OTHER fair-scheduling policy. The "chrt" command allows to change the scheduling policy for a process, e.g. from the common SCHED_OTHER fair-share scheduling to SCHED_FIFO real-time scheduling. "chrt" also allows to manipulate the static real-time priority of a process at runtime. For more information, please refer to the Linux documentation [71].

Dynamic Physical Node Data Static properties like number of CPU cores offered by a physical node and required by jobs or VMs are not subject to unpredictable fluctuations. In stark contrast to such properties there are resource usage metrics, like CPU usage on a node, that may exhibit volatile behavior, thus having an impact on achievement of the resource allocation goals. Such "dynamic" data needs to be retrieved and evaluated at runtime. The current CPU usage or load[16] of a physical machine is a criterion widely adopted by researchers to decide on where to allocate and run processes or VMs. Job schedulers like LSF, Condor and Maui allow to define thresholds on current CPU usage and load to specify which physical nodes are eligible for running additional jobs. LSF and Condor also use such thresholds for decisions on the preemption or stopping of jobs. In the context of VM scheduling Wood et al. [126] use CPU load to detect physical nodes with CPU bottlenecks. Tesauro et al. [109] and Choi et al. [26] use current CPU usage. Gmach et al. [47], Zhu et al. [135] and Rolia et al. [92] use historical trace-data about a physical nodes's CPU usage and extrapolate it to predict future CPU "overload". Other physical node metrics are also considered. Condor and LSF offer to incorporate the current paging rate, currently available free RAM, available swap space and even current disk I/O rate. For VM scheduling, the currently free memory on physical machines is considered by XU et al. [130]. Meng et al. [74] and Sonnek et al. [98]. The latter also take into account the current network traffic generated for a network interface of a physical machine. The disk I/O for a specific hard disk is used by Padala et al. [84]. Stoess et al. [106] use a physical node's thermal sensor data to indicate thermal overload which needs to be resolved.

Dynamic Process/VM Data Dynamic data for a physical node is queried using OS interfaces and may give hints whether a physical machine is overloaded or is suited for hosting additional processes/VMs. More specific information on already running processes or VMs seems helpful to some researchers. Job schedulers do not take into account runtime metrics for single processes, e.g the CPU usage caused by a process. While job schedulers allow the specification of constraints for jobs that determine when a job can be run or has to be stopped, such a specification always bases on metrics gathered for the whole physical node. In the context of VM scheduling, researchers propose to utilize VM-specific runtime information. Wood et al. [126] introduce the distinction between Black-Box and Grey-Box resource management. According to them, Black-Box resource management only uses VM-specific information that can be gathered via the physical node OS or the virtualization hypervisor.[17] On the opposite, Grey-Box management relies on data to be captured from the virtualized OS or application. While the latter allows to incorporate application-specific metrics and can therefore be used to tightly control SLT compliance, the former has the advantage of being application-agnostic and therefore suitable in general scenarios.

Black-Box Data Wood et al. [126] capture CPU load, swap activity and network traffic for every single VM. The CPU usage of single VMs is taken into account by several researchers [133, 89, 47, 111, 14, 8, 55, 65, 102, 54]. Meng et al. [74] and Sonnek et al. [98] measure network statistics for every VM, whereas Padala et al. [84] take into account not only overall disk I/O but also disk I/O specifically generated by single VMs. Wood et al. [127] make an interesting proposal for improving resource allocation decisions. These researchers claim that many VMs have similar contents in the physical node's RAM. If supported by the hypervisor, VMs with identical pages can be placed together on a physical node and those pages can be used together. Another paper by Verma et al. [118] discusses caching issues for VMs. They argue

[16]While CPU usage commonly refers to the shares of CPU cycles used by processes over a time interval, CPU load in most cases is the moving average for the CPU queue length over a time interval.

[17]Please see Section A.4 for details on virtualization technology.

that VMs dealing with data structures that fit into the L2 cache of a physical machine's CPU are more efficient compared to those dealing with data structures not fitting into this cache. They propose to place VMs together on a physical node if the summed size of their data structures does not exceed the cache's size. This latter proposal is not an explicit Black-Box approach because there are certain assumptions about the virtualized applications and the nature of the virtualization technology used. However, in comparison to the following Grey-Box approaches the researchers mentioned so far do not rely on monitoring the virtualized OS and applications.

Grey-Box Data Chase et al. [22] and Irwin et al. [64] propose to write application managers which process application log-files to decide upon the necessity of allocating additional resources. Meng et al. [74] propose to measure which VMs in a distributed environment communicate with each other in order to place them according to physical network topology. They show that network bottlenecks can be circumvented. A similiar approach taking into account current network traffic is chosen by Sonnek et al. [98] . They measure network traffic for VMs and place VMs with high communication traffic together on a physical machine.
As mentioned (see Section 3.1.1), the meeting of application SLT is one of the major goals in current research. A natural approach to achieving this goal is using application feedback. While adaptive control using application feedback is quite established in engineering, low-level network protocols and for specific applications like video conferencing systems and load-balanced web servers, this is less common for general purpose resource allocation frameworks like job and VM schedulers. This is due to one reason: Since every application has its own distinct SLT, a general way of incorporating arbitrary, non-standardized application-level data into a decision logic is tedious if not impossible. Hasselmeyer et al. [52] suggest that a human expert, based on historically retrieved measurements, manually translates desired web server throughput to low-level parameters like the number of web servers needed. Zhikui et al. [133], Turner et al. [111], Bobroff et al. [14] and Van et al. [115] propose to use response times of a virtualized application to determine appropriate VM allocations and properties. The additional usage of request rates is proposed by Wood et al. [126] and Van et al. [116]. Padala et al. [84] argue that response time and request rate are not sufficient and claim a benefit for measuring application throughput over time. A similiar argument provide Xiong et al. [129]. They incorporate variance and percentiles of response time as well as throughput rate.

Intrinsic Scheduler Data The types of data mentioned so far require access to external sources like physical node OS or application log-files. However, any scheduler also has internal metrics that emerge from past scheduling behavior. Such metrics encompass the computing time already given to a specific job, elapsed time since job submittance, time left to a potential deadline or averaged performance metrics like mean job throughput and walltime. Depending on the algorithmic approaches taken by a scheduler such metrics can be incorporated into decision making or even used for improving future scheduling.

3.1.4. Decision Techniques

It has been shown what data is incorporated into decision making and what primitives are used to modify resource allocation. The still missing link is the decision logic that evaluates the data and decides what primitives to apply at which point in time. A distinction can be made between scheduling of jobs or applications that do not run ("initial allocation") and modifying already running jobs or applications ("adaptive allocation").

Initial Allocation Mature job schedulers like Maui, Condor and LSF commonly prefer the run-to-completion paradigm. For this paradigm the questions *a*) which job to provision next and *b*) where to provision the processes of the job are important . All mentioned job schedulers can be configured with different scheduling policies. For the question which job to consider next, algorithms like first-come, first-served[18], fixed-priority scheduling[19] and fair-share with dynamic-priority scheduling[20] can be choosen as scheduling strategies. For finding appropriate target nodes, first-fit is the most commonly used technique. Advanced reservation, i.e. keeping specific nodes unused until all requirements of a high-priority job are met (instead of giving it to lower-prioritzed jobs), is another option offered by mainstream job schedulers. In the context of virtualization the same questions arise, i.e. which set of VMs hosting a specific application to provision next and where to put them in the cluster. For deciding which application to provision next, OpenNebula and Walters et al. [120] both use a first-come, first-served strategy. Walters et al. additionally incorporate the backfill strategy that allows to use unused resources for lower-prioritized or later submitted applications. First-fit decreasing[21] is also proposed by researchers[126, 118, 14, 63]. For finding an appropriate allocation of VMs to physical nodes the proposed techniques range from plain first-fit [120, 101] over the aforementioned first-fit decreasing approaches to more elaborated bin-packing search algorithms [135]. Gmach et al. [46] propose to use a genetic algorithm[22] to find a suitable allocation scheme. Van et al. [115] and Hermenier et al. [54] claim to have superior heuristics to find an optimized allocation using a "Constraint Satisfaction Problem" solver. Van et al. [116] suggest a search heuristic with application-specific utility functions for finding an appropriate allocation scheme for a set of VMs.

Adaptive Allocation The aforementioned approaches tackle the question when and where to run a job or set of VMs. A finer-grained resource allocation control, e.g. the preemption of running jobs, migration of a running VM or giving a running process more CPU shares, are not considered in these proposals. For run-to-completion approaches this question is irrelevant: Upon submittance of a job or virtualized application, a potential allocation is calculated and applied if feasible. Running applications or jobs are not manipulated. For approaches which claim to dynamically adapt resource allocation by manipulating running VMs or jobs, basically two questions arise: When to modify which running job or application (and its constituent processes and VMs) and how to modify it. Researchers focus on the first aspect, the approaches are therefore distinguished according to their method of deciding which running job or application to modify, i.e. how a decision to modify a specific job or application is triggered.

[18]First-come, first-served (FCFS) is a scheduling algorithm that assigns a resource to resource consumers in the order in which they have arrived, e.g. by submittance date.

[19]Fixed-prority scheduling is a scheduling algorithm that assigns a resource to resource consumers in order of their priorities. Priorities are given to a resource consumer only once and are fixed. For instance, a priority can be calculated depending on a resource consumer's temporal requirements, its importance for correct functionality of a system or credentials of a submitting user.

[20]Fair-share scheduling is scheduling approach that assigns each resource consumer an appropriate portion of a resource. One method to achieve this is dynamic-priority scheduling for which a resource consumer's priority is periodically updated in accordance with the time it has already consumed a resource.

[21]First-fit decreasing is a variant of the first-fit algorithm. In contrast to the latter, the eligible resource consumers are ordered and processed by decreasing size, i.e. by decreasing amount of required resources. That is, this algorithm also implies which resource consumer (application) to provision next.

[22]A genetic algorithm is a meta-heuristic that lets a set of sub-optimal algorithms evolve to an improved algorithm by applying techniques inspired by nature, such as selection, mutation and inheritance, on a set of possible algorithms.

Cooperative Decision Triggering For scheduling of VMs, Kiyanclar et al. [64] and Chase et al. [22] propose that every set of VMs hosting a specific application needs its own manager. This manager observes application statistics and decides when the set of VMs should be resized by starting an additional VM or a single VM should be attributed more CPU cores. These decisions are communicated to a central instance that applies the requested modifications. This is problematic: By delegating the responsibility for such decisions to third-party application managers, a central instance cannot find out about the priority and urgency of such decisions, relying on cooperative behavior of the managers. This problem is also known in cooperative scheduling for OS processes and has proven to be inefficient [103]. Later approaches therefore propose that decisions on dynamic resource allocation have to be made by a unified decision-making instance, a scheduler, that bases it decisions on a comparable metric like application priority.

Statically Triggered Decisions In order to decide about modifications to running jobs, LSF and Condor offer a language based on boolean and algebraic operators. Using this language, conditions on dynamic physical node data, e.g. CPU usage, can be specified together with a resulting modification like preemption or stopping of a job. For virtualized applications, multiple researchers propose rule or policy based techniques to trigger modifications. Approaches mentioned in this paragraph exclusively use warm and hot migration as the only manipulation primitive. Also, the focus of researchers is on when a migration has to occur, rather than where to migrate a VM. For this latter aspect, initial allocation algorithms (see above, Section 3.1.4) are used. The proposed policies or rules have in common that they contain a threshold, which triggers a decision upon its exceedance. Gmach et al. [46, 47], Rolia et al. [92] and Khanna et al. [63] propose fixed thresholds for CPU and memory usage, which trigger migrations when exceeded. Wood et al. [126] use a fixed threshold, but base it on a polynomial combination of multiple parameters like memory, network and CPU usage. They refine their decision by only triggering a migration if k out of n measurements already have exceeded and the next projected measurement will exceed the threshold. Bobroff et al. [14] and Kochut et al. [65] do not state when a decision has to be made and simply assume a periodic checking whether a better VM allocation is available. They stress the fact that a decision has to take into account the specifics of hosted applications, e.g. whether it has periodic fluctuations in CPU usage or application request rate. Both researchers model CPU usage of an application with periodograms and autocorrelation coefficients[23] to predict future overload on a server and to proact accordingly. Choi et al. [26] emphasize that fixed thresholds are not sufficient, but it is also important to know which VM (of multiple VMs running on a node) to migrate and where to migrate it. They propose to have a history about past migrations and record the benefit for decreasing the CPU load. Migrations are triggered if a threshold is exceeded and if migrations could resolve the overload in the past. Andreolini et al. [8] distinguish between the decision which server is overloaded and which VM needs to be migrated. For the first part, they use a cumulative sum model and for identifying a VM to be migrated they use a load-trend model based on linear regression.[24]

Dynamically Triggered Decisions Tesauro et al. [109] criticize the usage of fixed thresholds to trigger a modification. They propose to have an inital threshold which is incrementally im-

[23]Periodograms and autocorrelation are mathematical tools for estimating the spectral density of a time series or a stochastic process. Both reveal contained frequencies, i.e. periodicities and patterns. For more information, please refer to Box et al. [16].

[24]Linear regression is a statistical method used for fitting a linear function to measurement data. It therefore models a relationship inherent to the data.

proved using a neural network[25]. The aforementioned proposal of Choi et al. [26] to include a history about the success of past migrations can also be considered as a simple dynamic trigger model. Other researchers make decisions using mathematical models or Proportional-Integral-Derivative (PID) controllers[26]. Turner et al. [111] propose a general purpose mathematical model which combines resource allocation, CPU usage and response times of applications. They do not state how to use the model, i.e. when to make a decision, but emphasize its strength for matching high-level response time contraints (SLT) to low-level parameters like CPU usage. Zhikui et al. [133] propose a model based on queueing theory[27] which facilitates the assignment of CPU shares to a VM based on CPU load and response times of an application. Zhu et al. [134] propose nested feedback controllers for minimizing resource usage by also meeting response time constraints for a hosted web server. Their controllers dynamically adapt the CPU shares given to a specific VM. Padala et al. [84] claim that multiple parameters need to be adapted when aiming for server consolidation and SLT compliance. They modify CPU shares and disk bandwidth given to a single VM by using a feedback controller. Xiong et al. [129] also use a feedback controller to govern the CPU shares a VM is given on a physical node. They stress that their controller processes multiple input parameters like response, response time variance, application throughput and resource usage.

3.2. Evaluation of Relevant Approaches

As a reminder, it is the thesis goal to develop the concept and implementation of a software framework for exploiting unused resources in dedicated clusters running a time-critical application. Constraints are that by running additional applications *a*) the time-critical application should not be affected in its proper functionality, *b*) this has to be achieved without controlling the time-critical application and *c*) cluster resource (CPU) usage has to be increased and additional computational results have to be generated. The term Main Application (MainApp) is used to refer to the time-critical application, for example the HLT-Chain. An additional application that exploits unused resources via the provided framework is called a Secondary Application (SecApp). By comparing the thesis goal to the research goals presented in Section 3.1.1, the following can be stated: A goal adopted by researchers, the satisfaction of an application's runtime constraints, corresponds to the thesis sub-goal of preventing that a MainApp is affected in its proper functionality. Another goal adopted by researchers, to increase the efficiency of a cluster, corresponds to the thesis sub-goal of increasing cluster resource usage and increasing the result output generated by SecApps[28]. Three issues emerged while evaluating the state-of-the-art research with respect to meeting the thesis goal: First, approaches using only the run-to-completion approach were discarded because they do not offer the flexibility to manipu-

[25]A neural network is a computational model for processing data. It consists of a number of interconnected artificial neurons and maps input data to a desired output. Its bottom-up design and learning capabilities make this technique appropriate for processing unstructured data that is difficult to treat using deductive, symbol manipulation-based top-down approaches. Visual and audio pattern recognition are prominent examples for using neural networks. Please see Fausett [39] for more information.

[26]PID controllers are a special type of feedback controllers.

[27]Queueing theory is a mathematical model for waiting lines (queues). It uses probability distributions to model items that enter a queue (e.g. customer calls) and items that leave a queue (e.g. customers served). For instance, it allows to make statements on the average number of items in a queue or the average time an item spends in a queue. For more information, please refer to Gross et al.[51].

[28]Several approaches focus on server consolidation by densely packing applications on physical nodes. Though not precisely fitting the stated thesis goal, such approaches were also considered. Both the thesis goal and server consolidation require efficient scheduling to optimize the cluster usage.

late running applications such that Service-Level Targets (SLT) for a MainApp can always be met in case of changing resource needs[29]. Second, most of the research works do not provide ready-to-use implementations but rather present mere proposals, theoretical models, simulation results or empirical results gathered with prototypes in a cluster with few (≤ 5) physical nodes. Also, in most cases only a single aspect is discussed, either the efficient initial allocation of submitted jobs/applications or the manipulation of running jobs/applications. In the latter case only very few approaches propose to apply more than one primitive, e.g. using both migration and local resizing like Zhu et al. [135]. Another aspect is that many approaches explicitely deal with a specific time-critical application, like a web server, and incorporate application metrics to meet the SLT of this application. Using such Grey-Box data is an acceptable approach, but cannot be used for a general purpose scheduling framework. Third, except for one approach (Condor, see Section 3.2), it is always assumed that cluster resources are under full control of a decision instance and any running application can be manipulated by using one of the mentioned primitives. However, according to the stated thesis goal the MainApp is a black box and cannot be controlled by a scheduler[30]. Even when researchers did not acknowledge this specific constraint, their approach was not rejected per se. Instead it was evaluated whether the methods applied could be used such that the SLT of a MainApp can be met without actively manipulating this application.

In this section ready-to-use solutions that aim for satisfying application runtime constraints in a general way are presented. It is discussed whether they can be used to fulfil the goal of this thesis.

OpenNebula SVMSched Smart Virtual Machine Scheduler (SVMSched) [21] is a scheduling extension to OpenNebula [101]. It makes use of the interfaces offered by OpenNebula for creating, deploying and managing VMs. Its goal is to provide fair and efficient access to cluster resources for a number of virtualized applications. The product allows customers to submit requests to run an application hosted by a set of VMs. Then the needed VMs are instantiated and handed over to a scheduler that makes decisions when to provision which application. The developers distinguish between two classes of applications: Best effort and production jobs. Production jobs are time-critical and should be given sufficient resources to perform correctly. For scheduling of production jobs, priority scheduling with backfill is applied. Once provisioned, a production job is subject to the run-to-completion paradigm. With respect to the thesis goal a comparison can be drawn: A production job can be regarded as a MainApp that cannot be manipulated. Best effort jobs are scheduled using a first-come, first-served strategy with backfill and can be preempted. These jobs correspond to the SecApps of this thesis. However, there are two main aspects which render the conceptual approach taken by SVMSched unusable for the thesis goal. First, the preemption of best effort jobs only takes place when new production jobs are submitted, not when a currently running production job requires more resources. So a dynamic resource allocation according to the changing needs of a running production job is not possible. Such an approach is only feasible if a constant resource consumption for running production jobs is assumed. This assumption cannot be made for a MainApp in the thesis scenario. Second, only a single manipulation primitive, preemption, is used for best effort jobs. This lacks flexibility, especially when taking into account that there is considerable network overhead[31] associated with using this primitive. The network usage, which is a potential bottleneck that could affect a production job's performance is not considered by SVMSched when deciding on the preemption of best effort jobs.

[29]If a time-critical application has a varying data rate to process, such resource needs will vary as well.

[30]The primitives for modifying resource allocation cannot be applied to this application.

[31]Suspending a VM means saving its memory state to a shared storage facility. This causes network traffic.

The SVMSched extension for OpenNebula can therefore not be instrumented for meeting the thesis goal.

VMware vSphere VMware vSphere [114] is software framework for managing a distributed environment running the VMware ESX [123] virtualization platform. The product is mature and has a strong market penetration [113]. It provides an API, a GUI and a single-point-of-access to deploy, manage and survey virtualized environments and applications. Distinctive features are its ability to provide fault tolerance and high availability to customer applications and easy Infrastructure as a Service (IaaS) cloud[32] integration. The resource allocation of virtualized applications (sets of VMs) can be done manually. However, certain features that aim for dynamic resource allocation are provided. The modules of interest are the Distributed Resource Scheduler (DRS) and the vFabric Application Performance Manager. While the precise algorithms are not disclosed, the following can be stated: The second module specifically claims to make virtualized applications meet their SLT. However, to-date this vFabric Application Performance Manager only offers the specification and monitoring of Key Performance Indicator (KPI)s, sending out customer notifications for critical application states. Upon notification, manual interaction for tracing and resolving potential SLT violations (e.g. via adding a CPU core to a VM) is required. This module therefore is not sufficient for automated dynamic resource allocation.
The other module (DRS) aims at providing automated server consolidation and CPU load balancing for physical nodes. This is accomplished by migrating VMs. So the load balancing feature of DRS can be used to strive for application SLT compliance by avoiding CPU bottlenecks. It is possible to prioritize a specific application hosted by a set of VMs, thereby making it comparable to a MainApp of this thesis. However, vSphere cannot be configured such that migrations are only used for lower-prioritized applications. Dynamic resource allocation for all applications in the cluster is assumed, making this product inappropriate for achieving the thesis goal. In addition, the current network usage of physical nodes is not considered when deciding on migrations. Since migrations put a strain on the network, this could lead to a violation of the SLT of a high-priority application.

1000 Islands In their research work, Zhu et al. [135] provide a software framework that aims for both server consolidation and SLT compliance for virtualized applications. They propose to use both local resizing, i.e. dynamically attributing VMs CPU shares, and VM migrations. More explicitely they propose a three-layered architecture. On local nodes they propose to use feedback controllers to match high-level goals (e.g. response time of an application) to low-level actuators like CPU shares given to a VM. On a second layer they propose to identify overloaded servers based on historic CPU load trace data and to find potential VM migration target nodes via a simulated annealing based search algorithm. On a third layer they revise past allocation decisions for their benefit in order to improve future scheduling. With respect to the thesis goals the following can be stated: These researchers strive for SLT compliance of applications. They assume to have control over all applications, explicitely modifying their resource consumption and node allocation. The question is whether the framework can be used in a way, that a highly-prioritized application is monitored for SLT compliance and resource allocation decision are executed on lower-prioritized applications only. Their second and third layer, where VM placements and migrations are calculated, cannot be operated in this manner. The first layer in principle could be configured such that lower-prioritized applications are given less CPU shares, thereby automatically increasing the share available to a high-priority application. The approach used for the last layer could be used strive for SLT compliance without actively manipulating

[32]For an overview on clouds and the EaaS stack please refer to Section A.3.

a high-priority application. The framework as a whole, nevertheless, is not appropriate:
First, on local nodes (first layer) they only modify the CPU shares given to VMs. This is not sufficient because also bottlenecks for other resources like network can arise that harm an application's performance. Second, the methods applied on layer two and three do not support the exclusive migration of VMs of lower-prioritized applications, but assume to have control over all applications. Third, the framework does not describe whether additional applications can be submitted at runtime. The researchers assume that the number of applications running in a cluster is fixed, so no inital allocation decisions for not-yet running applications are supported. Because of these reasons the framework is not suited for meeting the thesis goal.

Condor Condor is a job scheduling framework. Being started in 1988 this project has since reached a mature status and its implementation is used in about 3000 clusters worldwide [32]. According to the project's own description [72] it focuses on high-throughput computing, i.e. an optimization in job throughput over weeks, months and years. Though initially targeted at desktop environments, they claim to be a usable in high-performance computing clusters as well. They explicitely address the problem of under-utilized computing nodes, aiming at exploiting unused CPU cycles by running additional jobs, a task they refer to as "CPU stealing". They specifically mention the aspect that Condor can be used in dedicated environments, i.e. where applications run that Condor has no control over, e.g. desktop environments. Therefore they formulate a goal comparable to the thesis goal: Exploitation of unused CPU cycles without affecting natively running applications. In order to specifiy when a physical node is eligible to run Condor jobs the ClassAd configuration mechanism is used. ClassAd offers a language to specify constraints on physical nodes, jobs and timing. These constraints steer the job scheduling to physical nodes. Thereby criteria can be defined when CPU resources on a node have to be freed in order to satisfy MainApp requirements. In Condor this is called satisfying the wishes of a resource owner - in contrast to satisfying the needs of resource users (Condor jobs). While this ClassAd language is powerful and detailed requirements can be specified, native support only takes into account CPU consumption (CPU load and usage) for the evaluation whether a MainApp is affected. Via extensions like Hawkeye other resources can be monitored and taken into account for job scheduling, like available network bandwidth or disk space.

So, Condor is a framework that via its extensions can be used to run additional jobs in a distributed environment with dedicated applications. However, there are several drawbacks that make its usage for the HLT-Chain application difficult: First, the primitives usable for dynamic resource allocation are starting/stopping and preemption. However, jobs have to comply with strict limitations [31] or need to implement a specific checkpointing library in order to be preemptable. Because starting/stopping is inflexible and leads to a low job throughput (i.e. SecApp result output), the scope of jobs that can be scheduled efficiently is severly limited. Other primitives for dynamic resource allocation by using migration or local resizing are not considered. Second, only jobs utilizing the same OS-platform configuration like the MainApp can be run. This additionally restricts the number of runnable jobs due to potential library conflicts or conflicting OS requirements. Third, there is no guarantee that buggy or vulnerable job code[33] won't compromise a running MainApp. The provided level of isolation between SecApps and MainApp is not sufficient.

Condor aims at similiar goals like this thesis does. It provides a mature, accepted and highly configurable framework to exploit unused CPU resources in a dedicated environment. It does however lack

[33] A job can have infinite loops, cause memory holes or can be vulnerable to hacking attacks.

isolation for MainApps, generality considering the type of runnable jobs and flexibility in modifying the resource consumption of running jobs.

3.3. Summary

Two main sub-goals of this thesis, meeting the performance targets of an application and improved cluster usage and result output are subject to existing and ongoing research. Basically two main scheduling paradigms are adopted for running distributed applications in a cluster: Run-to-completion and dynamic resource allocation. While the former only involves an initial decision on when and where to run an application, the latter also allows for manipulating running applications. For dynamic resource allocation, multiple manipulation primitives like preemption, migration and local resizing and different decision-making techniques like search algorithms and feedback controllers have been proposed by researchers.
No existing solution meets the goals stated for this thesis. Run-to-completion approaches are not sufficient because they lack the flexibility to manipulate running SecApps such that a MainApp's Service-Level Targets (SLT) can always be met. Among the dynamic resource allocation approaches most approaches do not provide a mature implementation or only discuss the use of a single primitive, thereby lacking in flexibility to adapt the resource usage of SecApps at runtime. Most proposals do not meet the requirement that a running MainApp cannot be actively controlled. Only a single product does so and only manipulates SecApps in order to meet a MainApp's SLT. However, this product, Condor, does not provide the level of isolation and flexibility needed in this thesis.

4. Conceptual Work

In this chapter the developed conceptual approach will be presented. The chapter is structured in fours sections that build on each other. The scope of the to-be-developed framework is briefly discussed in the first section. The second section follows with an introduction to basic terms like resource, resource usage and interference. Using this terminology, the basic conceptual cornerstones will be pinpointed in the third section: The usage of OS based metrics as the basis for making resource allocation decisions is motivated, leases and VMs as feasible SecApp containers are introduced, the role of policies as constraints for SecApp resource usage is discussed and finally a basic proposal for the functional design of the framework is given. The application logic of the framework and its constituents is discussed and algorithmic approaches are detailed in the fourth section.

4.1. Scope Discussion

As a reminder, the goal (see Chapter 2) is to develop the concept and implementation of a software framework for running additional applications ("SecApps") in a dedicated cluster with a time-critical application ("MainApp"). This goal is constrained by: *a*) The MainApp must not be affected in its proper functionality, *b*) this has to be achieved without controlling the MainApp and *c*) cluster usage has to be increased and additional computational results have to be generated. Before turning to the conceptual work, basic explanations are given to refine the scope of the thesis.

MainApp Scope For any MainApp, there is a precondition to being in the target scope of this thesis: A MainApp has to have a performance specification, i.e. a quantitative timing constraint. Without such an explicit constraint, the question whether SecApps affect a MainApp cannot be answered. This does not not exclude such MainApps from being used with the to-be-developed framework, yet achievement of the thesis goal cannot be evaluated for these. In the previous chapter, SLTs were introduced as a mean to describe such constraints for applications. Other, non-ITIL conform approaches are also valid, as long as they provide an evaluable functionality criterion. Proper functionality of a MainApp is assessed using this criterion.

SecApp Scope Since the amount of unused resources in a dedicated cluster is unknown, may vary or tend to zero over prolonged time phases, SLT-constrained applications do not qualify as SecApps. In this thesis the set of applications that can be run as SecApps is therefore restricted to non-interactive, non-deadlined applications that have no explicit SLT requirements. Still there is a great range of applications that conform to this requirement and can be scheduled using a best effort approach. This range encompasses most applications runnable in Volunteer Computing (Seti@Home [7], Boinc [6]) and Grid Computing (AliEn [10]) as well as typical job scheduling applications like scientific simulations, data analysis and code compilation.

Generality It is not reasonable to claim that a provided solution allows to run SecApps in an arbitrary dedicated cluster with a time-critical application such that the thesis goal can always be met:

Imagine a MainApp has to process a constant data-rate. The application is network-bound and most network links are nearly fully saturated throughout the runtime of the MainApp. The average cluster CPU usage is at 50%. The MainApp has a functionality criterion, say maximum response time per request of 100 ms. This criterion is met, yet the actual response time is permanently close to that upper limit, e.g. the mean response time is 90 ms with minimal variance.
In such a scenario, an additionally run SecApp most likely will cause the MainApp to violate its functionality criterion because an additional application will under most circumstances put a strain on at least one network link.[1] So, there are scenarios where both running SecApps and preserving proper MainApp functionality is a hardly achievable goal.

Therefore, it is not claimed that the framework will be capable of increasing the usage and result output of any dedicated cluster without compromising an arbitrary MainApp. Instead, a practical, inductive approach is taken: The framework has to satisfy the thesis goal for the HLT-Chain application, but should be applicable in other scenarios as well. From a functional perspective, this requires to use generic concepts and methods such that the framework can be deployed in alien environments. Whether using the framework in such an environment is indeed benefitial, depends on three main aspects: *a*) The amount, scale and temporal/spatial distribution of unused resources (when, where and how much resources are free), *b*) the sensitivity of the MainApp towards lack of computing resources (how does a short-term resource contention influence the performance of the MainApp and *c*) the tightness of the performance constraint (is there a considerable gap between worst observed performance and tolerated performance). Depending on these aspects, a trade-off between benefit of running SecApps[2] and negative impact towards a MainApp needs to be made. In this thesis, it is tried to find "configuration items" that allow to make such a trade-off. It will be evaluated to what extent these configuration items (policies) are suitable to making a trade-off in different scenarios.

4.2. Basic Terminology

A problem in achieving the thesis goal is that criteria and decision logic have to be provided that allow to control the access to cluster resources for SecApps such that the MainApp is not affected in its functionality. At the very core this is a resource allocation problem, i.e. how to allocate potential resource consumers (SecApps) to resource producers like cluster hard- and software. In this section the terms necessary to describe such an allocation are introduced. The phrase "a MainApp is affected in its proper functionality" will be defined by introducing the concept of interference.

Resources and Resource Usage A computing cluster comprises of interdepending hard- and software components and offers resources which can be used by resource consumers. Rather than giving a precise, from-scratch formalization of a cluster, the focus is on those components and their

[1]It is assumed that any non-trivial application in a cluster requires network communication. However, from a theoretical point of view this is a matter of scale. If a very fine-grained control over the network usage of this application can be exercised, then it is possible to make use of the few percent unused network bandwidth without producing a bottleneck. According to the explanation given in the first paragraph of Chapter 3, local scheduling in form of intervening with kernel-level process and I/O scheduling is not allowed as a method for achieving the thesis goal. Without this possibility, its is difficult to have the fine-grained control over network communication needed to exploit the few percent unused network bandwidth without incidentally stealing bandwidth from the MainApp. Even if a fine-grained control could be exercised, it is questionable whether this will yield a significant increase in cluster usage and SecApp result output.
[2]The benefit of running SecApps are increased cluster CPU usage and computed SecApp results.

properties that are regarded sensible for describing resource consumption in a cluster. Differing formalizations of distributed environments for specific purposes can be found in [24, 33]. A cluster comprises of at least three sets of architectural items:

$$cluster = (\{\, node_i \mid 1 \le i < n \,\}, \{\, switch_j \mid n \le j < m \,\}, \{\, storage_k \mid m \le k < p \,\})$$

Any $node_i$, $switch_j$ and $storage_k$ is a resource producer, also called cluster item. A cluster item provides resources, e.g.

$$node_i = (CPU_i, NetOut_i, DiskRead_i, Mem_i, ...)$$

These resource may correspond to physical hardware, like a CPU or an Ethernet interface, but not necessarily need to. Also a resource like $Users_i$, representing slots for user logins, is possible. For every resource, a metric is defined that denotes the usage of a resource:

Let $\Gamma = \{t_k := t_0 + k\Delta t, k \in \mathbb{N}\}$ be an equidistant set of points in time. It is assumed that $\Delta t = 1s$ unless stated otherwise. A function $f : \Gamma \mapsto \mathbb{R}$ maps these points in time to a set of measurements denoted by $\{x(t_k)\} = \{x_1, x_2, x_3, ..., x_k\}$ with x_k given in units of e.g. $[GBit/\Delta t]$. The absolute usage of a resource res at point in time $t_0 + k\Delta t = t_k$ is then defined by:

$$currentUsage_{res}(k, N) = \sum_{i=0}^{N-1} \frac{1}{N} x_{k-i}$$

According to a resource's specification or empirical knowledge, there is also a known maximum for its usage:

$$maxUsage_{res}(k, 1) = x_{max} = const \text{ , with } x_{max} > x_k \ \forall k \in \mathbb{N}$$

As an example, for an Ethernet interface the maximum throughput achievable for sending data is given by $maxUsage_{NetOut}(k, 1) = 1GBit/s$. Based on currentUsage and maxUsage, a normalized (relative) usage function can be defined:

$$Usage_{res}(k, N) = \frac{currentUsage_{res}(k,N) * 100}{maxUsage_{res}(k,1)}$$

By definition $0 \le Usage_{res}(k, N) \le 100$ holds. Thereby the free, i.e. non-used share of a resource is easily defined by $freeRes_{res}(k, N) = 100 - Usage_{res}(k, N)$. $Usage_{res}$ and $freeRes_{res}$ can be interpreted as a percentage which is why in this thesis the terminology "a resource usage of x%" is used.

Applications and Processes As mentioned, resources are used by resource consumers. An application is such a resource consumer. A running application is a set of processes $app = \{proc_i \mid 1 \le i \le n\,\}$. Any process $proc_i$ can use resources:

$$currentProcUsage_{res}(k, N, proc_i) = \sum_{i=0}^{N-1} \frac{1}{N} x_{k-i}$$

with x_{k-i} denoting the measurements for the usage of a resource res generated by process $proc_i$. The normalized resource usage generated by this process for the resource is defined accordingly:

$$ProcUsage_{res}(k, N, proc_i) = \frac{currentProcUsage_{res}(k,N,proc_i)*100}{maxUsage_{res}(k,1)}$$

Like for $Usage_{res}$, $ProcUsage_{res}$ can be interpreted as a percentage. In this thesis the terminology of "process y uses x% of a resource" is therefore used. The normalized usage of a resource can also be written as:

$$Usage_{res}(k, N) = \epsilon_1 + \sum_{i=1}^{P} ProcUsage_{res}(k, N, proc_i)$$

where P denotes the number of processes using this resource, i.e. with $ProcUsage_{res}(k, N, proc_i) > 0$ and ϵ_1 represents the residuals not captured by the modelled processes, e.g. the overhead of the OS. The definition is highly idealized and ignores both potential OS process inheritance and the specifc OS process scheduling techniques used in a modern OS. The term process used in this thesis refers to the entirety of an application's parent and (temporarily) created child operating system processes on a node.[3] A process needs to be allocated to a node in order to use resources provided by that node. The allocation relation $alloc(proc_j, node_i)$ is defined upon a single measurement interval Δt and forms the basic relation a resource consumer can have to a resource producer at a given point in time. A process can only be allocated to at most one node at a given point in time:

$$ProcUsage_{res_i}(k, 1, proc_j) > 0 \implies alloc(proc_j, node_i)^4$$
$$alloc(proc_j, node_i) \implies \nexists m : node_m \neq node_i \wedge alloc(proc_j, Node_m)$$

The most intuitive interpretation of an allocation is probably that of a non-distributed application that is started on a cluster node. Once the application is started, the allocation relation between the process (representing the application's local OS processes) and the node is established. Once the application is stopped, the allocation relation is dissolved.

Interference It is part of the thesis' goal to allow SecApps to exploit cluster resources without affecting the proper functionality of a MainApp. The concept interference is now introduced to capture this notion more precisely.
Let app_1 and app_2 be two applications. For app_1 there is a (quantitative) performance metric such as transactions per seconds, response time or latency. In scenario (A), app_1 is run and its performance metric is measured, yielding a value v_1. Scenario (B) extends scenario (A) such that, ceteris paribus, app_2 is also run and a performance metric value v_2 is measured for app_1.

Definition 4.1 *If there is a statistically significant difference between v_1 and v_2, then app_1 is said to be interferred by app_2, i.e. app_2 causes interference towards app_1.*

This definition allows a statement on whether a SecApp has impact towards (the behavior of) a MainApp. However, not every statistically significant difference for an observed performance metric is relevant. Relevance is tied to predefined thresholds (e.g. SLT) for such performance metrics. Let us assume there is an upper limit v_{max} for a performance metric that defines a reference value that can be used to assess whether an application is fully functional. Then v_{max} can also be used to assess whether a statistically significant difference for an observed performance metric is relevant.

[3] If the common interpretation of process as an operating system process is referred to, then the term "OS process" is explicitely used.

[4] This statement assumes that res_i is a resource provided by a $node_i$.

Definition 4.2 *If $v_1 < v_{max}$ for scenario (A), app_2 causes interence to app_1, and $v_2 > v_{max}$ for scenario (B), then app_1 is said to be strongly interferred by app_2, i.e. app_2 causes strong interference towards app_1.*

So Definition 4.1 describes statistically significant interference while Definition 4.2 describes relevant interference. Additionally, the term "amount of interference" is now introduced.

Definition 4.3 *A statistically derived absolute difference between v_1 and v_2 is called amount of interference.*

Such an empirical approach to assessing interference now has specific drawbacks:

- From a practical point of view, this definition requires a certain number of measurement samples to derive statistical statements about interference. This can be problematic in productive environments.
- This definition uses a white-box approach for assessing an application's performance. This means that application-specific metrics rather than OS-specific metrics are used. So depending on the application and its performance metric, e.g. logfiles have to be examined, APIs have to be queried or even customer feedback is needed to assess application performance. This makes the development of a general approach difficult.
- Most importantly, the definition does not make any statements about a general sensor metric that can be used at runtime of an application to detect and remedy potential causes of interference[5]. So these definitions serve the purpose of stating whether interference has occurred, but not how to control the probability of occurring interference at runtime.

Summing it up: A cluster consists of cluster items, e.g. nodes, each of which offering resources that can be used by resource consumers. Applications were introduced as prototypical resource consumers. An application consists of at least one process.[6] A process runs on at most one node. This correspondence is represented by the relation $alloc(proc_j, node_i)$. A process may use resources, the usage of a resource is described by a function $Usage_{res}(k, N)$. Applications of interest are the Main Application (MainApp) and the Secondary Application (SecApp). When both applications run concurrently, the performance of the MainApp may be compromised. This aspect is acknowledged by the introduction of the "interference" concept. The metric "amount of interference" enables a quantification of the performance impairment. The concept "strong interference" allows for an assessment whether SecApps affect a MainApp's performance such that it performs beyond its specification. Finally it was concluded that interference and application-specific performance metrics are no sufficient base to make runtime decisions on dynamic resource allocation.

4.3. Conceptual Cornerstones

As shown in the last section, the concept of interference helps to assess whether a MainApp's proper functionality is affected by additionally run SecApps. However, it was also stated that the mere observation of interference provides insufficient information for making resource allocation decisions at runtime. In this section it is explained what is regarded reasonable information to base such decisions

[5]If a performance metric is defined on a bigger time scale, e.g. customers served per day, the detection of strong interference can easily lag behind occurrence time of its potential cause.

[6]A process is the entirety of all OS (child) processes and threads of an application that run on the same node.

on. Also, an introduction to the basic design principles of the framework will be given. The general term "decision instance" is tentatively used in this section to represent a (not-yet-elaborated) decision logic of the framework. This section is now structured as follows:

1. As a first step, it is discussed as to how resource allocation decisions can be based on a sensor metric $Usage_{res}(k, N)$.
2. Next, the usage of platform virtualization as an enabling technology for the thesis goal is explained. The life-cycle of a VM will be defined.
3. Then the "lease" concept is described. A lease is a container to encapsule arbitrary SecApps and to standardize their access methods.
4. At this stage, "policies" as configuration items for the decision instance are introduced. The actual configuration of policies determines how the values provided by the sensor metric are interpreted by the decision instance and influence the decision making.
5. Finally the term decision instance is elaborated. A distinction between multiple functional units will be made and a tentative description of their purpose and interplay is given. This last step is the basis for the next section which exclusively deals with the functional, architectural and algorithmic design of the decision logic.

4.3.1. Resource Usage as Sensor Metric

As a reminder, the core challenge of this thesis is to prevent strong interference towards a MainApp when running additional SecApps. The argument was made that application-specific performance metrics can be used to assess interference, but are not sufficient as the only source of information when having to decide on resource allocation at runtime (see Section 4.2):
First, a certain number of measurements is needed to make sure that e.g. observed deviations from a target performance value are significant and require an immediate reaction. Second, application-specific performance metrics can vary considerably between applications (e.g. customer satisfaction vs. transactions per second) and make a general approach for decision making difficult. Third, there is no reason to assume that the observation of (strong) interference coincides with its cause. For instance there can be a substantial lag between detection of a problem and its cause: If for an application a target performance value of "serve 1M customers in August" is given, then a failure in achieving this goal can be due to a malfunctioning node on 08/01 but is only observed on 08/31. This clearly leads to the following conclusion:

- The interference Definitions 4.1, 4.2 and 4.3 can be used for the ex-post assessment of interference. In this thesis they are used to assess whether and to what extent interference occurred in experimental scenarios.
- A sensor metric is needed which can be used by a decision instance to make decisions on when and where to allocate which resource consumers such that the amount of interference is kept under control, i.e. that no strong interference occurs. This metric should possess several qualities. It should be:
  - quantitatively measurable,
  - updated in a (relatively) high frequency, i.e. a $1/s$ or $1/5s$ readout frequency should be supported and reflect a current state,

  - general, i.e. not dependent on used applications, hence based on common OS provided system data,
  - actable, i.e. a feedback loop between dynamic resource allocation and the sensed metric has to exist,
  - relevant, i.e. the metric has to reflect potential causes of interference.

To obtain such a metric, potential causes for interference in the sense of Definition 4.2 have to be considered:

- An application app_2 can introduce library conflicts or incompatible kernel parameters settings that cause runtime errors of app_1 or even prevent app_1 from starting.
- An application app_2 can introduce security risks that can be exploited by attackers. This may compromise the whole OS, e.g. leading to decreased performance. This will also affect app_1.
- An application app_2 can be buggy or badly coded, e.g. exhibiting a memory leak. This can severly decrease the amount of free memory available in the system. In order to resolve this issue, the OS will take counter-measures (e.g., "out-of-memory daemon" on Linux) which are known to reduce system performance temporarily. This will also affect the performance of application app_1.
- An application app_2 competes with app_1 for sparse resources. This may affect the performance of app_1.

The first three points can be avoided by using platform virtualization. Virtualization is an accepted technology to isolate applications and their processing environments from each other. More details are given later in Section 4.3.2. For the fourth point - competition for sparse resources - there is a metric that reflects this competition. Such competition is equivalent to the temporary existence of resource shortage (lack of free resources):

$$Usage_{res}(k,N) + \epsilon_2 > 100 \text{ , where } \epsilon_2 \text{ represents a safety buffer} \tag{4.1}$$

The previoulsy defined function $Usage_{res}(k,N)$ satisfies the criteria for a sensor metric. It is a quantitative metric and has a update frequency of 1/s. It does not depend on any specific application and can be retrieved by means of the OS. The development of metric values over time can be influenced by dynamically granting and revoking access to resources for SecApps. It also reflects a potential cause for interference, the competition for sparse resources.

Preventing resource competition is difficult: In context of this thesis, preventing resource competition on a single node means giving the locally running MainApp absolute preceedence over a locally running SecApp. That means, if both a MainApp OS process and a SecApp OS process request access to the same resource at the same time, the MainApp OS process is always granted access. The decision which OS process is given which resource share at which time is the responsibility of the kernel. The kernel coordinates the access to devices and resources and therefore employs several subsystems like a process scheduler for granting access to the CPU, an I/O scheduler for controlling disk access and a packet scheduler for managing network communication. For ensuring that these subsystems service requests from the SecApp only if there is no request from the MainApp, one needs to configure the scheduling used by these subsystems. It was mentiond in the first paragraph of Chapter 3 that modifying or patching the kernel strategies (such as OS process scheduling policies) is not appropriate for achieving the thesis goal because this approach lacks generality and may influence the system behavior in a way not permitted by the operator of the dedicated cluster. Nevertheless, there are Linux user space tools which to some extent help to realize that a MainApp preceeds a SecApp: The tool "nice" can be used for tuning the priority of an OS process. By giving a SecApp OS process the lowest possible priority, the process scheduler will assign this process only the smallest possible time slice (e.g. 20ms, depends on OS) and make it receive only a little share of the CPU resource over time. Unfortunately, for the standard OS process scheduling policy (SCHED_OTHER in Linux), the actually applied priority is periodically recalculated (usually every 20ms). If an OS process does have network or disk I/O, then the process priority will improve over time. This will give a SecApp a bigger share of CPU than originally intended by setting the lowest possible OS process priority. For disk access, a tool "ionice" exists that may be used for modifying the way the I/O scheduler grants access to disk resources for specific OS processes. Again, a low priority can be set for an OS process, making it less likely to receive read or write access to the disk. As with the CPU counterpart "nice", this tool can be used to influence the share of disk I/O attributed to a OS process, but cannot give guarantees that the MainApp always preceeds a SecApp. For network communication, things are more complicated: Here tools like "iproute2" and "tc" allow to classify certain packages, e.g. those originated by a SecApp, in specific Quality of Service (QoS) classes and to modify the scheduling policy for these classes ("traffic control"). This, however, is non-trivial to configure, not recommeded for production systems and might require changes to the system not allowed by the dedicated cluster operators.[7] Since SecApps will be hosted by VMs, it depends on the hypervisor of the virtualization product to which extent the application of these tools will also have an effect on the resource shares used by VMs. For distributed SecApps, hosted by multiple VMs on different physical nodes, an additional problems arises: If on one node a VM is given few or no resources, this VM may appear as an unavailable or crashed communication partner to other VMs. This may break the semantics of the distributed SecApp hosted by these VMs.

The bottom line of this discussion is: There are tools that can be used for giving SecApps less resources than the MainApp. Nevertheless, no guarantees can be given that no resource competition occurs. Since a proprietary approach (e.g. patching the kernel) is not allowed and existing tools do not suffice to avoid resource competition in total, the strategy how to deal with resource competition is phrased as follows:

Instead of trying to prevent any resource competition, it is the goal to reduce the probability that resource shortage occurs.[8] That means that potential resource shortage should be anticipated and counter-measures have to be taken. If resource shortage occurs, then it has to be resolved within a specific time frame.[9] The following proposal is made: The metric $Usage_{res}(k, N)$ is used to monitor the current resource usage. Based on this metric, policies are defined that constrain to what extent a SecApp is allowed to use resources, i.e. policies cap the share of resources usable by SecApps. If the total usage of a resource exceeds a certain threshold, a policy defines for how long any SecApp is still allowed to use that resource. Once that time elapses, any SecApp must stop using this resource. The to-be-developed framework has to realize the semantics of these policies. In this thesis, it will be

[7]For more information on kernel-level scheduling please refer to the Linux documentation [71]. Information on the mentioned tools can be found on their corresponding Linux man pages.

[8]For a scenario of duration t seconds, a retrieval of $Usage_{res}(k, N)$ is executed every second. If n is the number of times the Equation 4.1 is evaluated to true, then n/t should be small.

[9]For a scenario of duration t seconds, a retrieval of $Usage_{res}(k, N)$ is executed every second. If Equation 4.1 is evaluated to true for a point in time k, then the number i of successive points in time $(k + 1, k + 2, ..., k + i)$, for which Equation 4.1 is also evaluated to true should be small.

shown *a*) how the metric $Usage_{res}(k,N)$ can be used for policies, *b*) how policies are evaluated and decisions are based on these and *c*) how the amount of interference can be controlled by tuning policy parameters.

To avoid over-simplification some final remarks need to be made: First, not every resource that processes are competing for can be modelled or its usage can be influenced in a controlled manner. Depending on the granularity taken into account even a software mutex, a CPU cache or a CPU register can be regarded as a resource. So, there are hidden, non-modelled resources which OS processes compete for. Second, there are interdependencies between resources. For instance, while a single-threaded OS process is waiting for its currently being processed I/O request to be completed, it will not use CPU resources. So the usage of a disk resource is related the usage of the CPU resource. Another example are OS processes located on different nodes in the cluster that concurrently access a shared storage facility, e.g. a Storage Area Network (SAN). Here a bottleneck at the SAN will impact how these OS processes make use of their local CPU resources. A third example is that of a parallel MPI application that has barriers.[10] Such barriers synchronize the processing on one node with the processing on other nodes, which means that the resource usage on these nodes has an interdependency. Since not every possible resource and no interdependencies between resources are modelled and accounted for, it is virtually impossible to fully control the amount of interference by basing decisions on the usage of modelled resources only.

Nonetheless, it is assumed that the metric $Usage_{res}(k,N)$ is a reasonable sensor metric even if only applied to few resources like CPU, NetOut and NetIn. Further resources of interest are mentioned later in Section 4.3.5. Empirical tests will be made to evaluate whether controlling the SecApp usage of these resources using policies indeed allows to control the amount of interference.

4.3.2. Virtualization as Enabling Technology

Instead of running SecApps natively on the cluster nodes, in this thesis it is proposed to run SecApps using platform virtualization, i.e. SecApps run inside of VMs. This has specific advantages:

- Virtual machines act as a sandboxed container, encapsulating SecApps. This cleanly separates a MainApp running on the native cluster node from SecApps running inside of VMs. This isolation is provided by the virtualization platform and the effects of buggy or insecure SecApps are thereby contained to the VM and cannot cause interference towards a MainApp. Security and isolation using VMs are subject to continuous research and improvement. Please refer to Chen at al. [25] for more information on current achievements and challenges of this topic.
- VMs can host a multitude of different operating systems and applications. This broadens the spectrum of SecApps that can be run in a cluster. This aspect goes far beyond the trivial example of running a Windows-based software in a Linux cluster. As an example, in the HLT-Cluster a potential candidate for a SecApp (AliEN[10]) could not be run due to library conflicts.[11] Even though such an issue can be resolved with sufficient manpower and effort, in practice such seemingly minor issues generate inacceptable additional overhead. Virtualization therefore

[10]A barrier is an MPI primitive to synchronize distributed MPI processes. This will make MPI processes wait for each other.

[11]This observation was made for a previous Ubuntu-OS and AliEn version in 2009 and may not be relevant for current versions.

not only enables to run different operating systems but also different incarnations of the same operating system that are explicitely tailored to a single SecApp.

- VMs allow to define a maxmimum on the amount of certain resources used by a VM. For instance, prior to running a VM, the amount of RAM and the number of CPU cores usable by this VM can be defined. At runtime, the VM will consume no more than the specified maximum for these resources. This allows to statically cap the resource usage of hosted SecApps. Thereby even a memory leak in the virtualized application is contained to the VM only.
- VMs offer an abstracted interface[12] for modifying hosted SecApps, and thereby also their resource usage. Most VM products offer the complete set of manipulations like start, stop, suspend, resume and cold, warm and hot migration. This enables a flexible manipulation of SecApps without relying on application-specific support and interfaces.
- In addition to this abstraction which allows for a set of standardized manipulations of SecApps, another aspect is important. Any such manipulation has advantages and disadvantages regarding:
  - Their effect towards SecApp state. Stopping a VM will cause the stopping of the hosted application and a loss of not yet finished or saved computations. Suspending a VM however allows for a flawless later continuation of computations which were still ongoing when the VM was suspended.
  - Their overhead or effect towards resource usage while being applied. Suspending or migrating a VM requires the transfer of state information over network and therefore stresses this specific resource. On the contrary, stopping a VM only uses few CPU and network resources. Modifying the CPU usage of a single VM by adding a CPU core or by using Linux signal sequences (see Section 4.4.4) practically has no overhead at all.
  - Their duration. Suspending or migrating a VM may take several seconds while the hard stopping a VM requires only some milliseconds.

  It is therefore vital to have a range of different manipulation primitives to choose from according to the circumstances. VMs exactly provide this flexibility.

There are multiple virtualization technologies, products and vendors. Common technologies and products are briefly outlined in Section A.4. These products considerably differ in nomenclature and functionality. In this thesis an abstracted and simplified naming convention is used: A software entity governing states and properties of single VMs on a physical node is called "hypervisor". Via its API a hypervisor offers primitives to manipulate VMs. The main primitives used in this thesis are $\{start, stop, suspend, resume, migrate\}$. The correspondence of this naming scheme with the different vendor terminologies is given in Section A.4. Regarding the properties of a VM, it is assumed that a VM has a number of virtual CPU cores, an amount of allocated RAM, a permanent storage (virtual disk), network interfaces and a specific operating system setup. The mentioning of product-specific techniques and implementation of these properties is avoided whenever possible[13], because the design principles of the framework are independent from the used virtualization product.

[12]Abstracted interface means that no application-specific commands need to be used when modifying e.g. the state of an application.

[13]For example, the permanent storage of a VM can be on a local or remote disk, it can be realized via file or block-level access, it can be pre-allocated or allocated on-demand, its size can be modifiable or fixed throughout the life-time of a VM.

4.3.3. Virtual Machine Life-Cycle

With respect to the definitions made in Section 4.2, a VM is a process.[14] A VM has a state and can be allocated to at most one node at a time.[15] The decision instance decides on allocations of VMs to physical nodes and puts the decisions into practice by means of the local hypervisors. A VM furthermore has a specific life-cycle that describes the states a VM can be in and the state transitions. This life-cycle is shown in Figure 4.1 and described in the following. The life-cycle is administrated by the decision instance. The decison instance uses hypervisor calls to initate most state transitions.[16] A

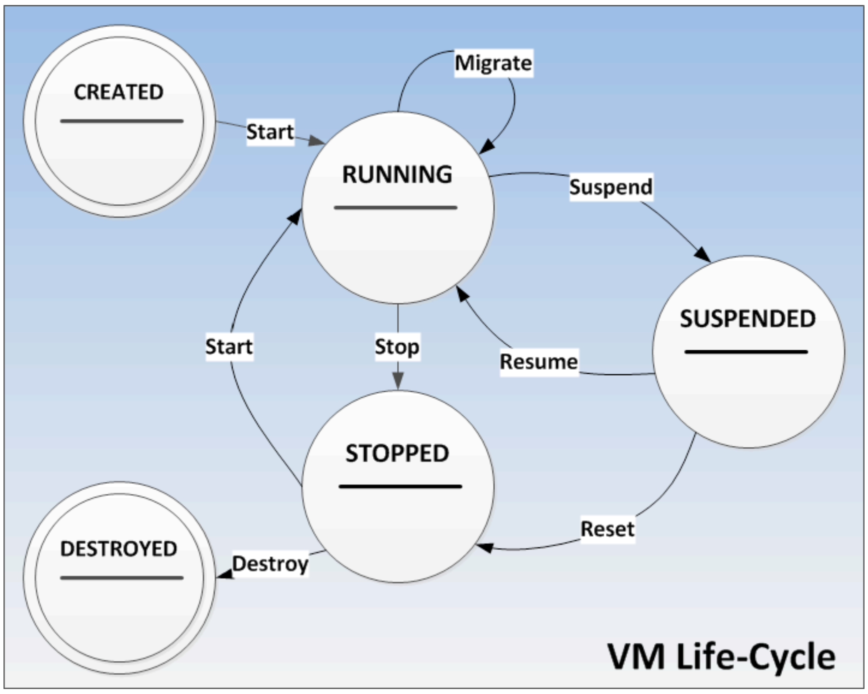


Figure 4.1.: Life-cycle of a VM: States and transitions.

VM is created by means, i.e. using the API, of a hypervisor. Along with a VM's creation its initial properties (CPU, RAM, storage, network) are defined and the VM is registered with the decision instance. Created VMs, i.e. their storage (virtual disk) and meta-data, are physically located on a shared storage medium and therefore instrumentable by all hypervisors on nodes in the cluster.

A created VM can be *started*. Starting a VM involves the allocation of the VM to a specific node by the decision instance and the actual start up of the VM via a call to the local hypervisor. This changes the state of the VM to "running". Now the VM uses resources of that node. A running VM can be stopped, suspended or hot migrated.

[14] As a reminder: A process subsumes all local operating system processes and threads that realize a common functionality on a single node. For the virtualization product VirtualBox such a VM process indeed corresponds to a single OS process.

[15] The allocation relation $alloc(VM, node)$ was defined in Section 4.2.

[16] The state transition from "running" to "stopped" that can also be realized without calling the hypervisor, e.g. by using Linux signals.

Stopping a running VM puts the VM in state "stopped" and dissolves the allocation relation between the VM and its allocated node. The stopping can by done by *a*) using a hypervisor call "soft-stop" which equals a warm shutdown of a computer system, *b*) using a hypervisor call "stop" which equals a cold shutdown of a computer system or *c*) using OS signals . The latter can be compared to unplugging a running computer system from the mains. In any case the current state of the hosted SecApp is lost, meaning that computations which had not been finished, saved or checkpointed are wasted.

Suspending a running VM puts the VM in state "suspended" and dissolves the allocation relation between the VM and its allocated node. A hypervisor call "suspend", which can be compared to a suspend-to-disk for computer systems, will cause the state[17] of the VM to be saved. So a flawless later resumption is possible without spoiling ongoing computations.

Migrating a running VM will not change the state of the VM. This transition does however dissolve the allocation relation between the VM and a current node and establishes an allocation relation between the VM and a different node. Thus, the VM keeps on computing but changes the host is is allocated to and whose resources it consumes.[18]

A suspended VM can be *resumed*. Resuming a suspended VM involves the allocation of the VM to a specific node and a "resume" call to the local hypervisor. It can be compared to opening a previously to-disk-suspended (closed) notebook. The VM and its hosted SecApp continue their processing as if no previous suspend/resume had taken place.

A suspended VM can be *reset*. This causes the VM to change its state to "stopped". The saved state[19] of the previously suspended VM is discarded.

A stopped VM can be *started*. Starting a VM involves the allocation of the VM to a specific node and the actual start up of the VM via a call to the local hypervisor. It can be compared to pressing the power button of a switched-off computer system, causing the operating system to boot from scratch. This changes the state of the VM to "running". Now the VM uses resources of that node.

A stopped VM can be *destroyed*. This involves the deregistration of the VM from the decision instance and the deletion of its virtual disk and meta-data.

Summing it up: Virtual Machines provide an isolated environment for SecApps. The life-cycle of VMs is administrated by the decision instance. VMs offer standardized and flexible ways of manipulating hosted SecApps and accordingly, their resource usage. State changes are primarily put into practice using API calls to the hypervisor(s) of a virtualization platform. The different VM state transitions have specific advantages and disadvantages regarding overhead, duration and influence on the computational state of hosted SecApps. Apart from VM state transitions there are other methods to modify the resource usage of SecApps. Such methods are either product-specific (like changing the number of CPU cores at runtime for XEN) or general OS means (like capping the CPU usage of a VM). The potential usage of such additional methods is discussed in Section 4.4.4.

[17]Not to be confused with a VM's state in the life-cycle. Here all information needed for a later resume, e.g. the memory contents, CPU queue and registers, I/O queues, et cetera are meant.

[18]Technically this involves minimal interruptions in the computation of SecApps, please see [119, 3] for details.

[19]Again, this is not the VM's state in the life-cycle. Instead a file containing information on how to resume the VM is deleted.

4.3.4. Secondary Application Abstraction

VMs offer flexible and uniform methods for encapsulating arbitrary non-distributed SecApps and manipulating their resource usage. They are, nonetheless, not sufficient for fully abstracting general SecApps. First, VMs merely provide a hosting environment similiar to a physical node with an operating system setup. No meta-information about the hosted SecApp like priorities, user privileges, credentials, runtime conditions or even SLT requirements come with a VM's description. Second, a VM represents only a single virtualized node. No distributed applications, e.g. an MPI application, can be abstracted using the VM concept only. The "lease" concept, inspired by Grit et al. [49] is therefore used in this thesis. A lease is a container for a set of VMs running a single, potentially distributed SecApp.[20] The relation $assign(lease, \{VM\})$ defines a temporary correspondence between a lease and a set of assigned VMs. At a specific point in time a VM can only be assigned to at most one lease. A lease has multiple properties. For now these are restricted to few basic properties which are the (lease) owner, the (lease) priority and the number and type[21] of assignable VMs. The decision instance then regards the set of assigned VMs of a lease as a unit, i.e. it makes sure that all assigned VMs are in the same state for any given point in the lifetime of a lease and accord with the requirements given by the lease's properties. The life-cycle of a lease is shown in Figure 4.2. There is a correspondence

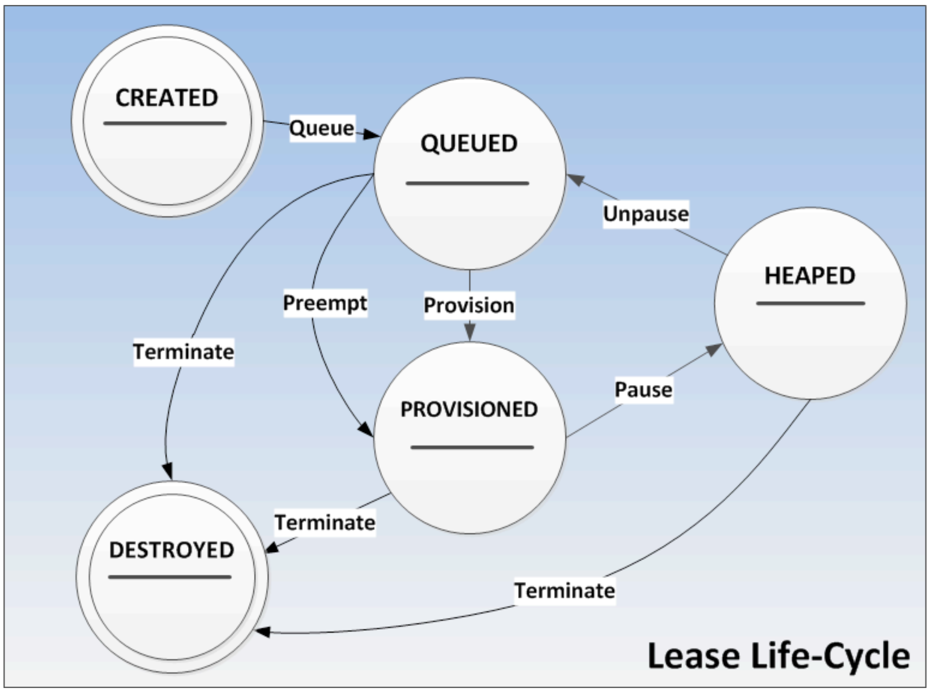


Figure 4.2.: Life-cycle of a lease: States and transitions.

between the life-cycles of leases and VMs. The life-cycle of a lease is now explained, corresponding VM state transitions are mentioned:

[20]Like a VM can be considered as a process according to the description given in Section 4.2, a lease corresponds to an application mentioned ibidem.

[21]The type of a VM simply denotes which SecApp is run by the VM and what configuration (CPU, RAM, etc.) the VM possesses.

When a user wants to run a SecApp, he submits a request to the decision instance. The request is evaluated and a lease is *created*. The lease is equipped with meta-information which is the priority, the owner[22] and information about to-be-assigned VMs. The lease is now in state "created". Upon triggering of an event internal to the decision instance, e.g. a certain date, the lease is transferred to state "queued".

Leases in queued state are those leases that are waiting to be provisioned to the cluster. Once the decision instance decides that for a queued lease *a*) there are appropriate usable resources in the cluster and *b*) there is a sufficient number of appropriate VMs available , the lease will be *provisioned*. The transition to state "provisioned" involves the assignment of VMs to the lease, the allocation of these VMs to nodes and the starting of these VMs.Throughout this thesis, the phrase "starting a lease" is repeatedly used to refer to this procedure.

It is possible that a lease in queued state already has VMs assigned to it. These VMs are in suspended state.[23] In this case the transition to state provisioned only involves the allocation of VMs to nodes and the resuming of these VMs. Again, the phrase "resuming a lease" is used to refer to this procedure. The transistion of a queued lease to provisioned state will cause the assigned VMs to attain running state.

A lease in queued state can be *terminated*. This can be due to the owner's explicit request or upon an event internally triggered by the decision instance (e.g. a date). If the lease has assigned VMs, then this relation is dissolved and these VMs are reset. Then the lease is destroyed, i.e. its data residuals are deleted.

A lease in provisioned state can be *preempted*. This is done by the decision instance and transfers the lease to queued state. Depending on the type of preemption any of two things can happen: Either all assigned VMs are stopped, their allocation relations to nodes and the assignment relation to the lease are dissolved. Or all assigned VMs are suspended, their allocation relations to nodes are dissolved but the assignment relation to the lease is maintained. The former is called "stopping a lease", the latter is called "suspending a lease". The phrase "preempting a lease" is used when referring to any of these procedures.

A lease in state "provisioned" can be *terminated*. Again, this can be due to the owner's explicit request or upon an event internally triggered by the decision instance (e.g. a date). In both cases all assigned VMs are stopped, their allocation relations to nodes and the assignment relation to the lease are dissolved and the lease is destroyed, i.e. its data residuals are deleted.

A lease in provisioned state can be *paused*. This only happens upon an owner's request. Such a request means that the owner wants a provisioned lease to be paused temporarily, i.e. to stop the SecApp from running in the cluster, but without terminating the lease. Pausing a provisioned lease will put the lease in state "heaped". All assigned VMs are suspended, their allocation relations to nodes are dissolved but the assignment relation to the lease is maintained.

A lease in heaped state can be *unpaused*. This only happens upon an owner's request. Such a request

[22]The user that submitted the request becomes the owner of the lease.
[23]This is the case if a previously provisioned lease was preempted or paused and later unpaused.

means that the owner wants a currently paused lease to be provisioned in the cluster again. The lease will attain queued state, which means that it is put under control of the decision instance which again decides on its provisioning.

A lease in heaped state can be *terminated*. This only happens upon an owner's request. All assigned VMs are reset, their allocation relation to nodes and the assignment relation to the lease are dissolved and the lease is destroyed, i.e. its data residuals are deleted.

The lease concept will not be defined in a more formal way. A lease merely represents a container of VMs, runs a specific SecApp, can be in different states and carries meta-information. Leases are administrated by the decision instance. Their state transitions are triggered by the owner of a lease or by the decision instance. It is assumed that the VMs appropriate for assignment to a lease are exisiting throughout the runtime of the decision instance, i.e. they had been created beforehand. That is, the creation of VMs is not captured by any phase of the lease life-cycle. Existing approaches [93, 82, 28, 110] regarding runtime creation of appropriate VMs ("appliance automation") are not considered relevant in context of this thesis.

4.3.5. Constraints on Resource Usage: Policies

So far resource producers (e.g. Nodes), resources (e.g. CPU) and resource consumers (VMs, leases) have been introduced. Their relations have been described in a basic manner: leases make up of VMs, VMs are allocated to nodes. The allocation of a VM to a node causes the VM to use resources of that node. Decisions on establishing or dissolving assignment and allocation relations are made by a decision instance. This decision instance incorporates resource usage data to make such decisions. Policies are statements about the desired usage of resources, that guide the interpretation of current resource usage values, and drive the decisions of the decision instance. Before formalizing the policy concept the terms "allocation pattern" and "cluster model" are introduced.

Allocation Pattern and Cluster Model It was mentioned before (see Section 4.3.1) that in order to control the amount of interference, resource usage in the cluster has to be controlled such that competition for sparse resources can be avoided or temporally limited. The only way to control resource usage in a dedicated cluster without control over the MainApp is to modify the resource usage generated by SecApps. VM life-cycle transistions have been introduced as a method to modify the resource usage of SecApps. A decision instance will therefore make use of the VM manipulations $\in \{start, stop, suspend, resume, migrate\}$. To describe a current state of leases, VMs, nodes and their relations the term allocation pattern is introduced. An allocation pattern is a snapshot of the cluster's nodes, leases and VMs taken for a single measurement interval Δt. Such an allocation pattern AP is defined as

Definition 4.4 $AP := \{\, (lease, \{VM\}) \mid assign(lease, \{VM\}) \,\} \cup \{\, (VM, node) \mid alloc(VM, node) \,\}$

The state of all leases and VMs contained in an allocation pattern is implicitely given and can be derived from the VM and lease life-cycle definitions stated above. Since an allocation pattern is defined as a snapshot of the cluster with its nodes, leases and VMs for a specific measurement interval, an order relation (<) can be imposed on allocation patterns, defining a timeline: $AP_k < AP_{k+1}$ with $\Delta(t_k, t_{k+1}) = 1s$. AP_k is said to be equal to AP_{k+i} if there is a bijective mapping f such that for every assignment and allocation tuple in AP_k there is a tuple in AP_{k+i} with f being the identity function.

Definition 4.5 *The transition of one allocation pattern to another: $AP_k \mapsto AP_{k+i}$ with $AP_k \neq AP_{k+i}$ is called change of an allocation pattern. It is achieved by establishing, modifying or dissolving allocation and assigment relations.*

The change of an allocation pattern is merely the execution of lease and VM state transitions. It may involve VM or lease state changes (e.g. for stopping a lease), but not necessarily needs to (e.g. migration of a VM). In this thesis, the terms "manipulating a lease", "manipulating a VM" or simply "manipulation" are often used to denote single state transitions of VMs and leases. An allocation pattern describes the relation between leases[24], VMs and nodes and therefore represents a state directly manipulable by a decision instance. There are other interesting properties of a cluster environment which are not represented by an allocation pattern, like modelled resources and their usage. A more comprehensive description of a cluster, its resources, existing leases and VMs is the cluster model. Again, a cluster model is a snapshot of the cluster taken for a single measurement interval Δt.

Definition 4.6 $CM := \{\{cluster\ item\}, \{(res, Usage_{res})\}, \{VM\}, \{lease\}, AP\}$

In this definition, $\{cluster\ item\}$ is the set of all known cluster items (e.g. nodes) and $\{(res, Usage_{res})\}$ is the set of all modelled resources and their usage. AP denotes an allocation pattern. In analogy to the allocation pattern the cluster model is situated in time, CM_k. A specific cluster model CM_k at a point in time t_k is also called a cluster model incarnation.

Policies The policy concept has been developed in order to equip the decision instance with criteria to assess whether further SecApps can be run, the share of resource usage generated by SecApps is appropriate (with repect to resource competition) or needs to be modified. A policy is a statement on the usage of a resource and specifies for how long SecApps are allowed to contribute[25] to a certain level of resource usage.
The policy $P_{CPU} = (CPU \leq 80, 10) \times (Node_4, Node_5)$ represents a typical example. Roughly it means that the CPU utilization on $Node_4$ and $Node_5$ must not exceed 80% for more than 10 seconds while SecApps are using this resource. The decision instance has to realize these semantics by adapting the CPU usage generated by SecApps (VMs) running on these nodes. The value 80 is called a threshold on the usage of resource CPU, the value 10 is called timelimit. The threshold and timelimit parameters are reconfigurable. This can be done prior to running SecApps or at runtime of the framework. A policy therefore is a configuration item which can be used to specify constraints on desired resource usage in a cluster.

More formally a policy P is characterized such: Let S be a predicate-logical statement that is interpreted in the cluster model. S has the form $S := \forall x \in CI : A(x)$. CI denotes a family of cluster item sets, e.g. $CI = \big\{\{Node_1\}, \{Node_2\}\big\}$.[26] Every $x \in CI$ provides a resource. A(x) is a statement about the usage of these resources. The evaluation of both $A(x)$ and S is bound to a specific cluster model incarnation CM_k, i.e. $eval : \{(S, CM_k)\} \mapsto \{True, False\}$. If there is no VM process using the resource provided by a specific x then $A(x)$ is $True$. If a VM uses this resource, then the truth value of $A(x)$ depends on the actual usage of the resource given by CM_k. An evaluation $eval(S, CM_k)$ is called a model-check. Assuming a measurement interval of $\Delta t = 1s$, a model-check occurs every second. A

[24] According to definition, stopped (queued), created and destroyed leases are not part of the allocation pattern. The same holds for non-assigned VMs

[25] A VM contributes to the usage of a resource res at point in time t_k iff $Usage_{res}(k, N, VM) > 0$.

[26] CI is a family of sets (instead of a set) because a resource can also be defined as a "cumulated" resource of multiple cluster items. An example is the total number of VMs running in a subcluster.

policy P is now defined as a tuple $P = (S, CI, n)$ with S being the statement over the cluster model incarnations, CI being the family of cluster item sets for which the model-check is done and $n \in \mathbb{N}$ representing a number of model-checks. A policy can either be valid or invalid. A policy is valid unless it has been found invalid at any point in time. A policy is invalid if

$\exists m : m \in \mathbb{N} \wedge \bigvee_{i=m}^{m+n} eval(S, CM_i) = False$. The goal of the decision instance is to keep policies valid.

The formalism is now explained using an example. Let $P = (S, CI, n)$ with $CI = \left\{\{Node_1\}, \{Node_2\}\right\}$, $n = 5$ and $S := \forall x \in CI : Usage_{CPU_x}(k, 1) \leq 80$. The statement S itself can be interpreted as: The usage of the CPU resource on $Node_1$ and $Node_2$ within the last measurement interval is below 80%. At a point in time t_{k+i} the policy P itself is valid if including the first model-check at $k = 1$ no six consecutive model-checks returning $False$ have occurred. This practically means that no node had a CPU utilization over 80% for more than 5 seconds while some VM was using the CPU resource. In simple words, a policy makes a statement on the resource usage in a cluster and additionally specifies "how long" this statement can be $False$ until the policy is considered invalid. A less complicated terminology to describe policies is used from now on:

$P_{CPU} = (CPU \leq 80, 10) \times (Node_4, Node_5)$ means that the CPU utilization on $Node_4$ and $Node_5$ must not exceed 80% for more than 10 seconds while a SecApp is using this resource. The statement $A(x)$ (here: $CPU \leq 80$) defines the threshold[27] and n (here: 10) is defines the timelimit. Since timelimits were introduced as a number of consecutive model-checks and a model-check is done every second, the timelimit value is interpreted as a time interval in this thesis, e.g. the timelimit in $P_{CPU} : (80, 10)$ is 10 seconds. When emphasizing the fact that the policy specification contains specific elements $x \in CI$ the phrase: "The policy is associated with x" is used. In this particular example the policy is associated with $Node_4$ and $Node_5$. At times an even shorter terminology $P_{CPU} : (80, 10)$ is utilized when discussing policies without mentioning associated cluster items. The operator "≤" is implicitely assumed for A(x) when such a simplified form is used.

Example Resources Policies are defined over resources and their usage. Throughout the previous sections the normalized function $Usage_{res}$ was used to refer to the usage of a resource. When applying the normalized function to an inhomogeneous cluster, e.g. consisting of nodes with a different number of CPU cores, this leads to information loss: 80% CPU usage on a 16 CPU core node is different from 80% CPU usage on a 2 CPU core node. When specifiying policies there needs to be an agreement for every resource type on whether the normalized (relative) $Usage_{res}$ or the absolute resource usage $currentUsage_{res}$ is used. In this thesis, the normalized, unitless function $Usage_{res}$ is used for the resources CPU, NetIn, NetOut, DiskIn and DiskOut when specifying policies. That is, thresholds are interpreted as percentage values for these resources. For any other resource, the absolute functions $currentUsage_{res}$ and $currentProcUsage_{res}(k, N, proc_i)$ are used instead. In this case thresholds are interpreted using their native units, e.g. MByte for a threshold on memory usage. Table 4.1 shows several exemplary resources with their producers and native units for the absolute functions $currentUsage_{res}$ and $currentProcUsage_{res}(k, N, proc_i)$. For all resources the measurement interval of $\Delta t = 1s$ is used and their usage functions are defined using the average over the last measurement interval, e.g. $currentUsage_{res}(k, 1)$.

[27]Despite of choosing the term threshold that does not mean that A(x) needs to be defined upon metrically scaled variables.

Resource	Producer	Explanation	Unit
CPU	Node	CPU Utilization	Jiffies/s
Mem	Node	Allocated memory	MByte
NetIn	Node	Incoming traffic on eth0	MByte/s
NetOut	Node	Outgoing traffic on eth0	MByte/s
DiskIn	Node	Disk0 write rate	MByte/s
DiskOut	Node	Disk0 read rate	MByte/s
AllocVM	Node	# VMs allocated	None
UsedCore	Node	# Cores used by allocated VMs	None

Table 4.1.: Example resources, their producer and intended meaning.

The resources CPU[28], Mem[29], NetIn, NetOut, DiskIn, DiskOut refer to architectural components of a computer system. Their maximum usage is restricted mainly by the properties of the hardware.[30] On the contrary, the resources AllocVM and UsedCore are (artificially composed) synthetic resources. AllocVM can be interpreted as allocation slots for VMs on a single node. Likewise UsedCore denotes the number of CPU cores usable by (virtual CPU cores of) VMs on a single node. Of particular use is a policy on the AllocVM resource: Since policies can be modified at runtime, such a modification from e.g. $P_{AllocVM} : (2, 10)$ to $P_{AllocVM} : (0, 10)$, will enforce the removal of all VMs on the associated nodes with 10 seconds.[31] A closer look at the relation between policies and their procedural semantics, i.e. how they trigger and influence the choice of manipulations will be taken in Section 4.4.

4.3.6. Decision Making Architecture

So far the term decision instance was used for denoting an entity that makes decisions on when to grant or revoke access for SecApps to cluster resources. This term will now be refined. To start with there are two differing scenarios:

Scenario 1: There are queued leases. A decision instance strives to provision these leases, i.e. giving them access to computational resources. This is done by repeatedly choosing a queued lease and finding allocations for its assigned VMs. Knowledge of the cluster model incarnation and specified policies is required to make such decisions. No automated preemption of running leases is used, i.e. a run-to-completion scheme is employed. This scenario is a classical job scheduling scenario. A functional unit performing this task is called **Global Provisioner (GP)**. The goal of this functional unit GP is to make provisioning decisions in order to use free resources in the cluster and to allow SecApps to compute results. Policy semantics have to be obeyed. The GP is more closely described in the next Section 4.4.2.

[28] A jiffy is the duration of one tick of the system timer interrupt and can be used to measure the CPU utilization by calculating how many jiffies/s the CPU spent in various execution modes.

[29] The meaning of "allocated memory" depends on the OS, virtual memory subsystem and the programmer's intention. Here the meaning is the Linux-specific value retrievable by querying procfs memory statistics: $used - (buffers + cached)$.

[30] Properties of the specific operating system may also play a role, for instance the 4GByte memory limit of a 32bit Windows XP OS.

[31] Such a manual policy modification will trigger a change of the allocation pattern and is very handy when an administrator wants to free a subcluster from VMs.

Scenario 2: There are running leases. Unless paused upon (lease) owner request, these leases run till termination. However, policies impose restrictions towards the usage of resources. If resource usage increases over the threshold level, the contribution of SecApps (VMs) to this usage needs to be decreased. Basically there are two different ways of achieving policy compliance, i.e. of keeping them valid in a critical scenario (e.g. high CPU usage):

- changing the current allocation pattern or
- modifying the resource usage of single VMs directly

The change of an allocation pattern has been defined before (see Definition 4.5). Feasible lease and VM manipulations that could lead to a decrease in resource usage are stopping and suspending of leases and migration of single VMs. Choosing appropriate lease and VM manipulations that keep policies valid requires knowledge of the current cluster model incarnation and specified policies. A functional unit which makes such decisions on how to change an allocation pattern is the **Global Reconfigurator (GR)**. This unit recognizes when there is the need for a change, decides on which to choose and applies it. The GR is more closely described in Section 4.4.3. A vital role in this scenario plays the estimation of effects of VM manipulations towards the usage of resources. This estimation is discussed separately in Section 4.4.1.

Changing an allocation pattern in order to modify resource usage has disadvantages: First, it takes considerable time till its effects kick in (e.g. a migration may take several seconds), second, the manipulation itself consumes resources (e.g. a migration uses network resources) and third, such a manipulation involves communication between separate cluster nodes and may therefore fail, e.g. due to network, OS and hardware failures. There are other options to modify the resource consumption of running VMs: The attribution of resource shares can be adapted locally on the node which the VM is running on. This relieves the dependency on a remote decision instance and provides greater flexibility. Means provided by the hypervisor (e.g. change of virtual CPU cores) and the OS (e.g. Linux signals) can be used to modify the resource usage of a VM locally. Such an adaption does not rely on knowledge about the current cluster model incarnation (e.g. the resource usage on other nodes). Knowledge of policies that concern the local node is required. In contrast to the coarse-grained manipulations choosable by the GR, a fine-grained adaption of resource usage (e.g. CPU usage of a VM) is possible. A closed loop adaption of local resource usage bases on feedback control theory and therefore can react to changes in the total usage of local resources flexibly at runtime. A functional unit implementing such a local resource usage adaption is called **Resource Usage Adaptor (RUA)**.

Three funtional units have been introduced: GP, GR and RUA. These units constitute what has been called "decision instance" up to this point. Having described these units from a functional perspective, now a tentative introduction[32] is given for the architectural design of the software framework. The software framework will from now on be referred to by using the terms VM-Scheduler, VM-Scheduling Framework or simply framework.

[32]A more detailed coverage of the software architecture will be given in Section 5.2.

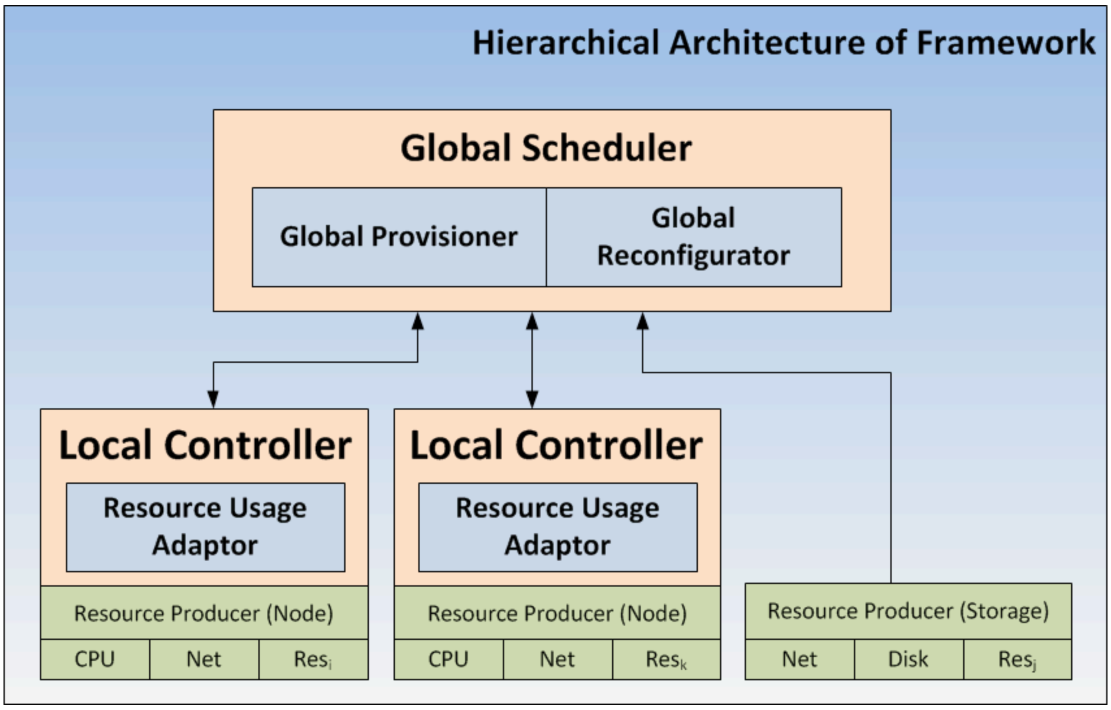


Figure 4.3.: Basic framework architecture with focus on distinction between Global Scheduler and Local Controller components.

Figure 4.3 shows the basic architectural components of the framework. GP and GR are part of the architectural component "Global Scheduler". This component is located on a global management server. Both units therefore possess a global view at the cluster and make decisions involving more than a single node. For instance, when a lease is about to be provisioned, target nodes (allocations) for all of its assigned VMs need to be found. A global instance (GP) with knowledge about the cluster state is therefore feasible. The RUA is part of the architectural component "Local Controller" depicted in Figure 4.3. Instances of the Local Controller are located on each node governed by the VM-Scheduling Framework. The knowledge scope of the RUA encompasses policies for the specific node and locally running VMs. Global Scheduler and Local Controller (i.e. GP, GR and RUA) interact by exchanging information about current resource usage, state of VMs/leases and currently active policies.

The architectural approach therefore is a hierarchical, two-layered one. The provisioning and preemption of leases is decided globally. Resource usage of running VMs is adapted locally unless a local adaption is not sufficient. This principle, subsidiarity, is considered a key factor to building an efficient framework. If local adaption does not suffice to achieve policy compliance, the GR steps in and overrides local adaptions via allocation pattern changes. Regarding the functional units hosted by the Global Scheduler the GP is subordinated to the GR which means that decisions by the GP (to provision a lease) can be blocked or canceled by GR for the sake of policy compliance, i.e. in order to prevent strong interference. In the next section, the algorithms and coordination of these functional units (GP, GR, RUA) are discussed. Since the identified main challenge for this thesis is to prevent a MainApp from being affected by running SecApps, this next section especially focuses on dynamic resource allocation realized by GR and RUA.

4.3.7. Summary

The basic concepts, terms and entities of this thesis were explained in this section. These are now summarized. The main challenge of this work is to avoid the potential impact of SecApps to a MainApp. This requirement for a valid solution was formalized using the interference concept. The argument was made, that the impact of SecApps to a MainApp requires the statistical evaluation of a performance metric of the MainApp. The developed interference terminology then allows to make statements on whether a MainApp is affected by additionally run SecApps, to which extent its is affected and whether this impairment leads to incorrect functionality. It was shown that the interference concept is feasible for ex-post assessments of past scenarios, but lacks as a base for runtime decisions on how to exploit unused resources. Potential causes of interference were considered and it was concluded that two necessary conditions have to be met in order to prevent strong interference: Isolation of the MainApp from SecApps and avoidance of resource competition.

To meet the first condition, the usage of platform virtualization was proposed for running SecApps: A SecApp runs in VMs, multiple VMs are assembled to a lease. A lease therefore represents a single, potentially distributed SecApp. A lease is equipped with additional meta-data that details requirements of the SecApp. Beside of the fact that virtualization efficiently isolates MainApp and SecApps and thereby confines negative effects of buggy and insecure software, the usage of VMs and leases also provides uniform and standardized interfaces for manipulating arbitrary types of SecApps.

To meet the second condition, the sensor metric $Usage_{res}(k,N)$ was introduced. This metric, i.e. the knowledge of the past and current resource usage, serves as base for making decisions on how, where and when to give SecApps access to resources. This metric has the advantage that it is MainApp independent, i.e. of generic nature, but at the same time also indicates when there is a potential risk of interference. It is desirable to prevent any kind of resource competition, i.e. when a SecApp is running on an node, the resource usage should be below 100% at all times. In this case, a SecApp does not steal resource shares, e.g. CPU cycles, from a MainApp. However, without intrusive kernel-level techniques that are not permitted in a dedicated cluster, this is difficult to achieve for some resources (e.g. network).

Resource competition is therefore counteracted by limiting the time span for which a SecApp is allowed to use a resource if the total resource usage exceeds a certain "threshold", e.g. 90%: If the threshold is exceeded, then the SecApp needs to reduce its used share of that resource. If the overall resource usage stays above that threshold for a certain time ("timelimit"), then the SecApp needs to reduce its share to 0%. This requirement is formalized using the configuration item "policy". For every resource, there is a policy with configurable threshold and timelimit parameters. By customizing these parameters, the impact of SecApps to a MainApp (amount of interference) can be controlled. This assumption is subject to later empirical evaluation (see Chapter 6). A two-layered architecture has been proposed for realizing the policy semantics and making runtime decisions on when to grant or revoke access to resources for SecApps. The functional units GP and GR are part of a "Global Scheduler". They decide on changes to the allocation pattern, i.e. they initiate lease state transitions and VM manipulations like start, stop, suspend, resume and migrate. The functional unit RUA is part of the "Local Controller" that runs on each cluster node. The RUA adapts the resource usage of locally running VMs. The existence and cooperation of these functional units serves the purpose of both keeping policies valid, hence controlling interference, while at the same time allowing the SecApp to use cluster resources and to generate additional results. The following section describes the functional units and their cooperation.

4.4. Resource Allocation Functionality

The VM-Scheduling Framework consists of two major architectural components: The Global Scheduler and the Local Controller. These host functional units, which are the the Global Reconfigurator (GR) and Global Provisioner (GP) on side of the Global Scheduler and the Resource Usage Adaptor (RUA) on side of the Local Controller. The algorithmic approaches chosen for these units and their coordination are portrayed in this section.

4.4.1. Feasibility Estimator

In the last section it was stated that in order to control interference, appropriate policies need to be specified and their compliance must be enforced. Both GP and GR make use of lease state transitions and corresponding VM manipulations $\in \{start, stop, suspend, resume, migrate\}$ for manipulating leases and realizing policy compliance. However, the execution of such manipulations carries a certain overhead towards resource usage. Based on this consideration the Feasibility Estimator (FE) is introduced. This facility is part of the architectural component Global Scheduler and used by both globally acting units GP and GR to assess the appropriateness of VM and lease manipulations with respect to policy compliance. In order to find changes of the allocation pattern that allow for provisioning of leases or prevention of policy invalidations, it is necessary to estimate the resource usage in the cluster model at future points in time. The Feasibility Estimator (FE) takes as input a proposed set of VM manipulations and predicts the resulting future resource usage under the assumption that these manipulations are instantly applied and no other changes of the allocation pattern occur. For a resource like AllocVM[33] this estimation is simple: If a single VM is to be suspended on a node, then the usage of AllocVM is decremented by 1 once the manipulation is finished. That is, the FE only needs to estimate the duration of manipulations to know the usage of a resource AllocVM at a future point in time. For resources like CPU, NetIn and NetOut this is more complicated: The manipulation itself, not just its outcome, also impacts the usage of these resources. In the following a simple model is given for calculating the duration of manipulations and their overhead towards resources CPU, NetIn, NetOut, thereby enabling the prediction of future resource usage on nodes. The model was not developed as a general purpose model to estimate these factors. The now presented model for estimating the duration and overhead is a prototypical approach for estimating these effects specifically for the test environment used in this thesis to evaluate the framework (see Section 6.3.1). A best-practice proposal of how to adapt the FE settings for a different cluster environment is given in Section A.5.

The FE is queried by functional units GP and GR at time t_k. These pass a set of proposed VM manipulations, a future point in time t_{k+i} and a resource as parameters to the FE. The FE returns the predicted usage of that resource at t_{k+i}. The FE knows about the current cluster model incarnation and currently being executed ("ongoing") VM manipulations at time t_k. The starting point to describe the functionality of the FE is to capture the concept of VM manipulations using the process and resource usage notion introduced in Section 4.2. A VM manipulation is a state transition in the life-cycle of a VM. Such a transition has a duration. The duration of a single VM manipulation is described by its starting point t_a and its ending point t_b with $duration = t_b - t_a$. As long as a VM manipulation is not finished the VM is considered to be in its old state. A manipulation $\in \{start, stop, suspend, resume, migrate\}$ is ongoing at t_k if $t_a \leq t_k \leq t_b$.[34] During a manipulation, there exists a manipulation process on the concerned

[33]The resource AllocVM represents a number of slots for running VMs on a node. See Table 4.1 for reference.
[34]For ease of understanding, points in time and manipulations are not indexed in order to refer to each other.

node $manipProc \in \{startProc, stopProc, suspProc, resumeProc, migOutProc, migInProc\}$ that realizes the primitive. These manipulation processes are abstractions and do not necessarily exist as separate OS processes. For a migration there are two such processes, migOutProc on the source node and migInProc on the target node. While all manipulations processes cause a certain CPU usage, some processes like migInProc or resumeProc generate usage of the resource NetIn and others like migOutProc and suspProc generate usage of the resource NetOut on a node.

The Equation 4.2 is used by the FE at a point in time t_k to calculate the usage of a resource at a future point in time t_{k+i}.

$$\begin{aligned} Usage_{res}(k+i,1) < Min\Bigg(100, \Big(& ProcUsage_{res}(k+i,1,MainApp) \\ & + \sum_{n=1}^{N} ProcUsage_{res}(k+i,1,VM_n) \\ & + \sum_{r=1}^{R} ProcUsage_{res}(k+i,1,VM_r) \\ & + \sum_{m=1}^{M} ProcUsage_{res}(k+i,1,manipProc_m)\Big)\Bigg) \end{aligned} \tag{4.2}$$

This equation consists of four $ProcUsage_{res}$ terms that as a sum represent the usage of a resource res at a future point in time t_{k+i}. These terms are now described. A specific term is referenced by the line number in which it occurs in the equation, e.g. line 1 references $ProcUsage_{res}(k+i,1,MainApp)$.

Line 1 This term represents the resource usage of a running MainApp at t_{k+i}.

Line 2 This term represents the summed resource usage of every VM_n that is in running state and contributes to resource usage at t_k and still runs or is subject to an ongoing manipulation at t_{k+i}.

Line 3 This term represents the summed resource usage of every VM_r that at t_k does not contribute to resource usage, but at t_{k+i} is expected to be running and contributing to resource usage (e.g. newly started, resumed or migrated VMs).

Line 4 This term represents the summed resource usage of every ongoing manipulation $manipProc_m$ at t_{k+i}.

At point in time t_k the FE only knows the current resource usage values for the MainApp and running VMs. Furthermore the FE knows about ongoing manipulations and proposed manipulations.[35] For calculating the future resource usage using Equation 4.2, it is therefore required to estimate the future resource usage values represented by the four $ProcUsage_{res}$ terms. It is now stated how these estimations are made.

- The future resource usage of the MainApp at t_{k+i} (represented by the term in line 1) is projected by using the MainApp's resource usage at t_k averaged over the last 5 measurement intervals: $ProcUsage_{res}(k+i,1,MainApp) = ProcUsage_{res}(k,5,MainApp)$
- The future resource usage of any VM that is in running state at t_k and is still running or being manipulated at t_{k+i} (represented by the term in line 2) is projected by using the VM's resource usage at t_k averaged over the last 5 measurement intervals: $ProcUsage_{res}(k+i,1,VM_n) = ProcUsage_{res}(k,5,VM_n)$

[35]Proposed manipulations are manipulations transmitted as parameters by the caller of the FE.

- The future resource usage of a VM that does not run at t_k but is expected to be running at t_{k+i} (represented by the term in line 3) for a resource like Mem was defined upon creation of that VM, e.g. 1GB. For resources like NetIn, NetOut and CPU, where no maximum usage can be defined upon creation, the future resource usage of a VM is 20%: $ProcUsage_{res}(k+i, 1, VM_r) = 20$. Every such newly running VM is subject to resource usage adaption by the functional unit RUA. The value of 20% has shown to be a feasible estimation for the inital resource usage of newly started, resumed or immigrated VM in empirical tests.

Since the last two items incorporate the state of a VM, e.g. whether it is running or currently being manipulated at t_{k+i}, it is required to have knowledge about the duration of manipulations. Additionally, the term $ProcUsage_{res}(k + i, 1, manipProc_m)$ in the fourth line of Equation 4.2 requires knowledge about the resource usage generated by an ongoing manipulation. The duration and resource usage of a single manipulation is estimated in the following manner:
It is assumed that a certain resource usage value can be attributed to any type of manipulation process.[36] This value is retrieved by measuring the resource usage while conducting a single manipulation in an idle environment, i.e. nodes, storage and network are idle except for the single VM and its manipulation. For instance, if a suspend manipulation takes 10 seconds, then the CPU usage is measured and the highest value of the 10 measurement intervals is taken. The difference between this value and the CPU usage generated by the VM prior to suspending it is considered as the CPU usage of the suspend process in an idle environment. Since the situation is commonly more complicated in real life, i.e. multiple concurrent manipulations occur, the MainApp uses resources and multiple VMs are running, there will most probably be competition for resources. This is assumed to prolong the duration of the manipulation and to decrease its resource usage both averaged over the whole duration but also for every single measurement interval. Therefore, the value obtained for a specific manipulation of a single VM in an idle environment is considered to be a maximum value for the resource usage of the respective manipulation process. This value serves as an estimation for the resource usage of a specific manipulation process at t_{k+i}. The values obtained for different manipulation processes and resources in the experimental environment (see Section 6.1) are given in Table 4.2.

manipProc	CPU(%)	NetIn(%)	NetOut(%)
suspProc	2	0	31
migOutProc	30	0	31
stopProc	0	0	0
migInProc	12	30	0
resumeProc	15	25	0
startProc	20	20	0

Table 4.2.: Additional resource usage caused by different manipulation processes. These values are specific for the KIP cluster and a VirtualBox VM with one virtual CPU core, 512 MB RAM and running GNU C Compiler (GCC).

In Table 4.2 it can be seen that multiple values are equal to zero. This either means that the specific manipulation does not use a resource (e.g. NetIn for suspProc) or that starting a manipulation immediately causes a drop in the resource usage (e.g. CPU for stopProc).[37] Please refer to Section A.5 for

[36]Such values are specific for a cluster and VM configuration. Here it is referred to the test environment used in this thesis (see Sections 6.3.1, 6.1).

[37]For instance, in the case of stopping a VM, the VM instantly stops consuming CPU cycles and the overall usage of

a more detailed description on how these values were retrieved and how they can be interpreted.

While the values in Table 4.2 are considered as maximum values for the resource usage of manipulations and can be used for a conservative estimation, the durations found for manipulating a single VM in an idle environment rather represent a minimum value which is probably larger if multiple manipulations occur at the same time. In order to retrieve a reasonable estimation for the duration of single manipulations, multiple experiments with concurrent manipulations have been executed. These are described in Section A.5. For interpreting the resulting estimation, several variables are introduced first:

$$\begin{aligned}
StartsAndResumes &= <total\ number\ of\ ongoing\ and\ proposed\ starts\ and\ resumes> \\
SuspAndStops &= <total\ number\ of\ ongoing\ and\ proposed\ suspends\ and\ stops> \\
ManipsSrcNode &= <number\ of\ ongoing\ and\ proposed\ suspends, stops\ and\ outgoing \\
&\quad migrations\ on\ source\ node> \\
ManipsTrgNode &= <number\ of\ ongoing\ and\ proposed\ resumes, starts\ and\ incoming \\
&\quad migrations\ on\ target\ node>
\end{aligned}$$

The variable $StartsAndResumes$ is the summed number of manipulations with read access to the shared storage that are already ongoing at t or are proposed by the caller of FE. Accordingly the variable $SuspAndStops$ for the manipulations with write access to the shared storage is constructed. The variables $ManipsSrcNode$ and $ManipsTrgNode$ represent the number of ongoing and proposed manipulations on a node that is subject to a VM migration.[38] Based on linear regression using these variables, a very simple model was established that enables the FE to estimate the duration of a single manipulation:

$$duration(suspProc) = \lceil 5.3 + SuspAndStops \rceil\ s \tag{4.3}$$
$$duration(stopProc) = \lceil 0.35 + SuspAndStops/12.5 \rceil\ s \tag{4.4}$$
$$duration(resumeProc) = \lceil 9.2 + StartsAndResumes \rceil\ s \tag{4.5}$$
$$duration(startProc) = \lceil 21 + StartsAndResumes \rceil\ s \tag{4.6}$$
$$duration(migOutProc) = \lceil 0.85 + 3.75 \max(ManipsSrcNode, ManipsTrgNode) \rceil\ s \tag{4.7}$$
$$duration(migInProc) = \lceil 0.85 + 3.75 \max(ManipsSrcNode, ManipsTrgNode) \rceil\ s \tag{4.8}$$

The variable $SuspAndStops$ is used to estimate the duration of a suspend and stop manipulation (see Equations 4.3, 4.4). The variable $StartsAndResumes$ is used to estimate the duration of a resume or start manipulation (see Equations 4.5, 4.6).[39] The proposed estimations for these manipulations assume that the network route and shared storage access are bottlenecks, read and write access do not interfer and that an increased number of concurrent manipulations increases the duration of a single manipulation. For the migration, the estimation only depends on the number of manipulations that are concurrently being executed on the source and target nodes of a migration, which is reflected by variables $ManipsSrcNode$ and $ManipsTrgNode$ (see Equations 4.7, 4.8). The effects of staged manipulations, e.g. a single suspend already runs for 5 seconds at t_k when the FE tries to estimate the duration of another suspend are not taken into account. Every at t_k already running manipulation is considered to have the same impact on the duration like a newly started one.

the CPU drops. Even though the stopping of the VM itself also uses CPU cycles, this amount is relatively small in comparison the CPU usage of a running VM. Therefore, the CPU overhead of a stopProc is considered to be zero.

[38]These variables $ManipsSrcNode$ and $ManipsTrgNode$ are node-specific. So for every node that is the target or the source node of a VM migration these variables need to be calculated separately.

[39]The duration of a start manipulation is determined by t_a = start of manipulation and t_b = start of SecApp inside of VM.

Using both the estimated duration and resource usage of single manipulations, the future usage of a resource at point in time t_{k+i} can be predicted using Equation 4.2. This model and the used estimations are fitted to the experimental setup used in the framework evaluation (see Chapter 6) and sufficiently estimate future resource usage. For a more detailed description and a proposal on how to adapt the estimations to other cluster environments please refer to Appendix, Section A.5.

4.4.2. Global Provisioner

The Global Provisioner (GP) has two main purposes: First, it processes requests of users ("lease owners") to submit, pause, unpause and terminate leases. The basic semantics of these lease primitives have been introduced in Section 4.3.3 and are skipped at this point. Implementational details can be found in Section 5.3. Second, the Global Provisioner (GP) is the instance that makes provisioning decisions, i.e. it decides on when to assign which VMs to what lease and which nodes to allocate these VMs to. Once such a decision is made it is applied to the cluster environment. For this purpose, the finding and applying of a change of the allocation pattern, the GP initially calculates the priority of a lease upon submittance of the lease request. The priority is calculated such that it incorporates all the known properties of a lease, such as its urgency, its size (number of to-be-assigned VMs), the owner's credentials and so forth. In this proof-of-concept approach the formula $Prio(lease) = ownerPrio + 0.5 \times (size)$ was used to calculate the priority of a lease. That basically incorporates a owner's importance $ownerPrio \in [1, 10]$ and the number of VMs that are to be assigned to this particular lease, thereby giving big leases (with many VMs) and leases of important lease owners an advantage over other leases. No updating of priorities during a lease's lifetime is considered, e.g. by incorporating the past queuetime of a lease ("priority aging") into the priority calculation. While this is a desirable feature in order to avoid starvation of leases, it is assumed that for the current proof-of-concept this is unnecessary because the main focus in this thesis lies on the avoidance of strong interference. In future work such improvements should be considered.

Once a priority is calculated, the lease acquires queued state. This literally puts the lease in a queue. This queue consists of leases in queued state, ordered by descending priority, therefore being called priority queue. This priority queue is processed periodically using priority scheduling, which means that leases in the queue are processed in descending priority order. Leases with the same priority are considered in first-come, first-serve order. The whole queue is processed lease by lease. For any lease it is tried to find assignable VMs and nodes which these VMs can be allocated to. The whole queue is processed independent from the fact whether a specific lease is found to be provisionable.[40] The processing of lower-prioritized leases, even if a higher-prioritized lease is not provisionable, is known under the term backfill. Since a MainApp with an unknown resource usage behavior exists, thereby preventing the long term prediction of resources available in the future, any advanced reservation or plan-ahead techniques are pointless. For any processing of the priority queue appropriate lease manipulations are collected. Once the whole queue is processed, the set of found lease manipulations is executed, i.e. applied to the cluster environment. The priority queue is processed periodically. Accordingly, any such periodic processing is called a "provisioning cycle". A provisioning cycle is divided into a decision phase, during which the whole queue is processed and applicable manipulations are collected, and an execution phase during which the found manipulations are applied.

For every lease in the queue a first-fit search heuristic is used during the decision phase. This search traverses the solution space, i.e. the set of possible allocations of VMs to nodes for the current queued lease, and evaluates the appropriateness of the manipulation. The appropriateness of a lease manipula-

[40] A lease is provisionable if a change of the allocation pattern could be found that transfers this lease into provisioned state.

tion is checked by using the previously introduced Feasibility Estimator (see Section 4.4.1), taking into account the overhead induced by the manipulations themselves and assumed resource usage caused by to-be-run VMs. A set of lease manipulations is appropriate if it will, according to the FE, not cause policies to be invalidated after applying the lease manipulations. The first appropriate manipulation of a lease is chosen, independently from the fact whether this allows for additional provisioning of lower-prioritized leases during the very same provisioning cycle. The method applied therefore is a first-fit heuristic with backfill. If during a provisioning cycle a manipulation for a high-priority lease had already been found appropriate, lower-prioritized lease manipulations are checked for appropriateness taking into account these already found manipulations. That means that for lower-prioritized leases a set of manipulations including those of higher-priority leases is checked rather than the single manipulation of the current lease only. A run-to-completion scheme is applied, i.e. no manipulations of running leases are considered by the GP. This means no preemptive actions are taken in order to provision queued leases. This also means a lease manipulation considered by GP only uses the start and resume primitives for manipulating the VMs assigned to a lease.

Once the whole priority queue is processed and a set of manipulations has been found, the decision phase ends and the GP enters the execution phase. During this phase the manipulations are tried to be applied to the cluster environment. For any lease manipulation the execution may succeed or fail. Even more, the execution itself can last some time until the outcome is known. Experiments showed that e.g. resuming a single VM can take 10 or more seconds. As elaborated later in Section 4.4.5.1, there is a global lock on the cluster model during the decision phase and a VM-level lock during the execution phase. These locks practically realize that unless the execution of a manipulation of a specific lease is not finished, the lease remains in queued state, but is not considered by later provisioning cycles. Once the execution of a lease manipulation is finished, the lease enters provisioned state if the execution succeeded or remains in queued state instead. In the latter case, the lease is considered for being provisioned again by future provisioning cycles.[41]
The GP is subordinated to the functional unit described next, the GR. While the GP only takes care of provisioning queued leases and therefore employs a rather simple algorithmic scheme, the GR is responsible for avoiding policy invalidations potentially caused by provisioned leases. The subordination therefore refers to the importance of the functional unit with respect to thesis goals, but also to the fact, that provisioning cycles of the GP can be interrupted during their decision phase by the GR.

4.4.3. Global Reconfigurator

The Global Reconfigurator (GR) is responsible for making sure that no policy invalidation occurs. It is supported in this goal by the GP, which tries to provision only queued leases that are unlikely to cause policy invalidations and by the Resource Usage Adaptor (RUA) which continuously aims to adapt resource usage caused by running VMs on nodes. However, there are cases where local adaption of VM resource usage does not suffice to maintain policy validity.

First, policies can be adapted at runtime. If an administrator decides to change a policy $P_{AllocVM} : (2,10)$ to $P_{AllocVM} : (0,10)$ for a specific node, trying to enforce the number of running VMs to decrease to 0 within 10 seconds, a local resource usage adaption is not feasible. In such a case the GR has to migrate the concerned VMs to other nodes or preempt the lease(s) the VMs are assigned to.

[41] If a lease manipulation fails potential residuals of this failed manipulation are removed first. Then the lease is considered for being provisioned again. See Section 5.4 for more details.

Second, for a given policy P_{CPU} : (80, 10) the RUA continuously tries to adapt the CPU usage of VMs such that the total usage of the CPU resource on a node does not exceed 80% for more than 10 seconds (i.e. avoiding 10 consecutively failing model-checks). If the MainApp now constantly demands a very high share of CPU resources, local adaption of VM-generated CPU usage might not suffice to keep the CPU usage below 80%. Based on the policy definition, the CPU usage of all running VMs then should be set to 0%. This however is considered not feasible, e.g. because it breaks inter-VM communication for a lease. Hence, the GR has to step in and take responsibility for policy validity by migrating VMs or preempting concerned leases.

Multiple questions arise in this context:

1. How and when does the GR recognize a potential policy invalidation which is it has to take care of?
2. How does the GR prevent a policy invalidation? If multiple conflicting solutions exist, what is the criterion for their goodness and how a choice is made?
3. The execution of manipulations has considerable temporal and resource usage relevant overhead. How is a deadline (timelimit) miss avoided? How is transactionality provided and to what extent computational parallelism can be used? How to make sure that manipulations do not cause subsequent invalidations of other policies?

Items 1 and 2 are covered in this subsection detailing the GR: First, the recognition of potential policy invalidations is targeted. Second, the prevention of policy invalidations is explained by sketching the algorithm used for the GR. The questions of item 3 can only be answered after all functional units of the framework have been introduced and their coordination was explained. They are therefore answered at the end of this section (see 4.4.5, 4.4.6.3).

4.4.3.1. Recognition Phase and Triggering

Before starting with the recognition of potential policy invalidations, the terms for describing the basic behavior of the GR are introduced:
In accordance with commonly accepted terminology the computational behavior of the GR is called scheduling. Like the phases of a provisioning cycle of the GP (decision + execution phase) the GR and its scheduling can be described along three phases:

(a) During the recognition phase periodical or externally triggered model-checks are performed. When certain critera are met the scheduling enters the decision phase.

(b) During the decision phase the GR searches an appropriate change of the current allocation pattern which is likely to prevent policy invalidation. Once such a manipulation is found, the execution phase is entered.

(c) During the execution phase the found change of the allocation pattern is applied to the actual cluster environment.

These three phases together are called a "reconfiguration cycle". When referring to any of the introduced cycles (provisioning cycle of the GP and reconfiguration cycle of the GR), without specifying which one is meant, the term "manipulation cycle" is used.

The recognition of potential policy invalidations is situated in the recognition phase. To know about potential policy invalidity, the cluster model is checked periodically, i.e. it is evaluated whether the

statement made by a policy on resource usage (threshold) holds and whether VMs contribute to that usage. Policy invalidity occurs if a number (timelimit) of consecutive model-checks fail. As an example, for the policy $P_{CPU} : (80, 10)$ the CPU usage has to remain below or equal to 80% and policy invalidation occurs if this requirement is violated for 11 consecutive measurements.[42] This policy invalidation the GR has to anticipate and take measures before it occurs.
Therefore, an additional criterion for every policy is introduced. This criterion is called a trigger. When this criterion is evaluated to *True* ("the trigger fires") the decision phase of a reconfiguration cycle is started. In this phase the GR tries to find manipulations that prevent the respective policy from being invalidated.

Every policy possesses two triggers: a global and a local trigger. The local trigger acts as last resort before a policy is being invalidated, i.e. shortly before the timelimit is exceeded and the thereby given deadline is missed. Its firing enforces a hard stopping of any VMs contributing to a potential policy invalidation. Architecture-wise the local trigger is not part of the GR. It is, as the name hints, realized directly on the local node to which the policy is associated. A functional unit, called Local Trigger Unit, takes care of all local triggers defined for a single node. The Local Trigger Unit is part of the architectural component Local Controller. Local triggers are discussed separately in Section 4.4.4.2 where the Local Controller is portrayed.

Global Triggering The checking of a global trigger is a task of the GR. The purpose of a global trigger is the anticipation of a potential policy invalidity. The global trigger reflects the probability that after encountering a number n of consecutive failed model-checks for a policy at points in time $(t_{k+1}, t_{k+2}, ..., t_{k+n})$ the policy will be invalidated at a point in time $t_{k+timelimit+1}$ if the GR does not step in. Without giving a calculation for this probability, it is possible by examining different policies whether this probability is high or low for a specific policy. This now is done for two example policies $P_{CPU} : (80, 10)$ and $P_{AllocVM} : (0, 10)$.
For a policy on the CPU resource, e.g. $P_{CPU} : (80, 10)$, the result of a model-check $(Usage_{CPU} \leq 80)$ may easily oscillate between *True* and *False* for successive, second by second model-checks. This is due to the unknown, potentially changing resource usage of the MainApp, the process scheduling techniques used in OS and the influence of the RUA that continuously adapts the resource share used by VMs. So if a single failed model-check is encountered for such a policy at t_{k+1}, the probability that this policy is invalidated at t_{k+11} if the GR does not step in is relatively low. Now imagine a policy on the resource AllocVM, e.g. $P_{AllocVM} : (0, 10)$ which is associated with a node that currently runs two VMs. This situation occurs if the policy was configured with $P_{AllocVM} : (2, 10)$ before and two VMs were allocated to that node. Now the administrator decides to remove these VMs from this node and reconfigures the policy at runtime from $P_{AllocVM} : (2, 10)$ to $P_{AllocVM} : (0, 10)$. In this case a single failed model-check at t_{k+1} is a very strong indicator that this policy is invalidated at t_{k+11} if the GR does not step in. If not the improbable cases occur that a lease owner incidently just at that time pauses or terminates the concerned leases and VMs, the concerned VMs crash or that the policy is modified again, that policy will be be invalid at t_{k+11}. This is due to the fact that VM allocations do not simply change if not explicitely commanded by GP or GR. Since the GP does not preempt running leases, the policy will be invalidated if the GR does not step in.

Based on these contrasting examples, a basic distinction is made: A policy $P_{res} : (threshold, timelimit)$ is called a static policy if the truth value of a model-check for this policy can only be modified by

[42]Policy invalidation only occurs if a VM contributes to the CPU usage of all 11 measurements.

changing the current allocation pattern. Any other policy is called a dynamic policy. In this thesis static policies are those that use resources AllocVM and UsedCore. For these policies the global trigger fires once a single failed model-check occurs.
For dynamic policies the global trigger is defined slightly different. Since e.g. the usage of the resource CPU can be highly volatile and the RUA is able to influence the resource usage, a single failed model-check is no sufficient indicator for an about-to-happen policy invalidity that has to be taken care of by the GR. A global trigger for dynamic policies as used in this thesis therefore fires when s subsequent model-checks have failed. The variable s depends on the timelimit and lies within $(1, timelimit - 1)$.[43]
The variable s is called "triggerSensitivity". Appropriate values for the triggerSensitivity of a policy were retrieved empirically, which is discussed in Section 6.4.3. For instance, the global trigger of a dynamic policy P_{CPU} : $(80, 10)$ has a triggerSensitivity=4. The empirical retrieval of the triggerSensitivity was subject to a compromise: Low values (for the example: triggerSensitivity < 4) lead to an increased number of potentially unnecessary manipulations. High values (for the example: 4 < triggerSensitivity < 9) limit the solution space of applicable manipulations because fewer time is left until the policy will potentially be invalidated.

A mathematical analysis for triggerSensititivity values, an automated adaption of triggerSensitivity based on historical feasibility or the even the consideration of differing trigger semantics using forecasting techniques like ARIMA[44] is a target for future work.

Policy Violation In the following several phrases are introduced that will help to describe how a GR reacts to a fired global trigger. If a global trigger has fired for a policy P_{res} at time t_{k+i}, then the policy is called "violated". Please take note of the distinction between a policy violation, which indicates that the GR has to act because a global trigger has fired and a policy invalidation, which means that the framework failed to comply with the policy semantics, something that must be avoided at all costs. Once a global trigger fires, i.e. a policy is violated, the GR needs to take measures to prevent policy invalidation.
Because a policy can be associated with e.g. multiple single nodes and a model-check in this case is the evaluation of the threshold statement for each of these nodes, a firing global trigger means that at least for one such node the threshold was exceeded for a number (triggerSensitivity) of subsequent model-checks. This is denoted by saying "the policy is violated for <node>". For instance, the statement "the policy P_{CPU} : $(80, 10)$ is violated for node1 and node4" means that on these two nodes the CPU usage was higher than 80% for at least 4 seconds (4 is the triggerSensitivity of this example policy). By definition a policy can only be violated if a VM uses the resource the policy was defined on. In this case any such resource-using VM is said to violate the policy: "<VM> violates policy <P>". A lease which a policy-violating VM is assigned to is said to be contributing to the policy violation: "<lease> contributes to violation of policy <P>". For any violated policy there may be multiple violating VMs and thereby also multiple leases contributing to this violation.

[43]To be more precise, the value for s depends on multiple factors: *a*) The timelimit of the policy, *b*) the probability distribution for obtaining further $timelimit - s + 1$ consecutive failed model-checks after having already encountered s failed model-checks if the GR does not step in and *c*) the estimated duration of potential changes of the allocation pattern that could avoid a policy invalidation . The mentioned probability distribution in turn depends on the resource type, i.e. how volatile is the usage of that resource, and on the fact whether the RUA has an influence on the usage of that resource.
[44]The AR(I)MA model family is commonly used in engineering, finance and economics to discover trends in time series data.

It is the goal of the GR to prevent policy invalidation once a policy is violated. What exactly does "preventing a policy invalidation" mean?
According to the policy semantics defined in Section 4.3.5, a specific number (timelimit) of consecutive model-checks must not fail. Since upon firing of the global trigger, already a number (triggerSensitivity) of model-checks have failed, only (timelimit-triggerSensitivity) further consecutive model-checks are allowed to fail. After that, a single subsequent failed model-check renders the policy invalid. Hence, the invalidation of a violated policy (whose global trigger fired at t_{k+1}) is successfully prevented if:
$\exists i : 1 < i \leq \text{timelimit-triggerSensitivity} \land eval(S, CM_{t_{k+1+i}}) = True$
So, at least one model-check needs to be true before the timelimit runs out. If the invalidation of a violated policy is successfully prevented, this is called "the policy violation is resolved". Alternatively, also the phrase "the violated policy is revalidated" is used.[45]

Summing it up: The global trigger of a policy is defined by a number of consecutively failed model-checks. This number is called triggerSensitivity. The evaluation of global triggers is done during the recognition phase of the GR's reconfiguration cycle. The firing of a global trigger indicates that the GR needs to change the allocation pattern. Once a single global trigger fires for a policy, this policy is called violated and the GR enters the decision phase. In this phase, the GR aims to find a set of lease and VM manipulations that will revalidate the violated policy. In the following subsection the algorithmic approach used by the GR in the decision phase is described.

4.4.3.2. Search Algorithm

Upon firing of a trigger, i.e. once a policy is violated, the GR enters the decision phase, searching for an appropriate change of the current allocation pattern. Potential manipulations only concern provisioned leases and their assigned running VMs. However, the manipulation is not restricted to violating VMs and violation-contributing leases, also non-violating VMs and leases that do not contribute may be manipulated. A change of the allocation pattern initiated by the GR upon policy violation consists of manipulations $\in \{migrate(VM), suspend(VM), stop(VM), suspend(lease), stop(lease)\}$. Any $suspend(lease)$ and $stop(lease)$ manipulation implicitely requires at least one $suspend(VM)$ or $stop(VM)$. Any violated policy specifies a timing constraint (timelimit). The GR therefore needs to take into account a deadline for the decision and execution phase of any reconfiguration cycle.
The developed algorithm employed by the GR for the decision-phase is a depth-first search with backtracking for traversing the solution space of potential manipulation sets. The choice of target nodes for VM migrations is based on a first-fit approach. The search algorithm utilizes a highly-informative help-function that excludes theoretically possible, yet impractical solutions. The algorithm therefore can be regarded as a guided search heuristic. In the following the algorithmic approach will be explained: First, the driving factors that determine the goodness of a chosen set of manipulations are given. Second, the search heuristic is explained using an example. Third, the algorithm is expressed using a more abstract pseudocode notation.

Goodness of Manipulations Manipulations initiated by the Global Reconfigurator (GR) are those of type $\in \{migrate(VM), suspend(VM), stop(VM), suspend(lease), stop(lease)\}$. The change of an

[45]Not every policy violation requires a change of the allocation pattern in order to resolve the violation. Depending on external disturbance, e.g. an instant decrease in resource usage of the MainApp, a policy revalidation could also occur without GR intervention. Nevertheless, upon firing of the global trigger the GR assumes that a policy invalidation is imminent and will take action.

allocation pattern therefore consists of at least a single VM manipulation. In most cases manipulating a single VM is not sufficient to modify resource usage such that policy violations can be resolved. More commonly, the change of an allocation pattern is a set of multiple manipulations, e.g. consisting of migrations of multiple VMs or even lease suspending or stopping. In order to choose an appropriate set of manipulations, the GR has to assess the goodness of potential manipulations. The driving factors for such an assessment are now discussed separately.

Prevention of Strong Interference It is assumed that enforcing the semantics of appropriately configured[46] policies allows to avoid resource shortage, which is one of the reasons for interference[47]. So, a set of manipulations has to avoid policy invalidations by resolving policy violations. Whether a set of manipulations is appropriate for resolving policy violations depends the type of manipulations and what impact they have towards the usage of the critical resource(s) for which the policy is violated. Any manipulation migrate(VM), suspend(VM) and stop(VM), once it is finished, will stop the respective VM from using resources of the node it was allocated to. This might suffice to resolve a policy violation. However, a manipulation itself takes time and uses resources on its own. So choosing a manipulation means *a*) taking into account whether the manipulation itself will succeed in decreasing the usage of the critical resource within the given timelimit and *b*) whether other policies might be violated by such a manipulation.

The answer to these non-trivial issue is delegated to the secondary unit Feasibility Estimator (FE), which was described above (see Section 4.4.1). This unit estimates the future usage of specific resources and thereby allows for statements on whether a policy violation will be resolved by a set of manipulations or if future policy violations occur. To exemplify the impact of manipulations on resource usage, it can be stated that stopping is the shortest and least resource-demanding manipulation, while suspending a VM comes with a considerable overhead to the NetOut resource and takes considerable time to complete. Migrating a VM carries considerable overhead to NetOut and CPU resources, but under most circumstances is faster than suspending a VM. For more information please refer to Section 4.4.1 which contains a description of FE.
No matter how precise such an estimation can be, it is very important to note that this is an estimation only, therefore valid with a probability of less than 100%. The GR will and must incorporate the FE's feedback, yet relying purely on it is not sufficient to avoid policy invalidations. This is one of the reasons[48] for the local trigger's existence. If the chosen set of manipulations fails to resolve policy violations, this local trigger will eventually prevent policy invalidations by forcefully removing violating VMs. The local trigger functionality is explained later in this section (see 4.4.4.2).

SecApp Result Output Apart from the GR's function to realize policy compliance[49], other goals also drive the scheduling of VMs. Manipulated VMs host SecApps and the effects of manipulations towards these have to be taken into account. Regarding result output there are considerable differences between choosable manipulations. The best choice from the perspective of the virtualized SecApp is the migration of VMs. Such a migration practically leads to a minimal

[46]Configuring policies is a task of the administrator. An appropriate configuration needs to be found by testing multiple setups and choosing one that does not cause strong interference to a MainApp

[47]The other identified reasons, security issues, library conflicts and buggy SecApps, are counteracted by using platform virtualization. If interference can be avoided then also strong interference is avoided.

[48]For instance, the local trigger also is required if the node hosting the GR has crashed and no decision could be made, or if due to a network failure the decision of the GR could not be executed.

[49]To realize policy compliance means avoiding the invalidation of policies.

application downtime and inter-VM communication within a lease is not affected in most cases. Towards a SecApp such a manipulation is nearly transparent.[50] Relative to other manipulations like stopping or suspending a VM, the migration of a VM is considered the best choice with respect to the SecApp result computation. This statement becomes more obvious when looking at the alternatives. Stopping VMs means shutting down the virtual computing system, either by powering it off or cleanly shutting down the OS. In either case this leads to an immediate stopping of the hosted SecApp. Assuming that no advanced techniques like checkpointing or saving of partial results is used by the SecApp, the ongoing computation will stop and has to be repeated once the VM is started again. For long-running jobs, e.g. an ALiEn [10] job computation taking two hours in average, this causes a severe drop in computing efficiency.[51] Suspending a VM is a better choice. This manipulation also stops the SecApp from computing, but saves its current state and therefore precisely avoids the result loss incurred by stopping a VM. So a later resuming of the VM will cause the SecApp to continue processing as if no suspending had occurred. This leads to a better ratio of computed results per time.
When comparing suspending and stopping of a VM to its migration another aspect needs to be considered. Since a lease that multiple VMs are assigned to represents a distributed application, the suspending/stopping of only a single VM of such a lease corresponds to the loss/failure of a physical node within a cluster environment. There are applications (e.g. job schedulers) which can tolerate the failure of a node at runtime, but for sake of generality this scenario is not considered in this thesis. This means when a single VM needs to be stopped or suspended, the whole lease will have to be stopped or suspended.

By looking solely at the effects towards the result computation of virtualized SecApps, there is a definitive preference of VM migrations over lease suspends and of lease suspends over lease stops. This aspect is referred to later on by calling it the impact of a manipulation towards a lease, which is smallest for a migration and biggest for a stop.

Fairness and Prioritization of Leases Apart from these two aspects, effects of manipulations towards policy compliance and SecApp result output, a third factor needs to be considered when choosing an appropriate set of manipulations. A goal of the thesis is to provide the capability to run multiple (competing) SecApps. So, the decision which of multiple provisioned leases to manipulate in case of a policy violation needs to be made. In this thesis all the aspects relevant for assessing a lease's claim to use free resources are boiled down to a single number, the lease's priority. Fairness between leases is therefore realized by taking into account the priority of leases, e.g. by preferring to stop a low-prioritized lease over stopping a higher-prioritized one.

So, the GR makes a choice for a set of manipulations based on these three aspects: the manipulation set's capability to maintain policy compliance, to balance resource usage between multiple leases and its impact towards the result output of SecApps.

Search Heuristic Introduction Rather than taking the common approach of weighting these factors using a utility function and searching for (sub)optimal solutions within the solution space, a

[50] Researchers [119, 27, 75] have evaluated this aspect for different environments and have shown that VM migrations allow to satisfy runtime constraints of virtualized applications under most circumstances. However, Voorsluys et al. [119] report that they observed a considerable downtime of up to 3 seconds for their virtualized applications in a specific cluster environment.

[51] Here the ratio of computed results per time is meant.

search heuristic purely based the priority of leases is employed. The algorithm strives to keep higher-prioritized leases running, e.g. by migrating their VMs at expense of preempting lower-prioritized leases. The decision between suspending and stopping a lease always tends to suspending a lease if the feasibility estimation allows for such a manipulation. According to these basic principles, a search tree is constructed. The first appropriate set of manipulations that, according to FE, will resolve policy violations is chosen. The main reason for utilizing such a heuristical approach is the timing constraint of violated policies (timelimit), which limits the time usable for finding a good solution. A complexity discussion for the portrayed approach is given in Section 4.4.6.1. While portraying the search algorithm, phrases like "the lease is suspended" are used for readibility. Nevertheless, in the decision phase these manipulations are only executed in memory. The manipulations are not applied to the cluster until the decision phase ends and the execution phase is entered.

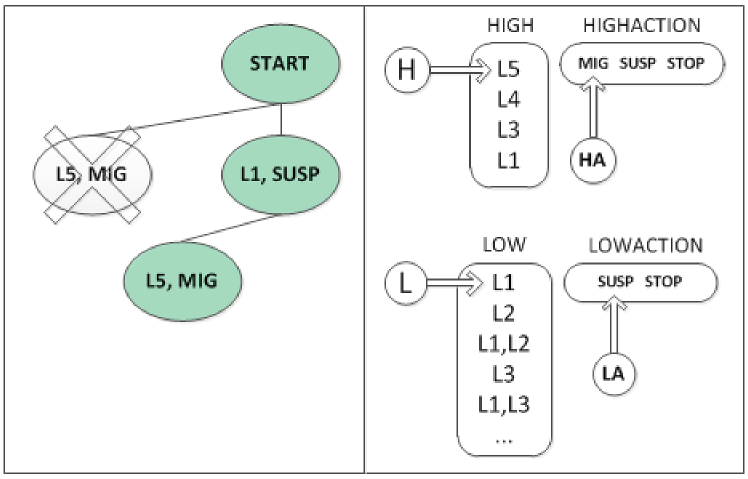


Figure 4.4.: Example search tree (left) and help data structures (right) for the search algorithm.

The algorithm is a depth-first search with backtracking. It operates on a search tree and traverses this tree in a depth-first manner. Backtracking may occur, which means that a current branch is abandoned and a jump is made to a node closer to the root of the tree. Beginning at this backtrack node, alternative branches are explored. A new backtrack node is set by commiting a branch. The tree consists of nodes labeled *(lease, manipulation)*, i.e. *(lease, Susp)*, *(lease, Stop)* and *(lease, Mig)*. While *(lease, Susp)*, *(lease, Stop)* represent the manipulation (suspending, stopping) of all the VMs of the particular lease, *(lease, Mig)* represents the migration of all violating VMs of the lease. An example of the search tree is given on the left side of Figure 4.4. In this example, the migration of violating VMs of lease L5 was tried and failed, e.g. because no targets in the cluster were available. The rejection of this manipulation is reflected by the crossed white node *(L5, Mig)*. Instead, the lease L1 is suspended and afterwards the migration of violating VMs of lease L5 is carried out. Both manipulations have been accepted (committed) by the algorithm, which is indicated in the figure by the green, non-crossed nodes *(L1, Susp)* and *(L5, Mig)*.[52] The generation of nodes in the search tree depends on highly-informative help data structures (ordered lists). An example for these lists is shown on the right side of Figure 4.4: Here the lists *HIGH*, *LOW*, *HIGHACTION* and *LOWACTION* are depicted. These lists

[52]The set of violating VMs of L5 which need to be migrated by the green node *(L5, Mig)* depends on the position of the node *(L5, Mig)* in the search tree. In the very example, the green node *(L5, Mig)* represents the migration of VMs of L5 that still violate a policy after *(L1, Susp)* has been applied. This is not necessarily the same set of VMs that would have been migrated if the white node *(L5, Mig)* had been accepted by the algorithm.

contain leases and applicable manipulations. In this figure also pointers H, L, HA, and LA are shown. These reference the currently active entry in the respective list. These currently active entries are used to generate new nodes in the search tree. The lists constitute of the following (see Section 4.4.3.2 for more details):

HIGH contains all leases contributing to policy violation(s) of the current reconfiguration cycle.
HIGHACTION contains the manipulations applicable for a lease in *HIGH*, i.e. Mig, $Susp$ and $Stop$.
LOW contains lease tuples consisting of provisioned leases.
LOWACTION contains tuples of manipulations that can be applied to lease tuples in *LOW*.

Using entries in these lists, nodes in the search tree are generated. Generated nodes are checked for appropriateness by a function called *checkBranch(branch)*. This function takes as input the currently active branch of the search tree and returns a value $\in \{RESOURCE, TIME, OK\}$. The output of the function determines which tree nodes are generated next. The meaning of the return values roughly corresponds to (see Section 4.4.3.2 for a more precise definition):

$RESOURCE$: There are not enough suitable targets for migrating VMs.
$TIME$: The set of manipulations represented by the branch is likely to invalidate a currently violated policy (e.g. a suspend will not decrease the resource usage below threshold on time).
OK: Neither of the above applies.

Having introduced the basic algorithmic entities: The search tree, the lists and a function assessing a current branch (*checkBranch(branch)*), the algorithm is now introduced by executing an example search first. Afterwards, the algorithm is described in a more abstract fashion by using peudocode, by explaining how the list are generated and how the checking of an active tree branch is done.

Example Search An example search is now carried out: The used setup is shown in Figure 4.5. It consists of nine cluster nodes ($Node1$, $Node2$ ... $Node9$) that are equipped with five provisioned leases. The leases ($L1$, $L2$, $L3$, $L4$, $L5$) and their assigned VMs ($VM1$, $VM2$, ..., $VM22$) are identifiable by their distinct colors. $VM1$, $VM2$, $VM3$ are assigned to lease $L1$, $VM4$, ... ,$VM7$ are assigned to lease $L2$, $VM8$, ... ,$VM12$ are assigned to lease $L3$, $VM13$ ... $VM17$ are assigned to lease $L4$ and $VM18$ ... $VM22$ are assigned to lease $L5$. The numbering of the leases also represents their priority, i.e. $L5$ has the highest priority and $L1$ the lowest one. For the sake of simplicity, each physical node is said to have the same physical resources (CPU, RAM, ...), also the 22 VMs are identical regarding their requirements for CPU cores, RAM and so forth. The cluster is in the state shown in Figure 4.5, when the administrator decides to change the policy configuration. After this change, each physical node is equipped with a policy $P_{AllocVM}$, defining the maximum number of VMs that are allowed to run on a single node. The policy threshold setting for a specific node is exemplified by white boxes on the right side of each node.[53] The number of non-crossed white boxes equals the number of VMs allowed to run on a node, e.g. $P_{AllocVM} : (0, 10)$ for $Node6$ and $P_{AllocVM} : (3, 10)$ for $Node4$. According to this explanation, the policy is violated for $Node1$, $Node3$ and $Node6$. $Node8$ has a free slot for an additional VM and the rest of the nodes are optimally filled with running VMs. With a single model-check the policy violation for $Node1$, $Node3$, and $Node6$ is recognized and the global triggers fire. Upon triggering, the decision phase starts by calculating the data structures *LOW* and *HIGH*.

[53]The timelimit of each policy $P_{AllocVM}$ is set to 10 seconds.

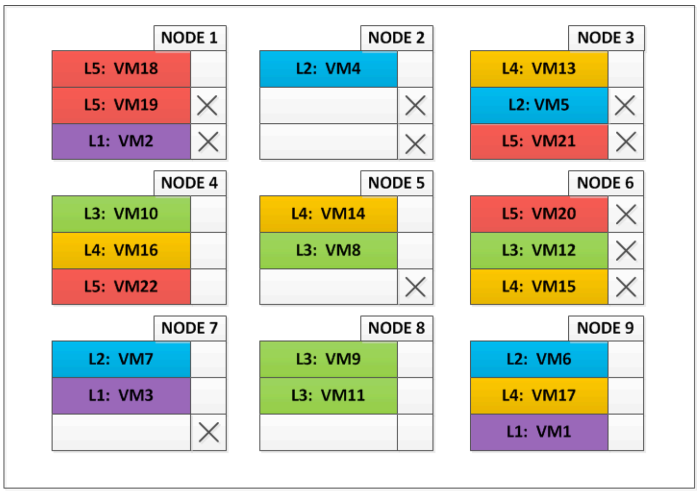


Figure 4.5.: Initial cluster state in example search. *Node*1, *Node*3 and *Node*6 have violated policies.

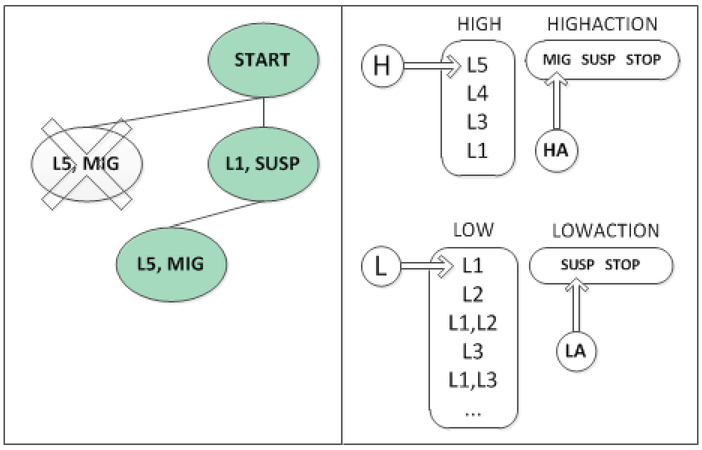


Figure 4.6.: Search tree (left) and help data structures (right) after lease $L1$ was suspended and VMs of lease $L5$ were migrated.

$HIGH$ is an ordered list of leases contributing to a violation: $HIGH = (L5, L4, L3, L1)$. LOW consists of preemptable lease tuples: $LOW = \big(L1, L2, (L1, L2), L3, (L1, L3), (L1, L2), ...\big)$.[54] The algorithm tries to keep lease in $HIGH$ running at the expense of leases in LOW and proceeds as shown on the left in Figure 4.6. First a migration of violating VMs of $L5$ is tried in order to keep $L5$ running. There are four violating VMs for lease $L5$: $VM18$, $VM19$, $VM20$, $VM21$. However, only three VMs of this lease need to be migrated to make $L5$ not contribute to a violation anymore: The migration of either $VM18$ or $VM19$ on $Node1$ suffices, it is not needed to migrate both of them. The migration of three VMs of lease $L5$ does not work out because not enough migration target slots are available in the cluster. This is indicated by $checkBranch(branch) = RESOURCE$. Therefore, this tree branch is skipped and instead the lowest-prioritized lease $L1$ from list LOW is suspended and violating VMs of $L5$ are

[54]The list LOW is an ordered power set of all provisioned leases. It has 2^5 entries and therefore is not fully detailed. Please see Section 4.4.3.2 for more information.

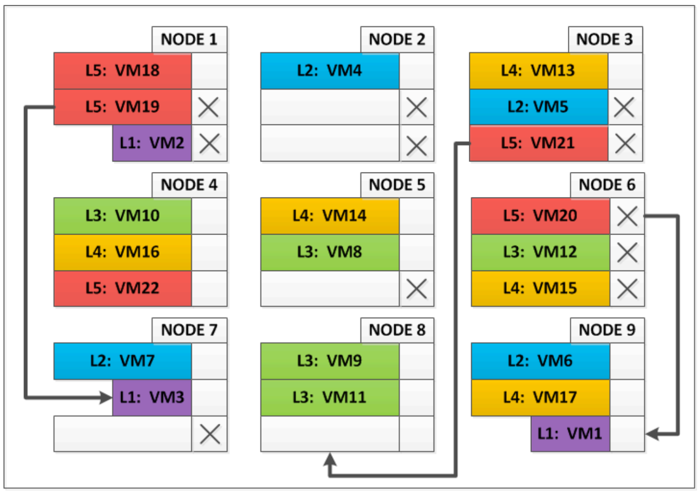


Figure 4.7.: Cluster state with suspend of lease $L1$ and migration of VMs belonging to lease $L5$.

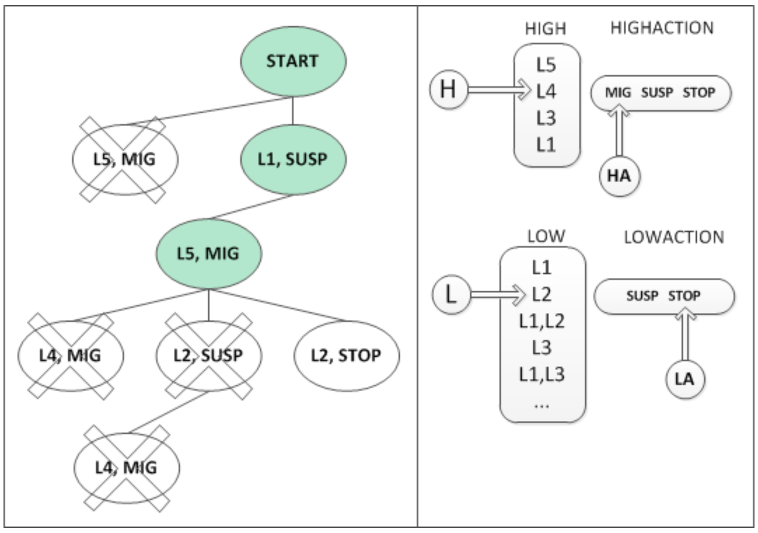


Figure 4.8.: Search tree (left) and help data structures (right) while evaluating the stop of lease $L2$.

migrated. The corresponding *LOW* and *HIGH* lists with their currently active entries are shown on the right in Figure 4.6. The effects of the suspending and migrating can be seen in Figure 4.7. Here the suspending is indicated by shortended VM bars and the migrations are indicated by arrows: All VMs of $L1$ are suspended ($VM2$ on $Node1$, $VM3$ on $Node7$ and $VM1$ on $Node9$). The migrated VMs of lease $L5$ are: $VM19$ on $Node1$ (to $Node7$), $VM21$ on $Node3$ (to $Node8$) and $VM20$ on $Node6$ (to $Node9$). The choice, which of the violating VMs of lease $L5$ on $Node1$ to migrate, is made using a first-fit decreasing approach, in this example this leads to the random choice of $VM19$.[55] Once these manipulations have been committed ($checkBranch(branch) = OK$), the algorithm proceeds with the next

[55]The violating VMs, here $VM18$ and $VM19$ are chosen in order of decreasing number of virtual CPU cores assigned to a VM. If these are equal, a random choice is made. In this example, the VM is chosen randomly because every VM has the same properties.

lease in *HIGH*: $L4$. The resulting search tree is given on the left side of Figure 4.8. A following migration of $L4$ does not succeed because not enough resources for the two to-be-migrated VMs ($VM13$ on $Node3$, $VM15$ on $Node6$) are available ($checkBranch(branch) = RESOURCE$). Therefore, $L2$ in *LOW* is tried to be suspended first before migrating violating VMs of $L4$. For this attempt, the $checkBranch(branch)$ function returns $TIME$. The interpetation is the following: The suspending of all VMs of $L2$ ($VM4$ on $Node2$, $VM5$ on $Node3$, $VM7$ on $Node7$, $VM6$ on $Node9$) and a subsequent migration of violating VMs of lease $L4$ would be feasible from a resource perspective; enough migration target slots are freed by suspending $L2$.[56] However, at least one of the involved manipulations

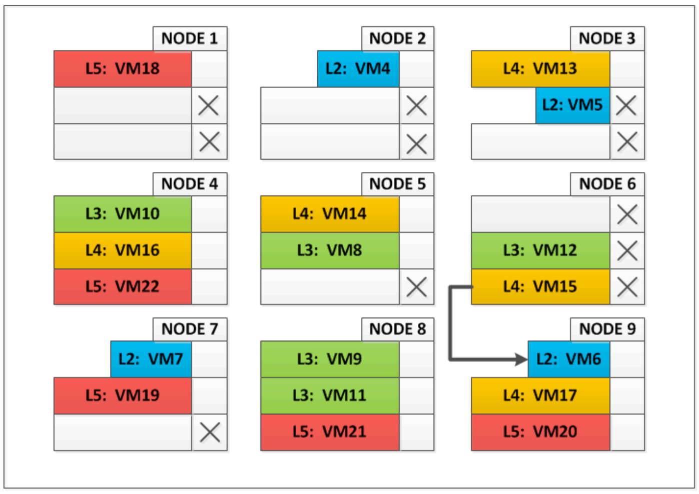


Figure 4.9.: Cluster state with stop of lease $L2$ and migration of $VM15$ belonging to lease $L4$.

will not finish within timelimit of 10 seconds. This could, e.g., be the suspending of $VM5$ on $Node3$. There are, according to the committed branch, already planned suspends (for lease $L1$). Further suspends might take too long, thus exceeding the timelimit. Therefore, the currently active sub-branch (*L2, Susp*), *L4, Mig*)) is rejected. Instead, the stopping of lease $L2$ is tested. The corresponding *LOW* and *HIGH* lists are shown on the right in Figure 4.8, the search tree is given on the left of this figure. The algorithm proceeds as shown in Figure 4.10: VMs of $L2$ ($VM4$ on $Node2$, $VM5$ on $Node3$, $VM7$ on $Node7$, $VM6$ on $Node9$) are stopped and violating VMs of lease $L4$ are migrated ($VM15$ on $Node6$ (to $Node9$). A check yields $checkBranch(branch) = OK$, so the branch is committed. This step is shown in Figure 4.9, where the stopping of VMs of $L2$ and a following migration of $VM15$ from $Node6$ to $Node9$ are indicated. At this point, only a single VM is left that violates a policy: $VM12$ on $Node6$. With H pointing to $L3$ in *HIGH*, the migration of this VM is tried. The corresponding tree is shown in on the left in Figure 4.10. At this point, the migration of $VM12$ would be possible resource-wise (e.g. to $Node2$, which has an open resource slot). This does however fail because $checkBranch(branch) = TIME$. This is reasonable, because on $Node6$ there are already two planned migrations (of VMs assigned to leases $L4$ and $L5$). Any further migration might take too long. Consequently, the next action on a lease in *LOW* is tried. Interestingly the lease in *LOW* is also $L3$. This lease is therefore tried to be suspended, which again fails because of $checkBranch(branch) = TIME$ which is due to several, already planned suspends. Therefore, $L3$ is chosen to be stopped. The corresponding lists are given in Figure 4.10 and the resulting cluster is shown in Figure 4.11. Now H

[56]Please take note that a suspending of $VM5$ on $Node3$ would have made the migration of $VM13$ of $L4$ obsolete. Only $VM15$ of $L4$ on $Node6$ still would require to be migrated.

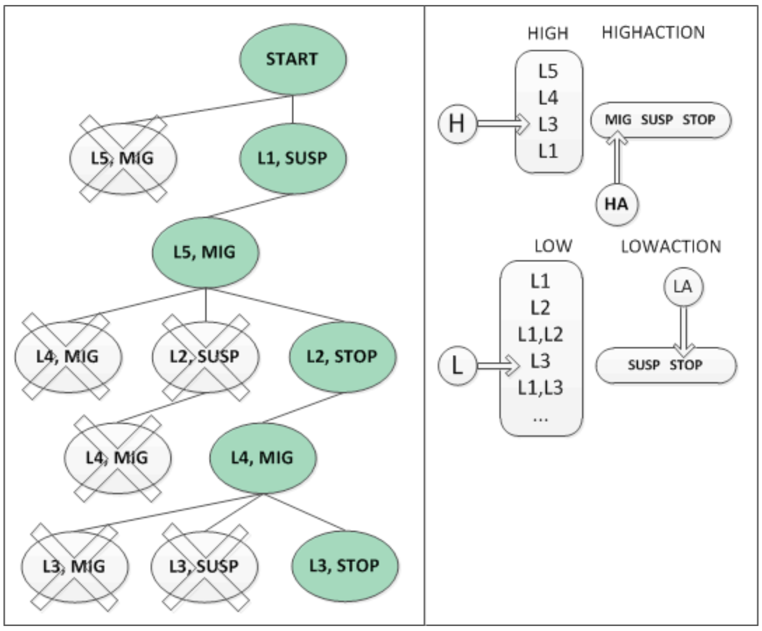


Figure 4.10.: Search tree (left) and help data structures (right) at the end of GR's decision phase. The branch in green is the found set of manipulations.

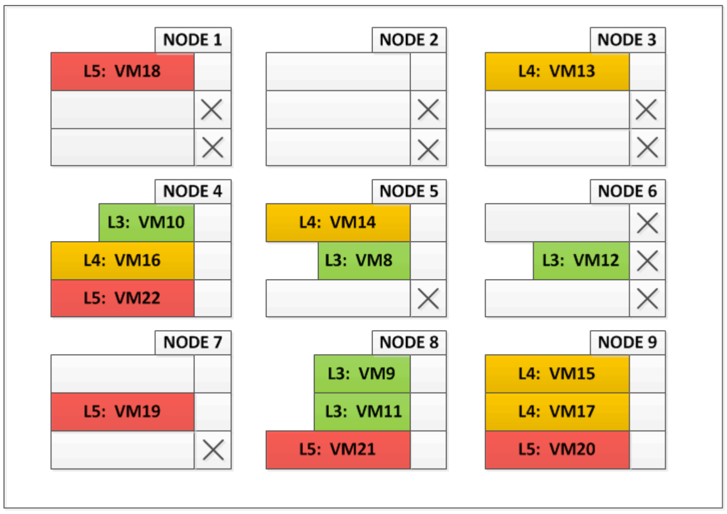


Figure 4.11.: Cluster state with the final stop of lease $L3$.

points to $L1$ as the last lease in *HIGH* which initially contributed to a policy violation. Since $L1$ had been suspended at an earlier stage of processing, the pointer H is incremented and therefore points to *NULL* now. This finishes the search algorithm and returns the committed branch, which is indicated in Figure 4.10 by the green-ovaled branch: $\big((L1, Susp), (L5, Mig), (L2, Stop), (L4, Mig), (L3, Stop)\big)$. Now a set of manipulations has been found, which most probably will revalidate the violated policies. The next step is the execution of the found manipulation pattern. The GR therefore enters the execution phase. This phase is explained later in Section 4.4.5.1.

Search Heuristic Abstraction Now that the algorithm has been introduced by example, it is described in a more abstract way using a pseudocode notation: As was shown in the example search, the algorithm operates on a search tree. Tree nodes are generated according to the currently selected entries in the ordered lists *HIGH*, *HIGHACTION*, *LOW* and *LOWACTION*. Branches of the tree are checked using a function *checkBranch*(*branch*). A sketch of the search is given in Algorithm 4.4.1:

```
Algorithm 4.4.1: ▷ (Search for appropriate lease and VM manipulations)

$H, HA, L, LA \leftarrow 0$                                                  (1)
$BACK \leftarrow START$
$CURRENT \leftarrow$ GENERATENODE($H, HA$)                                   (2)
while $H \neq NULL$                                                          (3)
  do
    $branchState \leftarrow$ CHECKBRANCH($START, CURRENT$)                    (4)
    if $branchState = ok$                                                    (5)
      then
        while $H$ does not contribute to any violation
          do
            $BACK \leftarrow CURRENT$                                        (6)
            $HA \leftarrow 0$
            $H \leftarrow H + 1$
            while $L$ has already been manipulated
              do
                $L \leftarrow L + 1$
                $LA \leftarrow 0$
        if $H \neq NULL$
          then $CURRENT \leftarrow$ GENERATENODE($H, HA$)                    (7)
    else if $branchState = resource$                                         (8)
      then
        $CURRENT \leftarrow BACK$                                            (9)
        $LA \leftarrow 0$
        $CURRENT \leftarrow$ GENERATENODE($L, LA$)                           (10)
        $L \leftarrow L + 1$
    else if $branchState = time$                                             (11)
      then
        $CURRENT \leftarrow BACK$                                            (12)
        $LA \leftarrow LA + 1$
        if $LA = NULL$
          then
            $L \leftarrow L + 1$
            $LA \leftarrow 0$
        $CURRENT \leftarrow$ GENERATENODE($L, LA$)                           (13)
return ($START, CURRENT$)                                                    (14)
```

The algorithm starts at a node (*START*) representing the root of the tree and iteratively generates child nodes in the search tree. For this generation it uses a function *generateNode(lease, manipulation)* (see breakpoints 2,7,10,13), which is parameterized with entries of the mentioned lists: The parameter *lease* is taken from either list *HIGH* or *LOW*, the parameter *manipulation* is taken from either list *HIGHACTION* or *LOWACTION*. For example, at breakpoint 2 a node is generated with the active entry in *HIGH* (a lease pointed at by *H*) and the active entry in *HIGHACTION* (a manipulation pointed at by *HA*).

The syntactical construct $H \leftarrow H + 1$ is used to denote that the next entry in the respective list (here: *HIGH*) is about to be processed. Accordingly, $H \leftarrow 0$ means that the first element in *HIGH* is about to be processed. $H = NULL$ occurs if H pointed to the last element in *HIGH* and a subsequent $H \leftarrow H + 1$ was issued. At start (see breakpoint 1), the pointers point at the first entry in each list. *BACK* and *CURRENT* are pointers to nodes in the search tree. While *CURRENT* references the latest generated node, *BACK* is a pointer to a node in the tree which can be jumped to using backtracking. Using such a jump abandons the branch which was generated and explored below the node referenced by *BACK* and resets *CURRENT* to *BACK*. This basically means that the current branch explored by the depth-first search provides no appropriate set of manipulations, and therefore some alternative branches need to be explored.

A short remark needs to be made at this point: While the code snipplet shows the main algorithmic approach taken by the GR, it still is an abstraction of the actually employed algorithm. For brevity and ease of understanding the data structures representing the lists and the search tree themselves are omitted. The layout of the tree derives from the usage of pointers *START*, *BACK* and *CURRENT* and the list entries are sufficiently referenced using pointers *H*, *HA*, *L* and *LA*. The currently active tree branch is therefore represented by (*START*, *CURRENT*) and corresponds to the shortest path in the search tree between the nodes referenced by *START* and *CURRENT*.

The basic procedure of the algorithm is the following: Breakpoint 3 starts a while loop that iterates over all entries in *HIGH*. At breakpoint 4 the current tree branch (referenced by (*START*, *CURRENT*)) is checked for appropriateness by the *checkBranch*(*branch*) function. Depending on the return value ($RESOURCE$, $TIME$, OK), one of the continuations at breakpoints 5,8 or 11 is used.

An interpretation of the algorithm is now given: The algorithm iterates over the high-prioritized leases (entries in *HIGH*) and tries to migrate their violating VMs. To determine the targets of such migrations, a first-fit approach is used, i.e. the first appropriate target node in the cluster is chosen. Appropriateness is determined using the aforementioned function *checkBranch(branch)*, which in turn makes use of the Feasibility Estimator (FE). The return value provided by this function is used to decide whether the migrations are appropriate or if free slots for VMs have to be provided first. In the latter case (breakpoint 8, 11), the low-prioritized lease (tuples) in *LOW* are iterated over to find appropriate $Susp$ or $Stop$ manipulations which in turn could provide these slots, or even render the successive migration of VMs assigned to the high-priority lease (in *HIGH*) obsolete. The difference between breakpoints 8, 11 (i.e. the difference between *checkBranch*(*branch*) return values $RESOURCE$ and $TIME$) is that for $RESOURCE$ no further ($susp|stop$) variations for the current tuple in *LOW* is tried. The outcome of these manipulations does not differ with respect to the slots providable to a lease in $HIGH$. Since $RESOURCE$ indicates that more suitable slots are needed, these variations are not tested.

Backtracking (see breakpoints 9, 12) occurs if a sub-branch (*BACK*, *CURRENT*) was found to be not appropriate (*checkBranch(branch)* returns $RESOURCE$ or $TIME$). Then pointers *L* and/or *LA* are incremented and alternative nodes are generated to suspend or stop low-priority leases. For a specific lease in *HIGH* this continues until either a branch is found appropriate (*checkBranch*(*branch*) $= OK$, breakpoint 5) or that lease itself is reached in the *LOW* list.[57] In the latter case, the high-priority lease is tried to be suspended or even stopped. This eventually leads to *checkBranch*(*branch*) $= OK$ (breakpoint 5) as well. A branch is committed (see breakpoint 6), once a set of manipulations was found that makes this lease not contribute to a violation anymore. At best, previously committed manipulations already resolve the policy violations and the lease does not contribute to a violation anymore. At worst, the lease needs to be stopped. Then the pointer *H* is incremented and the next lease in $HIGH$

[57]Every lease in *HIGH* also appears in *LOW*.

is active. Once all leases in $HIGH$ are iterated over, the algorithm finishes and returns the committed search tree represented by $(START, CURRENT)$.

The set of manipulations generated by this algorithm is mainly driven by two aspects: First, there is the *checkBranch(branch)* function deciding on the appropriateness of manipulations. If a manipulation is found to be not appropriate for resolving policy violations, the function yields a return value that gives additional information on why this is the case ($RESOURCE$ or $TIME$). Based on this additional feedback the search tree is constructed. Second, there are the mentioned lists *HIGH*, *HIGHACTION*, *LOW* and *LOWACTION*. These data structures, i.e. their entries and their ordering are crucial for the algorithm's behavior. The lists and the $checkBranch(branch)$ function are now explained.

The Lists Four lists are used by the algorithm to create new nodes in the search tree:

HIGH This list contains all leases that contribute to policy violations triggered for the current reconfiguration cycle. All policy violations are resolved if all leases of this list are manipulated. However, not every lease in *HIGH* needs to be manipulated: Manipulations of other leases in *LOW* or *HIGH* can render the manipulation of a lease in *HIGH* obsolete. The order within this list is defined by: $Prio(L1) > Prio(L2) \rightarrow Rank(L1) < Rank(L2)$, i.e. the leases in *HIGH* are descendingly ordered by priority. Leases with the same priority are ordered by the first-come, first serve principle. This list is generated every time the GR enters the decision phase.

HIGHACTION This list contains the manipulations applicable to leases in *HIGH*. The list is ordered by ascending impact of the manipulation towards the SecApp, i.e. $(Mig, Susp, Stop)$. This data structure is static and does not change.

LOW All leases in provisioned state are used to establish this datastructure. The algorithm strives to keep leases in $HIGH$ running. It does so even at the expense of suspending or stopping leases in *LOW* if this resolves the policy violations which a lease in $HIGH$ was contributing to or if this allows for migrating the violating VMs of a lease in $HIGH$. This even may involve suspending multiple low-prioritized leases. As an example, leases $L6, L5, L4$ with $Prio(L6) > Prio(L5) > Prio(L4)$ could be suspended in order to keep lease $L9$ running. Suspending lease $L4$ neither resolves all policy violations which lease $L9$ was contributing to nor does it provide enough free slots in order to migrate the violating VMs of lease $L9$. The same applies to suspending lease $L5$. At this point two options exist: Suspending lease $L6$ or suspending both $L4$ and $L5$. According to the algorithm's basic principle (keep high-priority leases running), suspending of $L4$ and $L5$ is tried next. More formally this approach is described such: The list *LOW* is created as the power set of all provisioned leases. Based on example leases $L6, L5, L4$, the resulting datastructure is $LOW = \big(L4, L5, L6, (L4, L5), (L4, L6), (L5, L6), (L4, L5, L6)\big)$. The entries in *LOW* represent sets of leases that may be manipulated. This list is now quick-sort ordered using the order-establishing comparison procedure given in Algorithm 4.4.2. This algorithm recursively determines which of two lease tuples is lower-prioritized, i.e. is ranked lower in the *LOW* list. The resulting list is $LOW = \big(L4, L5, (L4, L5), L6, (L4, L6), (L5, L6), (L4, L5, L6)\big)$, which precisely follows the principle of keeping higher-prioritized leases alive at expense of lower-prioritized ones.

```
Algorithm 4.4.2: ▹ (Establishes an order relation (>) for two lease tuples a, b)

  comment: $|a| = 0 \iff a$ has no elements
  comment: $max(a)$ is the lease in $a$ with the highest priority
  comment: $a - max(a) \equiv$ removes the lease in $a$ with the highest priority

  procedure Compare($a, b$)
   if $|a| = 0$
     then return ($FALSE$)
   if $|b| = 0$
     then return ($TRUE$)
   if $max(a) \neq max(b)$
     then $NEWa \leftarrow (a - max(a))$
          $NEWb \leftarrow (b - max(b))$
          return (Compare($NEWa, NEWb$))
     else return ($TRUE$)
```

Other ordering principles are possible. However, it is assumed that priority represents all the relevant aspects of a lease (e.g. number of VMs, urgency, owner credentials etc.) and that no aggregate metrics over lease tuples are needed. In such a case, the presented technique is sufficient for a proof-of-concept approach. This list *LOW* is generated every time the GR enters the decision phase. For a large number of provisioned leases, the complexity of establishing and sorting the list is not acceptable.[58] Therefore, *LOW* is not calculated from scratch every time the GR enters the decision phase. When the VM-Scheduling Framework is started, a preliminary *LOW* list is created for the (reconfigurable) number of 10 dummy leases. When the GR enters the decision phase, the actually provisioned leases are mapped to these dummy leases, making the computation of *LOW* for the lowest-prioritized n leases unnecessary.[59]

LOWACTION The datastructure *LOWACTION* contains the manipulations applicable for lease tuples in *LOW*. These manipulatons are only $Susp$ and $Stop$. Migrating VMs of a lease in *LOW* is considered not relevant.[60] Since *LOW* contains tuples of leases, *LOWACTION* contains tuples of manipulations. So for, e.g., the tuple $(L4, L5)$ the following manipulations are possible:

$$\big(Susp(L4), Susp(L5)\big), \big(Stop(L4), Susp(L5)\big), \big(Susp(L4), Stop(L5)\big), \big(Stop(L4), Stop(L5)\big)$$

LOWACTION is created as follows: For a specific entry (tuple) in *LOW* containing n leases, the corresponding *LOWACTION* consists of tuples $(\ Susp \mid Stop\ )^n$, so there are 2^n entries in *LOWACTION*. To decide which manipulations are tried first, an order relation upon *LOWACTION* is defined. Suspending a lease is considered to be preferable over stopping it and if stopping is needed, then lower-prioritized leases are stopped rather than higher-prioritized ones. The order is therefore defined as:

[58]The power set of a set of cardinality n has cardinality 2^n.

[59]When the number of leases known to the framework extends n, an extension to the dummy list is easily computed in background upon submission of an additional lease.

[60]There may be cases where the migration of a VM assigned to a lower-prioritized lease is benefitial for freeing resource slots for VMs of higher-prioritized leases. Such an optimization, which merely is a shuffling of VM allocations at runtime, is assumed to be rarely appropriate for resolving policy violations.

1. The leases of a lease tuple in *LOW* are ordered by ascending priority, e.g. a lease tuple in *LOW* is $(L4, L5)$, not $(L5, L4)$).
2. Any manipulation tuple $entry_i$ in *LOWACTION* can be coded as a sequence of binary numbers with $Susp \equiv 1$ and $Stop \equiv 0$. Thus, $(Susp, Stop)$ is coded as "10". The order relation imposed on *LOWACTION* is:
   $Rank(entry_1) > Rank(entry_2) \longleftrightarrow binaryCode(entry_1) < binaryCode(entry_2)$

For any to be processed tuple in *LOW*, the corresponding *LOWACTION* list is computed at runtime.

Entries and order of the lists *HIGH*, *HIGHACTION*, *LOW* and *LOWACTION* affect the set of manipulations produced by the algorithm. The other important factor is the *checkBranch(branch)* function, that decides on the appropriateness of a given set of manipulations and also returns additional information used for further processing.

The *checkBranch(branch)* Function This function assesses a given set of manipulations for its effect on resource usage in cluster. It associates this usage with configured policies and returns a value that helps the caller to decide on how to proceed in selecting lease and VM manipulations. In order to find out about the resource usage that comes with a given set of manipulations, the function makes use of the Feasibility Estimator (FE). Please refer to Section 4.4.1 for more information on the FE. The search algorithm processes the lists as given in Algorithm 4.4.1. It generates nodes in a search tree representing lease manipulations. At times (see breakpoint 4), the *checkBranch(branch)* function is called. The set of manipulations represented by the branch (*START*, *CURRENT*) is handed over as parameter. The function $checkBranch(branch)$ assumes that these manipulations are about to be executed instantly and in parallel. For assessing their effects on resource usage, the explicit constraints on resource usage (policies) are taken into account. For every involved node (e.g. a node that a to-be-suspended VM is allocated to, target and source node of a migration), its associated policies are determined. This yields a set of policies which is examined further:
For every policy which is violated in the current reconfiguration cycle, the FE is called to find out whether the set of manipulations could invalidate policies. The FE is given two parameters: *a*) The resource on which that policy operates and *b*) the date of the first model-check after the timelimit of the policy runs out.[61] The set of manipulations is not appropriate if the value returned by FE, i.e. the resource usage at this future point in time, exceeds the threshold of the violated policy.
For every policy that is not violated, the Feasibility Estimator (FE) is called to find out whether the set of manipulations could violate this not-yet-violated policy. Therefore, the FE is given two parameters: *a*) The resource on which that policy operates and *b*) the point in time at which the global trigger for this policy could earliest fire if the manipulations were instantly applied.[62] The set of manipulations is not appropriate if the value returned by FE, i.e. the resource usage at this future point in time, exceeds the threshold of the policy.

[61]More precisely: The global trigger of the policy has fired at t_k. That is, at t_k already $triggerSensitivity$ failed model-checks have occured. It needs to be checked whether at least one of the remaining model-checks at $[k+1, k+1-triggerSensitivity+timelimit]$ will yield *True* if the set of manipulations is applied. Otherwise the policy is invalidated, thus the proposed set of manipulations is inappropriate. In the prototypical implementation only the usage of the resource at $t_{k+1-triggerSensitivity+timelimit}$ (i.e. the first measurement after timelimit elapses) is taken into account: The set of manipulations is not appropriate if this value exceeds the threshold of the policy.

[62]This is at latest $t_{now+triggerSensitivity}$, but can also be an earlier point in time if several subsequent model-checks have already failed, but not yet led to a policy violation.

Depending on the outcome of these checks, the *checkBranch(branch)* function returns a value to the caller, i.e. to the search algorithm. The basic procedure for determining the return value is given in Algorithm 4.4.3.

Algorithm 4.4.3: ▹ (*Evaluates a tree branch*)

```
procedure checkBranch($START, CURRENT$)

  if At least one not yet violated policy will be violated
    then return ($RESOURCE$)
  if At least one currently violated policy will be invalidated
    then return ($TIME$)
  return ($OK$)
```

If the *checkBranch(branch)* function returns OK, then the current set of manipulations neither violates new policies nor invalidates already violated policies. In that case, the branch is committed (breakpoint 7) because the current high-priority lease does not contribute to a violation anymore. The return values $RESOURCE$ and $TIME$ indicate that the current set of manipulations is not sufficient to keep the high-priority lease running while at the same time avoiding new policy violations or invalidations of already violated policies. One may wonder why a distinction between return values $RESOURCE$ and $TIME$ is made. Essentially, this distinction is not needed to find appropriate manipulations. However, by giving the caller (the search algorithm) a hint on why a set of manipulations is not appropriate, the search algorithm can adapt the generation of new tree nodes according to this hint. This actually is done at breakpoints 8 and 11 in Algorithm 4.4.1, where the search proceeds depending on the return value of the *checkBranch(branch)* function. In both cases backtracking occurs (breakpoints 9, 12), i.e. the current non-commited branch is abandoned and processing continues at the backtracked node (*BACK*). An informal interpretation of the return values of Algorithm 4.4.3 is now given:
The return value of *RESOURCE* can be interpreted such that no sufficiently dimensioned migration targets for all violating VMs of the current lease in *HIGH* exist. In this case, the algorithm will choose leases in *LOW* to be preempted to provide such migration targets. As long as $RESOURCE$ is returned, there is no need to vary the method of preemption (suspend/stop), since both will lead to the same outcome, only differing in short-term resource usage and duration. Instead, the *LOW* list is traversed to find low-priority leases that either provide migration targets or, upon preemption, will resolve the policy violation without having to migrate VMs of the high-priority lease. In contrast, the return value of *TIME* means that the current set of manipulations will likely lead to policy invalidation, i.e. the selected manipulations will not succeed in decreasing the resource usage on time. In this case, before trying to preempt other leases in *LOW*, it is tried to choose shorter-lasting manipulations, e.g. to stop a lease instead of suspending it. So, the distinction between $RESOURCE$ and $TIME$ is an heuristical optimization that keeps the algorithm from traversing a sub-set of the solution space that is unlikely to provide appropriate results.
A remark concerning the computational effort for computing this function needs to be made at this point. The description of its functionality may suggest a high time complexity. However, many resource usage estimations do not require the full involvement of the FE. For instance, in order to find out whether the migration of a VM to a target node will violate a policy on the resource $AllocVM$, no estimation on the overhead and duration of such a manipulation needs to be made. In most cases, a simple addition on the number of available CPU cores is sufficient to know whether a node is a target

for a migration. Also, in any non-trivial case there is more than one configured policy. Since some policy types are more easy to evaluate (e.g. those for $AllocVM$) or are more likely to exhibit volatile model-check results (CPU), it is reasonable to prioritize the checking of some policies over others such that a return value is obtainable without having to check every resource for which a policy is defined. Furthermore, the checking task is parallelized on a per-resource level. These implementation-level optimizations are not detailed in this thesis, but eventually lead to a worst-case time complexity for this function of $O(|Policies|)$, i.e. the time-complexity scales linearly with the number of policies. This function *checkBranch(branch)* therefore is not crucial for determining the overall complexity for the proposed search algorithm. A discussion of this complexity is given later in Section 4.4.6.1.

As a summary for the *checkBranch(branch)* function it can be stated: This function is used by the search algorithm to assess the effect of a set of manipulations on the compliance with the policy semantics. Sets of manipulations that neither violate new policies nor invalidate already violated policies are considered appropriate. The assessment bases on future resource usage estimations made by the FE. A heuristical distinction between return values $RESOURCE$ and $TIME$ is used to shrink the solution space to be traversed.

Summary GR In this section the functional unit Global Reconfigurator (GR) was introduced. The GR is responsible for making sure that no policy invalidation occurs. It is supported in this goal by the Global Provisioner (GP), which tries to provision only queued leases that are unlikely to cause policy invalidations and by the Resource Usage Adaptor (RUA) which continuously aims to adapt resource usage caused by VMs on nodes. Two questions were posed at the beginning of this subsection:

1. How and when does the GR recognize a potential policy invalidation which is it has to take care of?
2. How does the GR prevent a policy invalidation? If multiple conflicting solutions exist, what is the criterion for their goodness and how a choice is made?

These questions were answered: The GR operates in cycles, the so-called reconfiguration cycles. A reconfiguration cycle comprises three parts: The recognition phase, the decision phase and the execution phase. The first question is subject to the recognition phase. For every configured policy there is a global trigger. This trigger is a criterion on model-checks. Once a trigger fires, i.e. the criterion is evaluated to $True$, a potential policy invalidation is indicated. This means, that a policy is likely to be invalidated if the GR does not take measures. The global trigger was defined as a number of subsequently failed model-checks. This number is called triggerSensitivity. The triggerSensitivity of a global trigger for a specific policy depends on the type of resource which the policy is defined on. Reasonable values for the triggerSensitivity of policies were retrieved empirically. Once a global trigger fires for a policy, the policy is called violated.
Upon policy violation, the GR enters the decision phase. In the decision phase the GR tries to find VM and lease manipulations that are likely to resolve a policy violation, i.e. to prevent the policy from being invalidated. Three aspects steer the choice of VM and lease manipulations: *a*) The assumed capability of a set of manipulations to decrease the resource consumption such that policy invalidation is prevented, *b*) the fairness between leases, i.e. every lease receives a share of computing resources according to its priority and *c*) the effect of a set of manipulations on the overall result output of SecApps represented by leases. These three choice criteria culminate in the basic approach that the search algorithm uses to find and assess manipulations: A search heuristic purely based the priority of leases is employed. The algorithm strives to keep higher-prioritized leases running, e.g. by migrating

their VMs, even at expense of preempting lower-prioritized leases. The choice of targets for migrating a VM is first-fit. First-fit decreasing is used if multiple VMs of a lease have to be migrated from the same physical cluster node. The decision between suspending and stopping a lease always tends to suspending a lease if this resolves policy violations and does not cause new policies violations. The search algorithm is a tree-based, depth-first search heuristic with backtracking. Backtracking, i.e. abandoning a branch and jumping to a preferred node in the tree, is based on highly-informative data structures. Their design is important for the search results. Therefore the approach is also called a guided search. Once a set of manipulations has been found by the search algorithm, the GR enters the execution phase. During the execution phase, the found manipulations are tried to be applied to the cluster. However, the GR is not the only player trying to make changes to the allocation pattern. Also the Resource Usage Adaptor (RUA) and the Global Provisioner (GP) interact with the cluster. Furthermore, there can be multiple concurrent reconfiguration cycles of the GR. The questions of how to realize transactionality and concurrency between these players and how to enforce policy semantics in the execution phase, are open issues still to be discussed. Before turning to these points, the decentralized part of the VM-Scheduler is introduced, the Local Controller.

4.4.4. Local Controller

The Local Controller is a component (agent) that runs on every node that is taken care of by the VM-Scheduling Framework. It hosts two functional units, the Resource Usage Adaptor (RUA) and the Local Trigger Unit. The purpose of the RUA is to adapt the resource usage of locally running VMs according to the policies associated with that node. By using a local adaption of resource usage, the probability that the Global Reconfigurator (GR) needs to intervene in order to enforce policy compliance can be reduced. The Local Trigger Unit realizes the local trigger functionality for all policies associated with the node. It therefore is an actor-of-last-resort that in emergency cases which makes sure that no policy invalidation occurs if both the RUA and the GR fail to do so.

4.4.4.1. Resource Usage Adaptor

The Resource Usage Adaptor (RUA) operates on policies defined for resources of the local node. It adapts the share of a resource (e.g. CPU) that is used by locally running VMs. It therefore is one of the functional units responsible for ensuring policy compliance. The Local Trigger Unit also realizes the policy semantics, i.e. prevents policy invalidations, but only as an actor-of-last-resort. The Local Trigger Unit is described in the next paragraph. Another unit responsible for ensuring policy compliance is the GR. This unit was described in Section 4.4.3. In contrast to the GR, the RUA has distinctive advantages: First it is located directly on the node. If, for instance, the management server hosting the Global Scheduler - the GR is part of the Global Scheduler - crashes, then no manipulations of VMs or leases can be found and executed by the GR. In this case it is benefitial to have a locally acting instance that independently tries to adapt the resource usage of VMs according to the configured policies. Also, it is possible that the network connection is unstable, which might as well be an obstacle to executing manipulations found by the GR: Decisions are not propagated to the target nodes or executions may fail. An unstable network connection can also lead to the fact that potential policy invalidations are recognized very late, i.e. there is only view time left to find and execute manipulations in order to prevent policies from being invalidated. So, complementing a remote decision instance, which cannot be relied on absolutely, with an independent, locally acting decision instance is reasonable.

A further aspect motivates the existence of the locally acting RUA: The RUA not only acts locally and independently from the GR, but also it uses different primitives to modify the resource usage of VMs. The GR manipulates VMs using coarse-grained primitives like migration, suspend and stop: Such manipulations have an all-or-nothing effect on the resource usage of VMs, either the VM uses resources on a node or not. GR induced manipulations carry considerable overhead concerning resource usage of the manipulation itself and they take considerable time (10^0 or 10^1 second scale) to finish. Due to their duration, the frequency of applying them is limited (a currently migrating VM can only be suspended once the migration is finished). So, complementing low frequency, coarse-grained VM manipulations with high frequency, fine-scaled local methods of modifying VM resource usage is sensible.

The methods used by the RUA are OS provided means to interact with running OS processes and the kernel from user space.[63] As an example, in most cases an OS process can be stopped from using any resources within tens of a second by using the Linux signal SIGSTOP.[64] However, the RUA does complement but not replace the GR. The primitives of the GR are powerful and some resource usage problems cannot be solved locally.
First, a local adaption is appropriate only for specific resources. Such an adaption needs to be transparent towards VMs allocated to different nodes but assigned to the same lease as the to-be-managed VM. Thereby a local adaption of the usage for the resource $AllocVM$ is not appropriate because changing the number of locally running VMs, e.g. shutting down a single VM would break the communication with other VMs of the same lease. Also, for sake of generality, the adaption should be supported by any common hypervisor or provided by means the the OS. Specialized features, like changing the number of virtual CPU cores attributed to a single VM (resource $UsedCore$) at runtime is only supported by XEN and VMware ESX and therefore not considered in this thesis. Second, local adaption is not sufficient in many cases: Let there be a node with 40% MainApp CPU usage. A policy $P_{CPU} : (70, 10)$ is associated with that node. A single VM hosting a SecApp runs on this node, using 20% CPU. The VM is assigned to a lease consisting of five VMs. The other four VMs run on other nodes. If the CPU usage of the MainApp rises to 75% and is bound to stay at or above that level for an hour, then a policy invalidation will occur: The CPU usage is above 70% and a VM is running on that node. By locally adapting the CPU usage of the VM, its share of CPU usage can be decreased to ,e.g., 1%. This, on the one hand, may make this VM look crashed from the perspective of the other VMs of that lease. This will most likely crash the SecApp as well. On the other hand, the policy is invalidated and the existence of the VM on this node might interfer with the MainApp if the MainApp's resource usage rises even further. This needs to be prevented. In such cases the VM needs to be migrated or the whole lease needs to be suspended. This can only be done by the GR.
In this thesis the functionality of the RUA is realized for three specific resources, CPU, $NetIn$ and $NetOut$. An extension to other resources can be done if transparency towards other VMs is given (i.e. inter-VM communication for a lease is not affected qualitatively) and hypervisor or OS provided interfaces for this adaption exist. The RUA in this proof-of-concept approach is a classical integral controller. A integral controller ("I-Controller") is a feedback control mechanism that sets the input value of a dynamical system as a sum of the differences between a target value and observed past output values of the system.[65] In this particular case, the dynamical system is the resource, its usage

[63]User space is a privilege domain in Linux based OS. The term refers to a part of the memory usable by user space OS processes and contrasts the area used by the kernel. Processes running in user space are restricted in directly accessing devices, memory regions or manipulating process and I/O scheduling.
[64]Linux signals are asynchronous and therefore no upper limit can be given.
[65]Please refer to Franklin et al. [42] for a comprehensive introduction to control theory.

caused by the VMs is the input value and the total usage of the resource is the measured output value. So, the RUA adapts the amount of resources usable by VMs on a single host. For each resource that is controllable (CPU, $NetIn$ and $NetOut$), there is a separate control loop with resource usage being the sensor metric (output value) and an OS provided interface being used as actuator (for setting the input value). For the resource CPU, the actuator is realized by capping the CPU usage for a specific VM, for the $NetIn$, $NetOut$ resources the amount of traffic receivable or sendable per time frame by a specific VM is capped. A control loop for a policy on an arbitrary resource is depicted in Figure 4.12. This control loop is now described using the terminology of control theory:

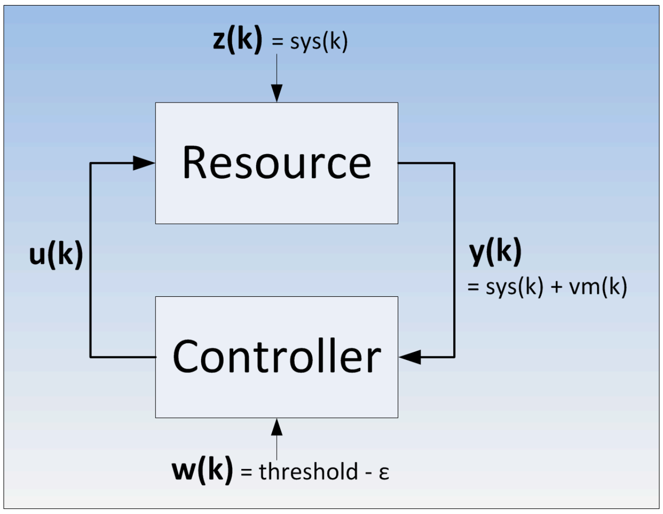


Figure 4.12.: Single control loop used by RUA for an arbitrary resource.

Let SI be the sampling interval, e.g. $1s$. Then the k^{th} sampling interval is defined as the time interval $[(k-1)SI, (k)SI)$. The controlled variable y represents the actual resource usage, thus $y(k)$ being its actual value for the k^{th} interval. The setpoint or reference value is $w(k)$, which is determined by the respective policy threshold for the resource: $w(k) = threshold - \epsilon$. The ϵ term represents a buffer (of up to 5% utilization) which is introduced to avoid accidental policy violations caused by a too ambitious reference value. The manipulated variable, i.e. the controller output is called $u(k)$. The error at the k^{th} sampling point is $e(k) = w(k) - y(k)$.
A integral controller is employed which calculates the new output as $u(k+1) = u(k) + K * e(k)$. K is the integral gain and is dimensioned empirically. From the actual resource usage $y(k)$ the share of resource usage caused by VMs can be discriminated: $y(k) = vm(k) + sys(k)$.
$sys(k)$ represents the disturbance variable (denoted z(k) in Figure 4.12 in accordance with common control theory notation) which can be identified as the MainApp and OS overhead. $vm(k)$ represents the summed actual resource usage of all n running VMs vm_i: $vm(k) = \sum_{i=1}^{n} vm_i(k)$. Therefore $u(k+1) = u(k) + K(w(k) - vm(k) - sys(k))$ holds. The resulting controller-output (i.e. the share of the resource usable by VMs) still needs to be applied by the actuator to the VMs, i.e. the controller output needs to be distributed over several running VMs. This is done by calculating $u_i(k+1) = u(k+1) * vm_i(k)/vm(k)$, which actually means that the controller output is distributed proportionally to the resource usage of VMs in the current sampling interval. It has to be mentioned that the controller does not directly set the resource usage share in the next sampling interval for a specific VM, but

rather sets an upper limit for this share. This is referred to by using the phrase "capping the resource usage".
If the disturbance is high, i.e. the resource usage of the MainApp is close or above threshold this can result in calculated resource usage shares for single VMs close or below zero. This needs to be avoided because this can break inter-VM communication. Therefore a minimum resource share of 5% is given to any running VM:

$$u'_i(k+1) = \begin{cases} u_i(k+1), & if\ u_i(k+1) \geq 5 \\ 5, & else \end{cases}$$

The actual applied controller output for all VMs is $u'(k+1) = \sum_{n=1}^{n} u'_i(k+1)$.
Such an adaption may cause the integral controller to remain in transient state longer than for a I-Controller without such an adaption. Also the threshold of the policy might be exceeded more often, which eventually could cause an increased rate of global or local triggering. This, however, is the trade-off to be accepted in order to assure proper functionality for every lease by preventing unresponsive VMs, i.e. VMs with an assigned resource share of close to 0%. More implementation-level information on how the resource usage adaption is realized for specific resources can be found in Section 5.4.

4.4.4.2. Local Trigger Unit

Beside of its task to adapt the local resource usage of VMs, the Local Controller also hosts the Local Trigger Unit which in turn realizes the local trigger functionality for all policies associated with the local node. Both the local adaption of resource usage and the manipulations chosen by the GR are meant to avoid policy invalidations, yet there is no guarantee they will do on time. The local trigger has the purpose to make sure the policy semantics are realized in any case. Like the global trigger, it bases on model-checks, yet with the difference that only local information is evaluated which comprises local policies, respective resource usage and locally running VMs. There is no need to know about the complete cluster model and allocation pattern. The local trigger is defined to be firing after $timelimit - 1$ consecutive failed model-checks[66] ($\Delta t = 1s$) if a violating VM is running on the local host. Thereby the trigger fires one second before a policy becomes invalidated. The firing of the trigger causes the instant ($< 1s$) stopping of all violating VMs on this very node such that VM processes stop to generate resource usage. Ongoing manipulations are aborted as well. This behavior is called "killing of VMs". There are two potential problems associated with the killing of VMs:
First, enforcing VM processes to stop using resources within one second requires a forceful shutdown of the VM[67] and to abort potentially ongoing VM manipulations, e.g. commanded by GR to resolve a policy violation. This may cause data loss. Implementation-wise this has to be handled such, that either this data loss is avoided, or affected VMs can be re-used after such a procedure. In Section 5.4 it is discussed how for the actual implementation of the VM-Scheduling Framework the avoidance of data loss and the recovery of affected VMs is managed.
Second, the local trigger firing is independent from potentially fired global triggers. This may cause the overlapping of manipulations caused by the local trigger with an ongoing decision or execution phase of the GR or the GP. This in turn may lead to inconsistencies in the cluster model maintained by the Global Scheduler. This aspect is discussed in Section 4.4.5.1.

[66]The Local Trigger Unit does not perform model-checks in the sense that the cluster model is queried. Only resource usage on the local node is examined.
[67]A comparable scenario is pulling the power plug of a physical computer system.

4.4.5. Functional Unit Coordination

Several functional units aiming at exploiting cluster resources (GP) and realizing policy semantics (GR, RUA and Local Trigger Unit) have been detailed. Now it will be discussed how these separate units are aligned in their functionality. This also includes the execution phases of the Global Reconfigurator (GR) and the Global Provisioner (GP). Afterwards it is discussed how policy semantics are enforced with the proposed concept.

4.4.5.1. Execution and Concurrency

Once the GR or GP end their decision phase they enter the execution phase. During this phase the found manipulations are applied to the cluster, i.e. the calculated change of the current allocation pattern is carried out. The Local Trigger Unit as part of the Local Controller may also cause a manipulation of running VMs. Because multiple independent units of the VM-Scheduling Framework can change allocation patterns, a strategy for avoiding transactional inconsistencies was developed.
Both functional units GR and GP operate on the same cluster model. That means they both use the same knowledge base to decide on changes of the current allocation pattern and need to update the cluster model upon successful execution. The most basic strategy to avoid concurrency and conflicts is a strictly sequential processsing, i.e. any functional unit trying to change the allocation pattern aquires a global lock which throughout all phases of a manipulation cycle blocks concurrent access to the cluster model until the end of a manipulation cycle. This is no viable general strategy: Based on empirical data, the potential duration of execution phases is known to be significant (e.g. resuming a VM can last more than 10 seconds). A strictly sequential processing will therefore cause functional units to wait for each other for a considerable amount of time. For the GP it is appropriate to wait several seconds, because this simply keeps the GP from provisioning leases, i.e. potentially usable resources remain unused and queuetimes of queued leases are increased. Concerning the GR such a global lock is no sensible option. A global lock on the whole cluster model (spanning even the recognition phase) would lead to the fact that potential policy invalidations are not recognized and global triggers would not fire. In this case, such potential policy invalidations would have to be treated locally, eventually leading to excessive killing of VMs by the Local Trigger Unit. A global lock which does not span the recognition, but only the decision and execution phase would lead to similiar behavior. In this case potential policy invalidations would be recognized and global triggers would fire. However, due to the lock these policy violation could not be handled (i.e. the decison phase could not be entered). This means that these policy violations would queue up at the GR, reducing the time available to resolve these violations up to the point that the Local Trigger Unit decides to kill VMs.

Therefore, the following decisions to enable a partial parallelization have been made:

1. There is at most one provisioning cycle and an unlimited number of reconfiguration cycles active at a point in time.[68]
2. There are two type of locks, one for the decision and one for the execution phase. There is no lock for the recognition phase. The lock for the decision phase is a global, pessimistic lock for the whole cluster model. This means that at any point in time only a single decision phase of any unit can be active. The lock for the execution phase is not a global lock, but a pessimistic

[68]Implementation-wise this means that the GR runs multi-threaded and the GP is represented by a single thread.

lock on VM-level. This means, that an ongoing manipulation of a VM will put a lock on this VM, excluding it from subsequent decision and execution phases. This also means, that in a subsequent manipulation cycle the lease which that VM is assigned to can not be manipulated.

3. The Local Trigger Unit is independent from the globally acting units GR and GP. This means that the mentioned locks do not apply to local triggering and the subsequent killing of VMs and abortions of ongoing manipulations. Any successful killing of VMs is immediately propagated to the Global Scheduler, the allocation pattern (represented in the cluster model) is updated and this change is taken into acount by following manipulation cycles. The abortion of an ongoing manipulation is automatically detected by GR or GP as a failed manipulation, i.e. these units update the allocation pattern accordingly.
4. Not every single policy violation makes the GR enter a decision phase. Policy violations are collected over a reconfigurable interval (e.g. $2s$) and processed together in a single decision phase. This may limit the time available for the decision and execution phases, but reduces the number of concurrently active reconfiguration cycles.
5. The GP is subordinated towards the GR. This means that while the GP holds the global lock for the decision phase, the GR can step in, acquire this lock and abort the ongoing decision phase of the GP. The VM-level locks for the execution phase that are held by the GP can not be captured by the GR, however.

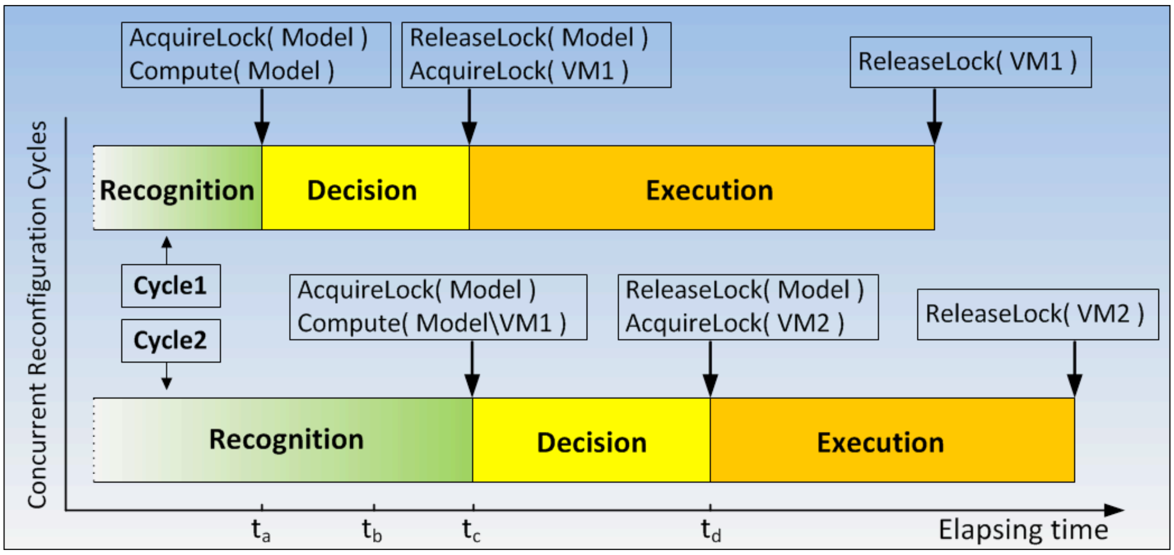


Figure 4.13.: Two concurrent reconfiguration cycles. $Cycle1$ blocks the decision phase of $Cycle2$, because $Cycle1$ holds a global lock. $Cycle2$ enters the decision phase at t_c, but excludes solutions involving $VM1$ since $Cycle1$ still holds a VM-level lock on $VM1$.

An example for the concurrency of multiple reconfiguration cyles is shown in Figure 4.13. Here the GR periodically (e.g. $1/s$) performs model-checks. This is indicated by the boxes labeled "Recognition" that belong to the upper and the lower reconfiguration cycles ($Cycle1$ and $Cycle2$) shown in this figure. The dashed line on the left of these boxes symbolizes that there is no specific starting point for this checking, but it is performed as long as the framework is running.[69] At t_a a policy violation is detected. A decision phase is started and the corresponding reconfiguration cycle ($Cycle1$) acquires a global lock on the cluster model. No active executions are going on, i.e. no VM-level lock is held by other manipulation cycles, therefore the whole model is processed for finding potential manipulations. During the search another policy violation is detected at a point in time t_b. This violation is taken care of by $Cycle2$. Since the global lock is held by $Cycle1$ this second cycle $Cycle2$ has to wait. $Cycle1$ finds an appropriate manipulation, e.g. the migration of $VM1$. At t_c it releases the global lock, acquires a lock on $VM1$ and starts executing the manipulation. Once the global lock was released by $Cycle1$ at t_c, $Cycle2$ grabs the global lock and enters the decision phase. Since for $VM1$ a VM-level lock is held by $Cycle1$, this VM and its lease can not be manipulated, therefore only manipulations excluding these are considered.[70] A manipulation for $VM2$ is found as a potential solution. At t_d $Cycle2$ releases the global lock, acquires the VM-level lock for $VM2$ and enters the execution phase. Both cycles successfully execute their manipulations and release their VM-level locks.

4.4.5.2. Enforcement of Policy Semantics

According to the defined policy semantics there is a strict upper limit for the interval (timelimit) VMs are allowed to contribute to the exceedance of policy thresholds. The functional units responsible for maintaining policy compliance (GR, RUA and Local Trigger Unit) have been introduced, yet no explicit discussion on how to obey the timelimit of policies was made so far.
Four aspects contribute to meeting the deadline set by a policy timelimit, i.e. to preventing policy invalidation:

1. First there is the RUA. By modifying resource usage of VMs locally, many situations that otherwisely would cause globally triggered changes of the allocation pattern can be handled locally. This is shown in an experiment (see Section 6.4.3). By diminishing the amount of policy violations treated by GR also the amount of decisions by GR that could invalidate policies due to a misestimation by FE is reduced. Therefore the RUA decreases the occurrence probability of invalidated policies.
2. Once a global trigger fires for a policy, the GR has to find a suitable change of the allocation pattern. It therefore uses the Feasibility Estimator to assess potential manipulations. Here only appropriate manipulations will be chosen that assumingly will not invalidate violated policies, e.g. by exceeding the timelimit when being executed during the execution phase. Given that a realistic estimation is made by FE, this approach reduces the probability of policy invalidations.
3. The former aspect dealt with the duration and overhead of manipulations during the execution phase of a reconfiguration cycle. However, it is also important that the finding of manipulations during the decision phase does not waste too much time at expense of the time available for executing manipulations. Two strategies have been adopted for limiting the duration of the decision phase:
   a) The algorithm used by GR for finding a change of the allocation pattern is a guided heuristic search. The guidance is provided by incorporating information returned by the *check-Branch(branch)* function and by having multiple lists that steer the search for appropriate

[69]Because there is no locking involved in the recognition phase, a single recognition phase is shared by all reconfiguration cycles. A new decision phase (reconfiguration cycle) is spawned upon detection of a policy violation.
[70]Implementationwise this also excludes manipulations involving the physical node to which $VM1$ is being migrated by $Cycle1$.

manipulations. This allows for an acceptable computational complexity, which is shown later in Section 4.4.6.1. This spares time for the execution phase which in turn decreases the probability that the chosen manipulations will invalidate violated policies.

b) A reduced time spent in the decision phase may decrease the probability of policy invalidations. However, any timespan needed to find appropriate manipulations reduces the amount of time available to the execution phase. Therefore this timespan needs to be taken into account during the decision phase. This aspect was not explicitely mentioned when detailing the GR algorithm, yet was considered during development. First, when *checkBranch(branch)* is called, the time left until a violated policy is potentially invalidated ($timelimit - (triggerSensitivity + time\ spent\ in\ decision\ phase)$) is taken into account when assessing the appropriateness of a set of manipulations. Second, the ongoing search for appropriate manipulations may, due to elapsing time, render already committed manipulations in the search tree inappropriate. This problem is solved by applying a meta-check to the search algorithm. During the decision phase, it is continually checked whether an already committed branch (a set of appropriate manipulations) can still be executed safely without invalidating violated policies if time advances further. If this check fails the search is immediately stopped. All leases which still contribute to a policy violation and are not part of the committed set of manipulations, are set to be stopped in addition to the already committed manipulations. Thereby the duration of the decision phase is supervised and limited such that any chosen change of the allocation pattern can be executed within timelimit. This decreases the probability of invalidated policies.

4. The previous points dealt with how to reduce the probability of invalidated policies. The Local Trigger Unit on each node makes sure that this probability will tend to zero. In the case that chosen manipulations take too long and a policy might get invalidated, the local trigger of a policy will prevent this from happening via killing of VMs and aborting manipulations. Also, if no manipulation decision by the GR was made on time (e.g. due to server crash) or failed to be executed (e.g. due to network failure), the local trigger will avoid the policy invalidation. Nevertheless, the measures mentioned above to reduce the probability of policy invalidity are still needed. Even though the local trigger makes sure that no policy invalidations occur, the primitive it uses has a negative impact towards the result output of the SecApp or even requires to recover an inconsistent VM afterwards. Therefore the mechanisms mentioned above not only contribute to the reduction of policy invalidation probability, but also decrease the relative amount of cases where the Local Trigger Unit needs to step in.

4.4.6. Further Considerations

The basic functional units and their interplay have been presented. Before describing implementational details in the next chapter, some additional considerations concerning the presented conceptual approach are made. First the computational complexity of the GR algorithm is analyzed. Then the scalability of the framework is discussed and finally the possibility of self-sustaining VM manipulation behavior is considered.

4.4.6.1. Computational Complexity

In order to maintain policy compliance it is required to obey the timelimit of specified policies. Unless satisfied by the RUA, this requirement particularly implies that the duration of the decision and the execution phases of a reconfiguration cycle is a relevant factor for avoiding the invalidation of violated policies. For the decision phase a measure can be given to describe the time properties of that phase: The (computational) time complexity of an algorithm is a measure to describe its runtime behavior with respect to the number of computational steps carried out for a given input. It can be used as an abstracted quantification of the algorithm's duration. The time complexity of the search heuristic employed by the GR in the decision phase is now determined.

Let there be a cluster environment with a set of nodes $\{node\}$ and a current allocation pattern with provisioned leases $\{lease\}$ and running VMs $\{VM\}$. The variables $N = |\{node\}|$, $L = |\{lease\}|$ and $V = |\{VM\}|$ denote the cardinality of these sets. There are no implicit restrictions which VM can run on which physical node, i.e. any possible allocation pattern is only constrained by the set of specified policies. Let there be a reconfiguration cycle which entered the decision phase upon the violation of at least one policy. For ease of mathematical handling it is assumed that any lease has the same number of assigned VMs, thereby $\frac{V}{L} \in \mathbb{N}$ holds as well. For making a statement on the worst-case time complexity of the developed search heuristic it is assumed that any single VM is contributing to a policy violation.

The basic unit (operation) on which the derivation of the time complexity is based is the check of a proposed set of manipulations for appropriateness using the *checkBranch(branch)* function.[71] This set consists of elements $\in \{suspend(lease), stop(lease), migrate(VM)\}$. Any $suspend(lease)$ represents the suspending of all assigned VMs, i.e. it merely abbreviates the $\{suspend(VM) \mid VM \in \Sigma \wedge assign(\Sigma, lease)\}$ notation.[72] For $stop(lease)$ this statement holds accordingly. For any VM at most one VM manipulation (migrate, suspend, stop) is allowed for a proposed set of manipulations in a single reconfiguration cycle. It is debatable whether this basic unit (checking a set of manipulations for appropriateness) is sufficient for determining the algorithm's time complexity. It is clear that using a finer granularity also increases the plausibility of the time complexity calculation. However, this will necessarily also require to increase the degrees of freedom for the time complexity equation, thereby requiring to consider more input parameters like the number and type (configuration) of nodes, policies and VMs as well as any policy's association with cluster items sets (nodes) and violation state. It is assumed that the proposed basic unit (checking a set of manipulations for appropriateness) merely comprises a number of condition checks that can efficiently be implemented and even be parallelized due to the atomic, independent nature of the single conditions. This is also the reason why the algorithm for doing such a check has not been detailed in this thesis and is not regarded as a computational challenge. Therefore, the time complexity of the proposed search heuristic is calculated with that basic unit (appropriateness check). The complexity then depends on input parameters N, V and L.
For having a comparison, first the time complexity of finding appropriate manipulations using a brute-force search is determined. Afterwards the time complexity of the search heuristic proposed in this thesis is discussed.

Brute-Force Search Using a brute-force search the complexity is calculated as given below. The worst-case time complexity is given by:

$$T(V, L, N) = 1 + \sum_{i=0}^{L-1} 2^i \binom{L}{i} (N-1)^{V - Vi/L} \tag{4.9}$$

[71] This means for a set of manipulations it is tested whether these will suffice to resolve a policy violation and do not trigger violations of other policies.

[72] Σ represents a set of VMs which is assigned to a lease.

The derivation of this Equation 4.9 can be found in Section A.2. $T(V, L, N)$ can be transformed to
$T(V, L, N) = 1 + (N-1)^V \sum_{i=0}^{L-1} 2^i \binom{L}{i} (\frac{1}{(N-1)^{V/L}})^i = 1 + (N-1)^V \sum_{i=0}^{L-1} \binom{L}{i} (\frac{2}{(N-1)^{V/L}})^i$
By applying the binomial theorem this gives

$$\begin{aligned}
&= 1 + (N-1)^V \left(\left(1 + (\tfrac{2}{(N-1)^{V/L}})\right)^L - (\tfrac{2}{(N-1)^{V/L}})^L \right) \\
&= 1 - 2^L + (N-1)^V \left(1 + (\tfrac{2}{(N-1)^{V/L}})^L\right) \\
&> 1 - 2^L + (N-1)^V \left(1^L + (\tfrac{2}{(N-1)^{V/L}})^L + (\tfrac{2}{(N-1)^{V/L}})^{L-1}\right) \text{ for any } L \geq 2 \\
&= 1 + (N-1)^V + (N-1)^V 2^{L-1} (N-1)^{V/L}
\end{aligned}$$

Therefore $T(V, L, N) \in O(N^V 2^L)$. This means the time complexity is exponential for non-constant inputs L or V. The time complexity is polynomial for a non-constant input N. These complexities hold both for the worst and the best-case if a brute-force search is used.

Proposed Search Heuristic Using the proposed search heuristic the complexity is calculated as given below. The worst-case time complexity is given by:

$$T(V, L, N) = L + \frac{(N-1)V}{L} \left(L + \sum_{i=1}^{L-1} \sum_{j=i}^{L-1} 2^i \binom{j}{i} \right) \tag{4.10}$$

Again, the derivation can be found in Section A.2. By applying the properties of summed binomial coefficients follows
$T(V, L, N) = L + \frac{(N-1)V}{L} \left(L + \sum_{i=1}^{L-1} \binom{L}{i+1} 2^i \right) = L + (N-1)V \left(\sum_{i=0}^{L-1} \binom{L-1}{1} 2^i \right)$
Application of the binomial theorem gives $T(V, L, N) = L + 3^{L-1}(N-1)V$.

Therefore $T(V, L, N) \in O(3^L NV)$. This means time complexity is exponential for a non-constant input L. For non-constant inputs V or N the complexity is linear. So the proposed algorithm shows an acceptable time complexity for a non-constant number of VMs and nodes, while for a non-constant number of leases it still exhibits an exponential complexity in the worst-case. However, the worst-case time complexity is relevant only to certain extent.
First, the worst-case time complexity corresponds to a scenario that will not occur in real life. In this scenario any possible proposal for manipulating leases and VMs via migration and suspend is checked and dismissed because no set of manipulations is appropriate such that it will resolve policy violations on time. This will cause the algorithm to loop over all possible set combinations proposable by the algorithm. Finally all lease-contributing leases will be stopped. This scenario will not occur:
If a given timelimit of a violated policy indeed renders any suspend and migration impossible, then the aforementioned meta-check will stop the search before that timelimit is supposed to be exceeded, which also results in stopping of the remaining violation-contributing leases. So for a big number of leases and small timelimits, the search is stopped by the meta-check before looping through all possible manipulations. For a big number of leases and and greater timelimits, the algorithm does not need to loop through all lease manipulations because appropriate migrations or suspends will be found.
Second, there is also the best-case and average-case time complexity that matter. The best-case time complexity O(VN) is given by

$$F(V, L, N) = V(N-1) \tag{4.11}$$

which corresponds to a migration of any violating (in the assumed case every running) VM. Here the complexity is linear for non-constant inputs V or N and constant for the number of leases. The search heuristic is optimized for the average-case, assuming that the worst-case is handled by the meta-check. Giving a mathematical derivation for the average-case time complexity is difficult because assumptions on the range of parameters V, L, N and their frequency distribution for this range have to made. Also, the timelimit of violated policies (i.e. the range and distribution) then needs to be considered. This derivation is left for future work. A plausibility argument is given instead: As shown for the worst-case complexity, the critical factor is the exponential growth of the number of basic computational units with a rising number of leases $O(3^L)$. This growth derives from the fact that for any high-priority lease that contributes to a policy violation, a k-combination of preemptable low-priority leases from LOW list is tested in order to keep the high-priority lease alive. This behavior is intrinsic to the algorithm and does not vanish in the average-case. Still, several aspects suggest that the average time complexity is acceptable:

a) If lease tuples are tested for preemption in order to keep a high-priority lease running, only in rare cases the whole set of k-combinations of provisioned leases in the LOW list needs to be iterated over. For instance, due to the ordering of the LOW list, its $7th$ ($12th$) entry suggests to preempt 3 (4) low-priority leases in favour of a high-priority lease in $HIGH$. The preemption of multiple low-priority leases in most cases either resolves the policy violations which the high-priority lease was contributing to or frees enough resource slots to migrate the violating VMs of a high-priority lease. Assuming that e.g. the preemption of 4 lower-prioritized leases always suffices for keeping alive a single high-prioritiy lease, $O(3^L)$ becomes $O(128L)$[73] for the worst-case that any variation of $(suspend \mid stop)$ needs to be tested on the lease tuples in LOW. This is an acceptable time complexity.

b) With increasing timelimits the time complexity tends to $O(2^L)$ (instead of $O(3^L)$) because the variation of $(suspend \mid stop)$ for preemptable leases is spared, eliminating the 2^i factor in Equation 4.10.

c) For any lease that contributes to a policy violation and whose VMs cannot be migrated because of a low timelimit the exponent in $T(V, L, N)$ decrements by 1.

For these reasons, the computational complexity of the algorithm employed by the GR is considered acceptable from a theoretical point of view. Empirical results on how the GR fares with respect to timing and goodness of found changes of the allocation pattern[74] can be found in Chapter 6.

4.4.6.2. Scalability

As mentioned in the previous Section 4.4.6.1, the time needed for the decision phase in the worst-case exponentially rises with the number of leases and linearly rises with the number of VMs and nodes. Not only the decision phase, but also the execution phase is subject to such scalability issues: With an increasing number of VMs, leases, policies and nodes the number of potential policy violations and thus also the number of executed manipulations rises. Functional units have been introduced that *a*) reduce the number of policy violations (RUA) and *b*) take countermeasures against excessive execution times for manipulations (Local Trigger Unit). Still, it is reasonable to discuss additional

[73] More precisely, the complexity degrades to $O(128 \times <$ number of initially violation-contributing leases $>)$.

[74] Though also driven by other parameters, an indicator for excessive durations of a decision phase is the stopping of all leases for a reconfiguration cycle. An indicator for the goodness of choices made by GR is the number of computed results by SecApps.

measures that help to scale the framework with a growing environment.[75] In order to do so, potential bottlenecks for the choice and execution of manipulations have to be considered.

Global Reconfigurator The functional unit GR has to decide on which manipulations to choose in case of policy violations. The computational complexity of this task (see Section 4.4.6.1) influences the choice of manipulations. If the number of VMs, leases and nodes increases, then a computational bottleneck can occur which would lead to a preferred choice of stop manipulations or even to locally triggered kills of policy-violating VMs. The proposal is made to introduce multiple GR instances in order to circumvent this issue. This can easily be done by splitting a cluster environment into two logical subclusters, each served by a separate GR. This also requires the set of leases and VMs to be split among these.[76] Each GR then manipulates its own leases and VMs that run on the set of nodes in its scope. Since these separate GRs share no common data that needs to be synchronized and every GR only deals with a subset of all leases, VMs, policies and nodes, such a separation will eliminate a computational bottleneck for the GR.

Shared Storage Another potential bottleneck is the shared storage facility used for hosting VMs, more specifically the read/write access to its persistent storage media. An existing bottleneck increases the execution times of manipulations like suspend, resume, start and stop. In order to counteract a bottleneck for the shared storage in a growing environment, the proposal is to use separate shared storage facilities, each associated with a single GR and hosting only the VMs in its scope.

Networking Another factor contributes to increased execution times: The network bandwidth and topology. This factor is relevant for inter-node communication (e.g. for migrations), but also for the node to shared storage communication (e.g. for suspends). In this thesis it is proposed to connect a single shared storage facility to every switch serving worker nodes. Additionally a single GR is responsible for the worker nodes connected to this switch. Using this topology, the number of physical nodes communicating with each other (via migrations) and with a shared storage (e.g. for suspends) is limited and manageable.

By employing these proposals, the framework scales with an increasing number of physical nodes, VMs, leases and policies.[77] An empirical evaluation for this statement is desirable, but out of scope for this thesis.

4.4.6.3. Self-Sustaining Behavior

Any VM manipulation carries some overhead towards resource usage, e.g. the transfer of a VM's memory state over network for a migration. This means that manipulations can trigger new policy violations. Thereby a phenomenon can occur: The handling of policy violations causes manipulations which in turn again cause new policy violations. This can lead to a self-sustaining behavior which is not intended. An informal approach to capturing this notion is: Given that running applications (MainApp, SecApps) do not change in resource usage, the frequency of manipulations should tend to

[75] A growing environment means an increase in the number of VMs, leases, nodes and/or policies. If no additional measures are taken, the percentage of locally triggered kills for all executed manipulations will rise for growing environment.

[76] Accordingly also two separate Global Scheduler components with separate GPs are needed.

[77] For a rising number of policies the acquisition of resource utilization statistics needs to be scalable as well. There are existing approaches like SysMES [66] , Ganglia [44] and Nagios [76] that can be used for a scalable monitoring of cluster resources.

zero over time, i.e. VM manipulations have to be justified by changes in the resource usage of running applications. If this does not hold, then the framework exhibits self-sustaining behavior.
This problem was approached on a very basic, non-mathematical level. The goal was to avoid such behavior in the prototypical implementation. Multiple heuristic methods that aim to reduce its occurrence probability have been applied.

1. Local resource usage adaption decreases the relative number of cases where the GR needs to issue manipulations in order to avoid policy invalidation.[78] A decrease in the relative number of VM manipulations also leads to fewer policy violations caused by such manipulations.
2. If the GR needs to react, then by using the Feasibility Estimator (FE) such manipulations are preferred which most probably do not trigger new policy violations. Ongoing manipulations and their overhead are taken into account by the FE.
3. Nodes involved in a manipulation of VMs initiated by GR and GP will not be target of future manipulations commanded by GR and GP for a (reconfigurable) period of $30s$ after this manipulation. During this time interval these nodes are excluded from the search space of the GR and GP, any policy violation on such nodes will only be treated by the Local Trigger Unit. Practically this means that a node involved in a policy violation will be blocked from being considered for global VM manipulations for a certain time.
4. VMs for which a manipulation has finished are blocked for successive manipulations initiated by GR and GP for a (reconfigurable) period of $30s$. During this time interval these VMs are excluded from the search space of the GR and GP.

The latter two heuristics[79] cause that a high number of manipulations ongoing at a point in time shrinks the search space for GP and GR for a certain time interval, i.e. the amount of choosable and executable manipulations within this time frame is decreased. Using this approach, an inhibition is introduced that automatically corrects high manipulation frequencies. This basic method should in most cases suffice to counteract manipulation frequencies that are not justified by changes in the resource usage of running applications. A theoretical and more general discussion of this topic is left for future work (see Chapter 8).

4.4.7. Summary

In this section the decision logic of the framework has been detailed. Its overall goal is to give SecApps access to cluster resources and to dynamically modify their resource usage so that the MainApp is not affected, additional SecApp results can be computed and cluster usage can be improved. Several functional units realize this decision logic. First, the functional units of the Global Scheduler were explained. The Global Scheduler is a component that runs on a global management server and hosts the functional units Global Provisioner (GP) and Global Reconfigurator (GR). The GP has the purpose of giving a queued, i.e. non-provisioned lease access to the cluster resources by starting or resuming its VMs. It uses a non-preemptive, run-to-completion approach and employs a priority-scheduling algorithm with backfill. The other functional unit hosted by the Global Scheduler, the GR,

[78]This is shown in Section 6.4.3.

[79]These mechanisms, the blocking of VMs and nodes involved in a previous manipulation for a certain time frame, is not used for static policies, e.g. $P_{AllocVM}$ and $P_{UsedCore}$. The violation of such policies will be handled by GR without excluding previously concerned nodes and VMs from the search space. The reason herefore is that such policy violations and resulting manipulations cannot provoke successive violations of static policies since the usage of resources like AllocVM and UsedCore is under full control of the framework.

is a runtime scheduler that decides on how and when to manipulate a provisioned lease by suspending, stopping or migrating its VMs. Its purpose is to modify the resource usage of SecApps in order to satisfy policy constraints. The GR makes these decisions based on an informed depth-first search heuristic with backtracking. Both the GR and GP use VM manipulations to change the allocation of leases in the cluster. Since such manipulations carry considerable overhead concerning resource usage and duration, the appropriateness of manipulations with respect to the policy semantics needs to be evaluated. The Feasibility Estimator (FE) was developed as a mean to estimate future resource usage. GR and GP incorporate these estimations into their decisions and thereby only choose manipulations that presumably will not lead to further policy violations and resolve existing ones. However, this approach of manipulating VMs based on estimations in order to dynamically adapt the resource usage of SecApps has two drawbacks. First, the control of resource usage is relatively coarse-grained. Second, centralized and remote decision making is inherently prone to (hardware, software) errors and the made decisions themselves may be inappropriate because they base on estimations. In order to complement the global, coarse-grained decision making of the Global Scheduler, the component Local Controller has been developed. The Local Controller runs decentralized on each node and hosts two functional units, the Resource Usage Adaptor (RUA) and the Local Trigger Unit. The RUA continuously modifies the share of resources like CPU or NetOut usable by a locally running VM. The adaption is realized by a closed loop integral controller per single resource. The RUA provides a fine-grained control over the resource usage of SecApps and therefore complements the coarse-grained resource allocation of the GR. The Local Trigger Unit is an actor-of-last-resort and realizes the policy semantics by forcefully stopping and aborting VMs, unless policy compliance can be maintained by GR and RUA. The Local Trigger Unit therefore counteracts the second issue mentioned for the Global Scheduler: Erroneous global VM manipulation decisions will not lead to policy invalidations because policy-violating VMs are stopped locally from using resources on time.

The proposed approach realizes the semantics of a set of defined policies. The distinction between different functional units allows for a partial parallelization of the conceptual approach: The units of the Local Controller (RUA and Local Trigger Unit) act independently and in concurrency to the functional units of the Global Scheduler (GP and GR). The GR is parallelized such that it can run multiple concurrent reconfiguration cycles that take care of different resource usage problems (policy violations) occurring in a staged manner. For the coordination between multiple reconfiguration cycles of the GR and provisioning cycles of the GP, a specific locking scheme was developed to enable transactionality while still providing a partial parallelization. Potential self-sustaining behavior of the framework is counteracted by excluding VMs and nodes that have taken part in VM manipulations from future manipulations for a reconfigurable time frame of one minute.

5. Realization

Based on the ideas detailed in the last chapter, the proposed approach was implemented in software. The developed software is referred to as "VM-Scheduling Framework" or "framework" throughout this chapter. Purpose and functional aspects of the framework have already been discussed, so the first section of this chapter briefly portrays the software environment and used tools. The framework design is discussed in the second section. The third section introduces use cases and actors. In the fourth section several details concerning the environment-specific implementation are described.

5.1. Environment, Products and Tools

The development of the framework as well as the empirical evaluation was done on x86-64 servers running a Linux derivate, Gentoo Linux. Platform-independent technologies were chosen for the sake of generality and portability. Where a platform-specific implementation could not be avoided it was ensured that these components are encapsulated and therefore easily replacable if the framework is to be run in a x86-64 cluster equipped with a non-Unix compatible Operating System (OS) like MS Windows OS. This concerns platform-specific components *a*) for the retrieval/read-out of resource usage statistics and *b*) the local adaption of resource usage caused by Virtual Machines.

Development The VM-Scheduling Framework was developed following the "rapid prototyping" software development pattern.[1] The development was done using command-line editors Vim and GNU Emacs. A software versioning system, Apache Subversion, was utilized.

Programming The framework is implemented in the Python programming language. Python is an interpreted language with interpreters available for MS Windows, Mac and Unix OS families. This ensures the aforementioned portability for the developped software. The adopted programming paradigm is object-oriented programming. Inter-process communication between framework components was implemented using Python socket libraries. Both synchronous and asynchronous communciation paradigms were employed. Parallelism in the framework was realized using Python threading libraries. The implementation of platform-specific components responsible for resource usage read-out and adaption uses Linux signals, procfs and tools like pv and cpulimit. These components are mentioned separately in Section 5.4.

Virtual Machines As virtualization platform Oracle VirtualBox 3.14 ("VBox") was used for the development and evaluation of the framework. The product was configured to run in the so-called "headless" mode which means that a Virtual Machine (VM) runs decoupled from a user's session in background and no Graphical User Interface (GUI) is needed to manipulate the state of VMs. VBox offers multiple interfaces to communicate with the hypervisor and single VMs, of which the Command-line Interface (CLI) "VBoxManage" and the Python Application Programming Interface (API) are used in

[1]The rapid prototyping pattern is a software development approach that bases on minimal initial planning and incremental refinement of specification and implementation.

the framework implementation. These interfaces are instrumented in a wrapper class that encapsulates and abstracts the access to arbitrary VMs. In order to run the framework using a different virtualization platform, the current wrapper class needs to be exchanged with a wrapper class that instruments the specific interfaces for the other product.
A running VBox VM is represented by a single OS process ("VBoxHeadless"). This offers the additional possibility to communicate with a specific VM using Linux signals, e.g. a VM can be hard-stopped by sending the respective process a kill signal. Sending a kill signal will also abort ongoing manipulations (e.g. a suspend).[2] This communication type is also encapsulated inside of the wrapper class. Virtual Disks used by a VBox VM are represented by Virtual Disk Image (VDI) files. VM specific meta-information like virtual hardware specification is contained in an eXtensible Markup Language (XML)-formatted settings file. Both VM specific files, the virtual disk file and the settings file are hosted on a shared storage server via Network File System (NFS). Any physical node equipped with a VirtualBox installation mounts the remote filesystem. Thereby the local hypervisor acquires access to these files and can run and manipulate VBox VMs.

5.2. Framework Design

The functional units of the framework (Resource Usage Adaptor (RUA), Global Provisioner (GP), Global Reconfigurator (GR)), their purpose and semantics have been discussed in the previous chapter. Also the two main architectural components which host these functional units, the Global Scheduler and the Local Controller, were tentatively introduced. Regarding software design of the framework, the implementation-level description is now stated more precisely. The two architectural components Global Scheduler and Local Controller are software packages. These packages are organizational units that group the framework from an architectural point of view. The basic design of the framework is given in Figure 5.1. It shows the two software packages and their communication paths. The packages are now discussed separately.

Global Scheduler Package This package is installed on a management node[3] and consists of specific classes and components. For every functional unit involved in making decisions about the change of the allocation pattern (GP, GR) there is a component with the same name that is part of the Global Scheduler package. These components implement the functionality of the respective functional units. Additionally, the package contains classes used for implementing the cluster model and interface classes. The **Cluster Model** in Figure 5.1 is a set of classes that not only represent the cluster model as defined in Section 4.3.5, but the whole data base that all decision instances operate on, i.e. the cluster model, actual and potential future incarnations, the current allocation pattern, the queue, non-provisioned leases and non-assigned VMs. The interface communication classes ("User IF", "Cluster Model IF" and "Manipulation IF") encapsule communication logic in order to provide interfaces (in the generic sense of the term) to actors and components outside of the scope of the Global Scheduler package. These communication partners are users of the framework (see Section 5.3) and Local Controller instances.

[2]Sending signals is not the specified way of communicating with VBox VMs. It is however a method to instantly relieve a node from resource load generated by a VBox VM.

[3]A management node is a physical host that is dedicated to hosting administration and management applications only, rather than performing, e.g., scientific computations.

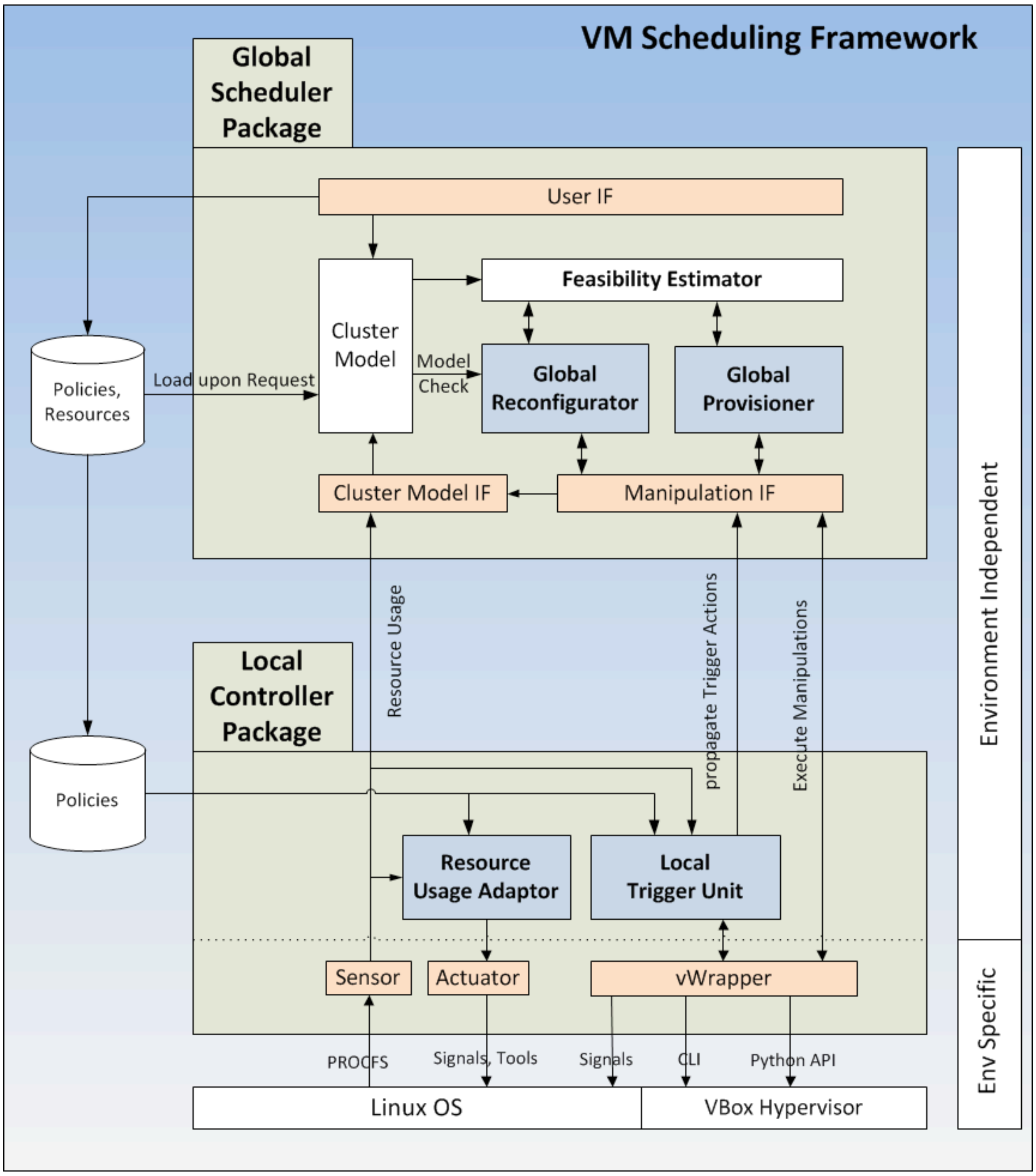


Figure 5.1.: Abstracted package diagram of the VM-Scheduling Framework with the main architectural components and their interaction.

Local Controller Package This package is installed on a worker node.[4] Since a cluster consists of more than one worker node, multiple installations of the Local Controller package commonly exist.

[4]A worker node is a physical host eligible to run a Secondary Application (SecApp), so basically every node with a VBox installation.

These instances are referred to as agents.[5] The Local Controller package contains the like-named components implementing the functional units RUA and Local Trigger Unit. Also it contains interface classes that encapsulate the communication with the local environment:
First there is the **Sensor** class responsible for acquiring utilization statistics for every relevant local resource, second there is the **Actuator** class that realizes the adaption of resource usage commanded by the RUA and third there is the **vWrapper** class that bundles the logic for communicating with local VMs and the hypervisor. All packages, components and classes mentioned so far are implemented in Python. Additionally these are fully platform- and environment-independent except for the interface classes of the Local Controller package. These interface classes are tailored towards the specific virtualization platform (vWrapper) and operating system (Sensor, Actuator) and therefore make use of existing, platform-specific APIs and CLIs.

Persistent Storage Both packages rely on an existing local installation of SQLite Database Management System (DBMS) for persistent data storage. The Global Scheduler package uses a database for storing cluster item and resource specifications as well as all configured policies. The Local Controller package uses a database for storing the set of specified policies associated with the local node.

Package Communication The Global Scheduler communicates with all Local Controller agents. Implementation-wise this is realized using Python sockets. The communication is steered by two interface communication classes: First there is the **Manipulation IF** class. This class takes control of every VM manipulation which is commanded by the Global Scheduler and initiated by GR or GP. The Manipulation IF class informs the Local Controller agents about to be executed VM manipulations. It thereby keeps track of these manipulations and captures their execution status, i.e. whether a manipulation successfully finished or failed. A locally triggered abortion of manipulations is thereby recognized by the Manipulation IF class as a failed manipulation. A locally triggered killing of a VM is also announced by the Local Controller agent to the Manipulation IF class. So the Manipulation IF class is the single-point-of-control that knows about actually performed changes to the allocation pattern. This knowledge needs to be incorporated into the Cluster Model. Therefore the Cluster Model classes have a unified interface class that allows to apply changes to them. This **Cluster Model IF** class is therefore used by the Manipulation IF class to update the current representation of the allocation pattern. Additionally, this Cluster Model IF class also takes responsibility of updating the resource utilization statistics. Depending on the type of cluster item, this information is either retrieved by publish/subscribe from the Sensor classes of Local Controller agents or actively pulled from passive Simple Network Management Protocol (SNMP) devices like switches. According to this description, any Local Controller interacts with the Global Scheduler. However, no separate communication classes were implemented for this communication on side of the Local Controller Package. That means that the Sensor, Local Trigger Unit and vWrapper classes directly communicate with the corresponding interface communication classes on side of the Global Scheduler package.

5.3. Actors and Use Cases

For the actual software implementation two main actors were considered: the lease owner ("owner") and the administrator. The owner is an abstracted person (role) that is interested in running its proprietary application (SecApp) in a dedicated cluster. The administrator is an abstracted person (role)

[5]In contrast an instance of the Global Scheduler is called master or scheduler.

that both takes care of the business goal associated with the service (providing a cluster environment to owners for running their SecApps) and the goal of maintaining the functionality of the Main Application (MainApp), i.e. to avoid strong interference. The use cases associated with these actors are shown in Figure 5.2.

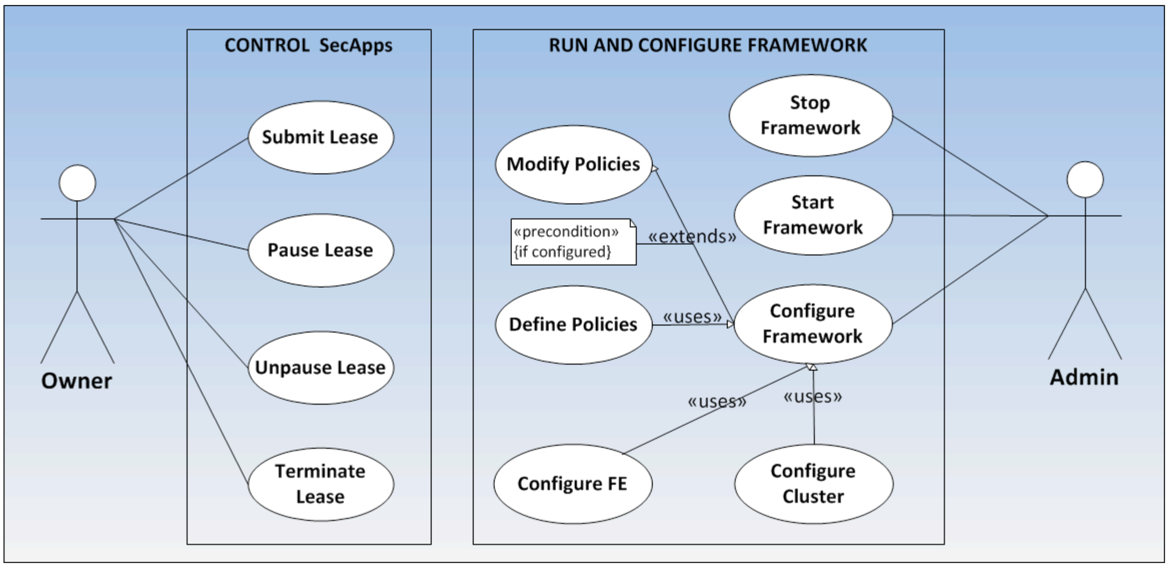


Figure 5.2.: Use cases for the actors owner and administrator.

The owner controls the state of its proprietary SecApp via lease manipulations, i.e. by submitting, pausing, unpausing and terminating of a lease. These actions correspond to the respective state transitions in a lease's life-cycle. The administrator controls the configuration of the framework and thereby defines the trade-off between providing resources for running SecApps and interference generated towards the MainApp. A main task of the administrator therefore is to define to which extent resources can be used by SecApps. This is done by configuring a target environment (cluster items, nodes, resources) and a set of applied policies. The framework provides a single point-of-control such that the "Start Framework", "Stop Framework" and all owner use cases are initiated by using the **User IF** classes of the Global Scheduler on the management node. Specific tasks of the "Configure Framework" use case require less convenient manual interaction, like installing the VirtualBox platform on worker nodes.

Owner Interfaces For the owner role, four use cases $(Submit \mid Pause \mid Unpause \mid Terminate)\ lease$ exist. Any of these possibilities to interact with the framework is provided as a dedicated CLI. The python scripts implementing these use cases can be found in Table 5.1, along with a short description of their purpose.

Script Name	Purpose
vsub.py	Submit a lease
vsusp.py	Pause a lease
vres.py	Unpause a lease
vdel.py	Terminate a lease
vinfo.py	Query Information

Table 5.1.: Command-line interfaces for lease owners.

The precise syntax for using a script can be queried by using the command-line parameter "–help". As an example for instrumenting the interfaces, the submission of a lease (**use case "Submit lease"**) is done as follows:

```
[tism001] ~/vsched $ ./vinfo.py --app
distcc  10
oge     15
[tism001] ~/vsched $ ./vsub.py --app distcc --size 5 --prio 10
Your lease has been created with ID:    1
```

In this example, the framework is first queried for existing appliances (preconfigured VMs). The scheduler returns that 10 VMs hosting a distcc[6] SecApp and 15 VMs hosting an OGE [81] job scheduling client are governed by the VM-Scheduling Framework.[7] Using this information, a lease request for five VMs running distcc is submitted.[8] The request is accepted, so a lease is created and information on it is returned. Using the returned lease ID, the owner can reference the lease for later manipulation or querying status information.

Administrator Interfaces According to the presented use cases, for the administrator additional options to interact with the framework exist. A precondition to starting and stopping the framework (use cases "Start Framework", "Stop Framework") is the proper configuration of the environment and the framework. This encompasses defining and preparing the worker nodes in scope of the framework (use case "Configure Cluster"), the configuration of policies (use case "Define Policies") and the initialization of the Feasibility Estimator (FE) (use case "Configure FE").

For the **use case "Configure Cluster"** the installation of the software packages and SQLite DBMS on management and worker nodes as well as the installation of the virtualization platform VirtualBox on worker nodes is required. For the **use case "Configure Framework"** information on the worker nodes needs to made be available to the Global Scheduler and policies need to be defined. A simple approach has been chosen in the current implementation: All information is directly manipulated in the database on the management node using an SQLite client. Table "Resource" contains all cluster items (e.g. nodes), their resources (e.g. Central Processing Unit (CPU)), static information on that resource (e.g. total CPU cores available on node) and information on how to access the resource utilization statistics, e.g. whether the utilization is delivered by a Local Controller sensor or is

[6]Distcc is a distributed compiler for the C programming language.
[7]The creation of appliances is not in scope of this thesis. Appliance building and registration with the framework needs to be done in advance.
[8]In this prototypical implementation the lease owner is allowed to freely choose a priority $\in [1, 10]$.

queried via the SNMP protocol. The **use case "Define Policies"** is used for adapting the framework to the MainApp and realized accordingly: Table "Policy" contains the configured policies, a resource it is defined on, the threshold and timelimit. A table "PolAssociation" contains the association between policies from the "Policy" table and nodes specified in the "Resource" table. For instance, $P_{CPU} = (80, 10) \times (Node_4, Node_5)$ is a policy that specifies usable CPU resources on $Node_4$ and $Node_5$. Accordingly its defined as a join over tables "Resource", "Policy" and "PolAssociation".
The **use case "Configure FE"** needs to be executed once for every new cluster environment in order to initialize the FE. At first it requires the finding of appropriate estimation parameters for VM manipulations. In Section A.5 a proposal for an according procedure is made. These parameters then have to be inserted into the `feasibilityEstimator.py` file according to the annotations given in that file. After having configured the framework and environment, the framework can be started (**use case "Start Framework"**). Via "`./schedMaster.py <loglevel>`" the Global Scheduler is started on the management node. The configuration in the database is read and the cluster model is set up using this data. Also Local Controllers agents are started automatically on all worker nodes specified in the "Resource" table. All relevant policies for a specific node are propagated to the Local Controller agent's database. Now the framework can be used by lease owners via the use cases mentioned above.

For the administrator role the framework offers the possibility to modify policies at runtime of the framework (**use case "Modify Policies"**). This is done by modifying entries in the "Policy" or "PolAssociation" tables manually via an SQLite client (e.g. the threshold of a policy on AllocVM can be changed for multiple nodes). This change needs to be announced to the framework, which is done by executing a dedicated python script "`vpol.py --update`", which in turn causes the Global Scheduler to re-read the configuration, to modify the internal cluster model and to propagate relevant changes to concerned Local Controller agents. An immediate model-check is performed by the GR to recognize potentially introduced policy violations.[9] The stopping of the framework (**use case "Stop Framework"**) is realized via issueing "`./schedMaster.py stop`".

5.4. Platform-Specific Implementation

In the Local Controller package the three classes interfacing the local environment and hypervisor (Sensor, Actuator, vWrapper) use platform- and environment-specific methods. Their functionality is now briefly explained for the prototypical implementation.

Primitive Realization In this thesis the primitives for manipulating VMs are $\{start, stop, suspend, resume, migrate\}$. These primitives are implemented in the vWrapper class. Table 5.2 shows these VM manipulations and the methods that are used for executing a manipulation. For migrating a VM, the command-line interface is used due to licensing restrictions for Oracle VirtualBox. The killing of VMs caused by the Local Trigger Unit is implemented using Linux signals, more precisely SIGTERM and SIGKILL. Upon local trigger firing a SIGTERM signal is sent and the VBox processes are expected to terminate. If this does not succeed within a second because the hypervisor blocks the termination (e.g. for transactional reasons), a SIGKILL is sent. This will forcefully terminate the VBox processes, i.e. runnning VMs are immediately stopped and ongoing manipulations are aborted. Such an approach to get rid of VMs, i.e. to end the resource usage caused by them within timelimit is certainly debatable. It was however the only method found for implementing the local triggering

[9]The administrator may for instance change the number of allocatable VMs for several nodes to zero. In this case an immediate reconfiguration cycle will be started if these nodes are running VMs.

Manipulation	Interface	Interface Provider
start	Python API	Hypervisor
stop	Python API	Hypervisor
suspend	Python API	Hypervisor
resume	Python API	Hypervisor
migrate	CLI	Hypervisor
kill	Linux signals	OS

Table 5.2.: Interfaces used in vWrapper class for VM manipulations.

for VBox in reliable fashion.[10] So, this method does yield the desired effect of enforcing policy compliance but on the other hand raises the question for its impact on the consistency of VMs and hosted SecApps. The main problem encountered in empirical evaluation was the fact that file system inconsistencies for the virtual disk of VMs occurred occasionally (in about 3% of all local trigger firings). After restarting a previously killed VM, these prevented the VM from either booting correctly or the SecApp from running without errors. Since no reliable method was found to safely abort VBox VMs and ongoing manipulations within a guaranteed short time frame, two options were left: Either the policy semantics are weakend or these inconsistencies are recovered. In the evaluation phase for the prototypical implementation it was opted for recovering of VMs: After a local trigger fires for a VM, the lease is queued, all assigned VMs are stopped and the assignment relation between the lease and VMs is dissolved. So far this is the functionality as described in Section 4.3.4. However, in order to cope with the occasional inconsistencies, this functionality was extended such that *a*) after initial creation of a VM and *b*) after every stopping of a VM in the VM's life-cycle a copy of the virtual disk file is created . Now every time the local trigger fires for a VM it becomes assignable to leases again only after a remedy procedure was executed. The procedure checks whether the VM is able to boot and run the SecApp. If not, the lastest copy of the virtual disk file is used as the new virtual disk file. This resolves the inconsistency issues at the expense of keeping a killed VM from being assigned to a lease again for about a minute (instead of instantly making it assignable again).
Further improvements *a*) using runtime disk-snapshots instead of offline copys of the virtual disk and *b*) distributed checkpointing for leases to avoid semantical, i.e. application state problems were considered but not implemented due to timing constraints. The empirical evaluation of the framework discussed in the next chapter showed the chosen remedy procedure to be sufficient for the tested environment and SecApps.

Resource Usage Metrics Acquistion As stressed in conceptual work (see Section 4.3.1), information on resource usage is the key to decide on changes of the allocation pattern. For some resources defined in Table 4.1, such as VmAlloc and SwAlloc, this information is contained in the cluster model and exclusively modified by the Manipulation IF Class. For these resources no read-out functionality for the retrieval of resource utilization data is needed. On the contrary, for resources like CPU and NetOut, the information on resource usage needs to be retrieved from the resource at runtime. This retrieval is done using the Sensor class of the Local Controller package. The Sensor acquires the current utilization using the OS-provided interfaces with the period given in Table 5.3.

[10]Using the VBox provided abort command initially promised to be a feasible solution but blocked in several cases.

Resource	OS Interface	Period (s)
CPU	/proc/stat	0.5
Mem	/proc/meminfo	5
NetIn	/proc/net/dev	1
NetOut	/proc/net/dev	1
DiskIn	/proc/diskstats	1
DiskOut	/proc/diskstats	1

Table 5.3.: Interfaces used in Sensor class for resource usage data retrieval.

The "/proc filesystem" ("procfs") offers a low-level, fine-grained access to resource utilization statistics. The current sensor implementation reads and processes the files in procfs via the Python standard library provided I/O interfaces. Processing is needed for converting cumulative statistics in procfs to current usage numbers averaged over the past measurement interval. The data obtained by the Sensor class is of interest to the Global Scheduler, the RUA and the Local Trigger Unit. The components of the Local Controller package, i.e. the Local Trigger Unit and the RUA query the Sensor class periodically to retrieve the metrics relevant for the active policies. The data for the Global Scheduler is actively propagated via publish/subscribe to the Cluster Model IF.

The retrieval, processing and propagation of resource statistics by the Sensor class induces some overhead to the CPU usage on the local node. In the evaluation up to 0.5% CPU were consumed by these tasks. The propagation of statistics to the Cluster Model IF of the Global Scheduler caused network traffic of less than 0.5 kByte per Local controller agent and second. This is acceptable for mid-sized clusters. However, an alternative read-out implementation using C and the Linux System Call Interface (SCI) and the revision of the communication paradigm between Local Controller agent and Global Scheduler should be considered in a future implementation.

Resource Usage Adaption The current implementation considers the adaption (capping) of resource shares used by local VMs for resources CPU, NetIn and NetOut. The general approach of using an integral controller to realize this adaption has been introduced in the previous chapter in Section 4.4.4.1. For the resource CPU the values for Kp and SP, i.e. the gain and sampling/actuating period of the controller were choosen to be $K = 1, SP = 0.5s$. The Actuator class is commanded by the RUA to cap the CPU usage share for each running VM according to the formula stated in Section 4.4.4.1. The method used by the Actuator class for the CPU resource is the alternating sending of signals SIGSTOP and SIGCONT to a VM-process. The variable periods for sending these signals determine the maximum CPU utilization of a (VM-)process. The Linux tool "cpulimit" with this functionality was used in the current implementation. Using this method, a maximum deviation from the target CPU usage of 0.5% was achieved.

For the resources NetIn and NetOut the values for K and SP, i.e. the gain and sampling/actuating period of the controller were choosen to be $K = 1, SP = 1s$. The Actuator class is commanded to set the maximum resource usage share for each running VM according to the aforementioned formula. While it was possible to dynamically cap the incoming and outgoing network traffic for single VMs using the virtualization platform Kernel-based Virtual Machine (KVM) [62] with the Linux tools "tc" and "iptables’", this attempt failed for Oracle VirtualBox. Therefore, a rather peculiar method was chosen for capping the network traffic in the Actuator class. The dynamical limit to network throughput was achieved by adapting the CPU usage share of the VM process. The Actuator class itself employs a control algorithm that dynamically adapts the CPU usage of a VM such that the minimum of both target settings for NetIn and Netout was set as maximum cap for both resources as a side-effect. Due to this unconventional approach the actual CPU usage target applied by the actuator therefore was the minimum of the three values for target CPU usage, calculated CPU usage to cap NetIn utilization and calculated CPU usage to cap NetOut utilization. It is clear that this is not an approach that should be used in a productive environment because it also slows down VMs (i.e. decreases their CPU usage) if only one of the resources CPU, NetIn or NetOut is heavily utilized. This leads to a less efficient SecApp performance.

However, it is possible to cap resource utilization for any of the presented resources with the proposed method. The implementation of a kernel hook for dynamic VBox traffic shaping is a task left for future work. Alternatively using a different virtualization platform like KVM should be considered. For the resources DiskIn and DiskOut which were not used in the evaluation of the implementation, the tool "ionice" can be used by the Actuator class for adapting the resource usage at runtime.

6. Experimental Results

In this chapter the experimental results obtained for the thesis are described. It is evaluated how these results relate to the thesis goals of providing an adaptable framework for exploiting unused resources without strongly interfering a Main Application (MainApp) while still improving Secondary Application (SecApp) result output. The chapter is structured as follows:
The testing environment where the evaluation took place is introduced first. Afterwards, an overview over the presented experiments is given. Then each experiment and its findings are discussed step-by-step. Finally the found results are summarized.

6.1. Test Environment

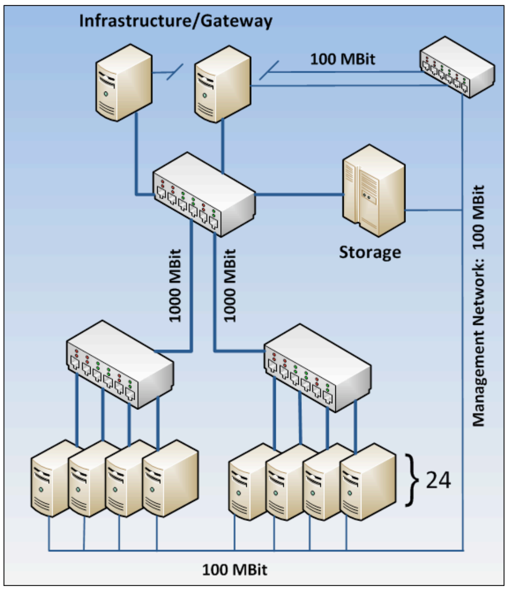
The experiments were conducted in a commodity hardware based cluster which is located at Kirchhoff Institute for Physics (KIP) in Heidelberg, Germany. This cluster consists of 24 worker nodes, several infrastructure nodes and Gigabit Ethernet interconnects. Specifically, the cluster environment makes up of:

• 2 Infrastructure Servers:	4 Intel CPU x 2.2 GHz x 4 Cores , 16 GB RAM, Linux Gentoo OS, Naming & Directory Services
• 1 File Server:	4 Intel CPU x 2.2 GHz x 4 Cores , 16 GB RAM, Linux Gentoo OS, NFS
• 24 Worker Nodes:	1 AMD CPU x 1.8 GHz x 2 Cores , 4 GB RAM, Linux Gentoo OS
• 1 Cisco Switch:	100 MBit, 80 Ports, Management/BMC Network
• 3 Netgear Switch:	1000 MBit, 40 Ports, Full Bidirectional Bandwidth

The network topology can be seen in Figure 6.1a. There is a switch hierarchy: The top-level switch connects the storage and infrastructure nodes and the low-level switches. Each low-level switch connects 12 worker-nodes. Beside of this Gigabit Ethernet based network, there is also a 100 Mbit management network for Board Management Controllers (BMC) on each node. A photographic picture of the cluster environment is given in Figure 6.1b. The virtualization platform Oracle VirtualBox 3.14 is installed on all worker nodes. VM virtual disk files are hosted on a fileserver using NFS and each worker node mounts the respective remote directory.

6.2. Experiment Overview

In this section an overview about the experiments and their contribution is given. As a reminder, the purpose of the framework is to *a*) prevent strong interference for a non-controllable MainApp and *b*) to improve efficiency metrics like generated results output and increased resource utilization for the cluster. In the conceptual work of this thesis it was proposed to use policies as configurational items that offer the possibility to find a trade-off between these two potentially conflicting goals. In



(a) Network topology and nodes

(b) Photograph, taken in november 2011

Figure 6.1.: Test environment KIP cluster in Heidelberg, Germany.

the following, experimental results are detailed that exemplify the behavior of the VM-Scheduling Framework and show the effect of policy configurations towards these goals for multiple scenarios. These experiments culminate in a final empirical test with the HLT-Chain application by which the thesis goal achievement is shown.

Policy Modification Experiment In this experiment the reaction of the VM-Scheduler towards *a*) a single policy modification and *b*) a sequence of policy modifications is evaluated. This is done by using static policies[1] that operate on the UsedCore resource, thereby specifying the maximum number of CPU cores usable by VMs on a node. By modifying these policies at runtime, policy violations are forcefully introduced which can solely be resolved by the Global Scheduler, specifically by Global Reconfigurator (GR). By recording the type and number of manipulations chosen by GR for a single modification, the goodness of choices made by GR can be assessed. For a sequence of policy modifications, the cooperation of both GR, that resolves introduced violations and Global Provisioner (GP) that tries to provision leases to free resources can be assessed. Again, this is done by recording the type and number of chosen manipulations as well as the results generated by a virtualized SecApp.

Efficiency Experiment The above mentioned "Policy Modification Experiment" discusses how the GR and GP cope with forcefully changed resource availability for SecApps introduced by explicitely modified policies on the UsedCore resource. However, there are resources, e.g. Central Processing Unit (CPU), whose usage is not fully under control of the VM-Scheduler. Their utilization not only depends on the resource usage generated by SecApps, but also on that of the MainApp. In this second

[1]As a reminder, static policies are those policies that operate on resources whose usage is dependent only on the VM-Scheduling behavior. Examples are the UsedCore and AllocVM resources whose usage only changes upon framework initiated manipulations. All other policies are called dynamic policies.

experiment dynamic policies operating on the CPU resource are employed. By applying different host load patterns which mimic different MainApps and modifying policy parameters timelimit and threshold, the effect of these parameters towards the efficiency of exploiting unused cluster resources and generating additional SecApp results is evaluated. The goal of this experiment is to demonstrate that policies with their parameters timelimit and threshold are feasible configuration items to strive for the thesis goal of improving the efficiency of cluster usage independent of a particular MainApp resource usage behavior.

Interference Experiment In the former experiment it is examined for arbitrary MainApp load patterns what effects policy parameters have towards cluster usage efficiency. However, in order to assess the interference generated by SecApps, a MainApp with a performance metric is needed. In this "Interference Experiment" a MainApp with a performance metric is run which generates considerable CPU and network utilization. Two dynamic policies operating on resources CPU and NetOut are applied. The performance metric for this MainApp is measured for different threshold and timelimit configurations. Based on the performance metric values, the amount of interference caused by additionally running SecApps is assessed. The goal of this experiment is to demonstrate that policies with their parameters timelimit and threshold are feasible configuration items to strive for the thesis goal of controlling interference towards a MainApp.

HLT-Chain Experiment In this experiment the HLT-Chain MainApp is run together with the VM-Scheduler. This experiment proves the appropriateness of the thesis approach by showing that there is a policy parameter configuration for which no strong interference occurs, the cluster utilization is improved and additional SecApps results are computed.

6.3. Policy Modification Experiment

The effect of runtime policy modifications, i.e. deliberately changed resource availability for SecApps, is examined in this experiment. A typical scenario is the decision of an administrator to restrict the usage of a (sub)cluster by SecApps at runtime. This he could do by changing the threshold parameter of static policies operating on the UsedCore[2] resource, thereby forcefully introducing policy violations for nodes with a running Virtual Machine (VM). An objective of such a modification is that concerned, now policy-violating VMs are removed from nodes within the timelimit of the respective policy and that these VMs are treated gently. Gentle treatment refers to the choice made by GR how to remove a violating VM from a node: While a migration keeps the VM alive and computing, suspend will pause the computation of results and stopping a VM even causes the loss of already computed and not yet saved results. The manipulations chosen by GR are recorded and discussed.

In a second step of this experiment the runtime modification of policies is taken further: A modification is not only applied once, but repeatedly during the experiment's run. This continually changes the number of CPU cores usable by VMs and accordingly how many VMs are allowed to run on nodes. This sub-experiment shows how GP and GR cope with this changing resource availability: The GP takes care of provisioning leases if CPU cores become "marked" as usable, the GR removes VMs from nodes if CPU cores are blocked for SecApps, i.e. when policies are violated. This sub-experiment therefore demonstrates the basic cooperation of GP and GR. In addition, it also represents an important scenario in the ALICE High Level Trigger (HLT) context: The runtime reconfiguration of the

[2]Alternatively, also a policy on the AllocVM resource is an option to control the number of VMs running in a cluster.

HLT-Chain MainApp. As described later in this chapter (see Section 6.6.1), this application consists of a multitude of processing components distributed over the cluster nodes. A reconfiguration of the HLT-Chain reallocates components from one node to another, spawns new or removes existing processing components. The modification of UsedCore policies is a possibility for the HLT-Chain to explicitely announce such a reconfiguration to the VM-Scheduler and to enforce an according adaption of VM allocations. So, in this sub-experiment a pattern of policy modifications ("reconfiguration pattern") is applied and the manipulations executed by the VM-Scheduler are recorded. Additionally, the goodness of the manipulation decisions is assessed. This is done by running a prototypical SecApp inside of VMs and comparing the amount of actually computed results to the amount of computable results in the optimal case.[3] The effect of the timelimit parameter of modified policies towards SecApp result computation and gentleness of GR decisions is evaluated.

6.3.1. Experiment Layout

The experiment is carried out in the environment described in Section 6.1, consisting of 24 worker nodes. Each cluster node is equipped with a single $P_{UsedCore}$ policy, delimiting the number of CPU cores that can be used by VMs on this node. The initial policy configuration for every node is $P_{UsedCore} : (2, t)$, indicating that the whole cluster, i.e. all CPU cores, can be used for running VMs. Before the start of each sub-experiment, leases are submitted to the VM-Scheduler. Every lease is assigned a specific number of VMs. There are

- 12 leases with 1 VM,
- 6 leases with 2 VMs,
- 2 leases with 4 VMs,
- 2 leases with 8 VMs,

making 22 leases with 48 VMs in total. The number and size of these leases has been chosen that way to avoid overly trivial scheduling scenarios, like e.g. 48 leases with 1 VM or 1 lease with 48 VMs. The VMs are configured to use one CPU core and 512MB RAM. Practically this means that any physical CPU core represents a slot for a single VM and if all leases are optimally provisioned, all physical CPU cores are used by VMs. All leases have the same priority in this experiment. The leases and their VMs are equipped with RedHat Enterprise Linux (RHEL) 5.1 [91] and run a CPU-bound SecApp, which is a GCC [43] compiler benchmark running in a closed loop. This SecApp is set up such that nearly 100% of the available CPU resources are consumed. In order to measure the number of results computed by the SecApp, a single result is defined as the completed compilation of a specific amount of source code inside of a VM. If a VM receives 100% of a CPU core[4], a single result computation takes 2.5 min.

For the **first sub-experiment**, the policy configuration is changed at runtime from $P_{UsedCore} : (2, t)$ to $P_{UsedCore} : (0, t)$ for a number of nodes $\in \{6, 12, 24\}$ and timelimits $t \in \{5, 10, 25, 50\}$. The resulting manipulations are recorded. Three experiment runs are conducted for every parameter configuration.

[3]For a non-distributed SecApp the number of optimally computable results per time is retrieved by running a single VM on an idle node for a certain time frame without manipulating the VM.

[4]This statement corresponds to running the VM in an idle environment without competing applications and therefore represents the optimal case that can be used as a reference to assess the SecApp result output achieved in the experiment.

For the **second sub-experiment**, periodical changes to policy configurations are made for random sets of nodes. These periodical changes affect the number of CPU cores available to VMs on cluster nodes. The parameters modified in this sub-experiment are the frequency of policy changes and the timelimit of the changed policies. The resulting manipulations and the number of computed SecApp results are recorded.

For repeatedly modifying policy configurations, a pattern of modifications is generated and applied throughout a sub-experiment's run. Such a reconfiguration pattern consists of reconfiguration events. A single reconfiguration event E is described by $E = (\{node\}, \{timelimit\}, \{change\}, \{type\})^C$. Variable C represents the number of nodes affected and is a random number between 0 and 24. This also means that every reconfiguration event changes the policy configuration for a randomly determined set of nodes with a cardinality of C. The timelimit variable determines the timelimit to be set for the policy of the node. For a specific experiment run, the timelimit variable is fixed to one value $\in \{5, 10, 25, 50\}$. The *change* variable represents the quantitative change in the number of CPU cores freed or blocked for VMs on a node. This natural number is modelled using a random walk with variable step size and reflective barriers at zero and two. The *type* variable denotes whether the number of CPU cores usable by VMs on a node is diminished ("down") or increased ("up"). With a probability of $1/3$ all nodes of a reconfiguration event either have $type = up$, $type = down$ or a randomly chosen value $\in \{up, down\}$. This actually means that for a single reconfiguration event either on all chosen nodes new VM slots (physical CPU cores) are provided ($P = 1/3$), VM slots are removed ($P = 1/3$) or that on some nodes new slots are provided while on others these are removed ($P = 1/3$).

A reconfiguration pattern is a sequence of events E occurring every $RI \in \{20, 40, 60\}$ seconds. RI is called the reconfiguration interval, i.e. denoting how much time lies between two reconfiguration events. For every setup (timelimit, RI), an experiment run with 30 reconfiguration events is conducted. Every combination of parameter settings is tested three times.

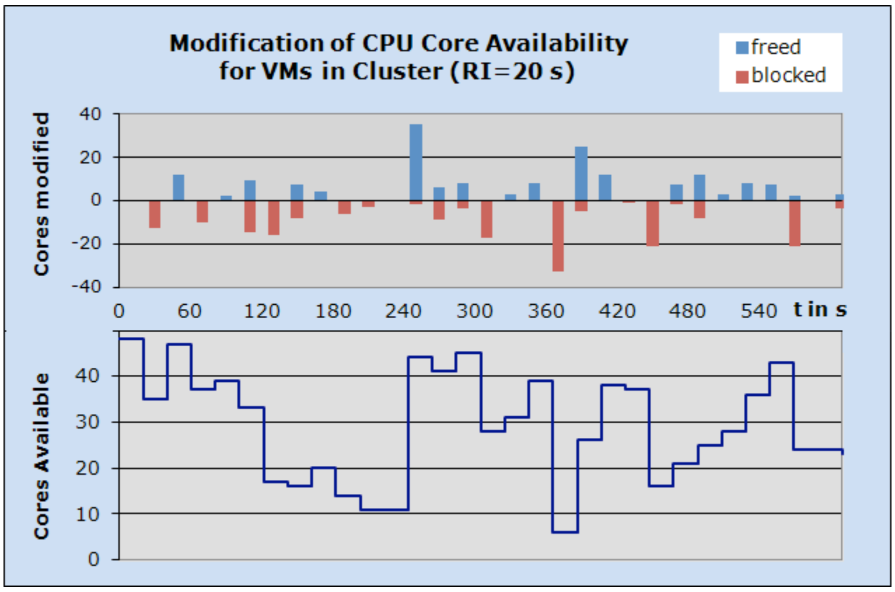


Figure 6.2.: Exemplary reconfiguration pattern ($RI = 20s$, above) and its effect on the availability of CPU cores for VMs (below).

Figure 6.2 shows an exemplary effect of a reconfiguration pattern with 30 reconfiguration events and $RI = 20s$. The total number of CPU cores usable by VMs at a specific point in time is shown in the lower part of this figure. The upper part of the figure shows the actual reconfiguration events that lead to this CPU core availability. At times, CPU cores are only taken away from VMs ("blocked") or made available to VMs ("freed"). Alternatively both happens for the same reconfiguration event (e.g. at about $120s$), which means that on some nodes CPU cores are blocked while on others they are freed.

6.3.2. Results

The **first sub-experiment** conducted is the singular blocking of a subset of cluster nodes. Figures 6.3, 6.4, 6.5 show the manipulations chosen and executed by the VM-Scheduler for blocking 6, 12 and 24 nodes using different timelimits. In any of the setups it is visible that the timelimit has a considerable effect on the type of manipulations choosen: The lower the timelimit, the less gentle the VMs are treated, i.e. the more often stops are chosen. This is especially obvious for the timelimit of 5 seconds.

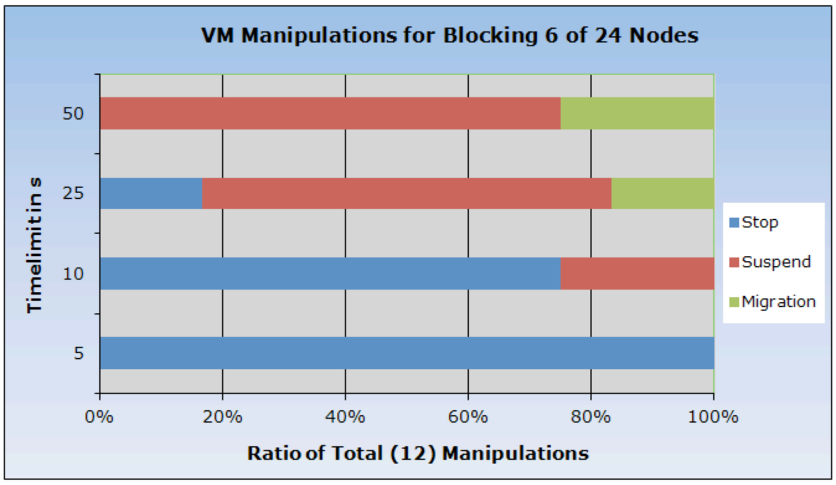


Figure 6.3.: VM manipulations chosen by GR when ordered to remove 12 running VMs from 6 nodes within a given timelimit.

An interesting point is the existence of migrations in two setups (Figures 6.3, 6.4) for timelimits of 25 and 50 seconds. At first glance, one would not expect migrations because only a number of nodes are blocked and no additional target nodes for VM migrations are made available. However, this becomes clear when taking into account the existence of leases containing more than one VM. If such a lease with e.g. 8 VMs has a single violating VM, then either this particular single VM has to be migrated to another node ("emigration") or the whole lease needs to be preempted. By stopping or suspending the lease and all of its VMs, CPU cores on nodes without a violated policy are freed which can be used for migrations of violating VMs to such a node ("immigration"). This is exactly what happens in these cases. It has to be noted that in this particular sub-experiment two configurations were experiencing kills, i.e. manipulations which could not be finished within timelimit and had to be killed by the Local Trigger Unit. For blocking 24 nodes with 48 running VMs in 25 seconds (Figure 6.5), 5 suspend manipulations had to be aborted. For blocking 12 nodes with 24 running VMs in 10 seconds,

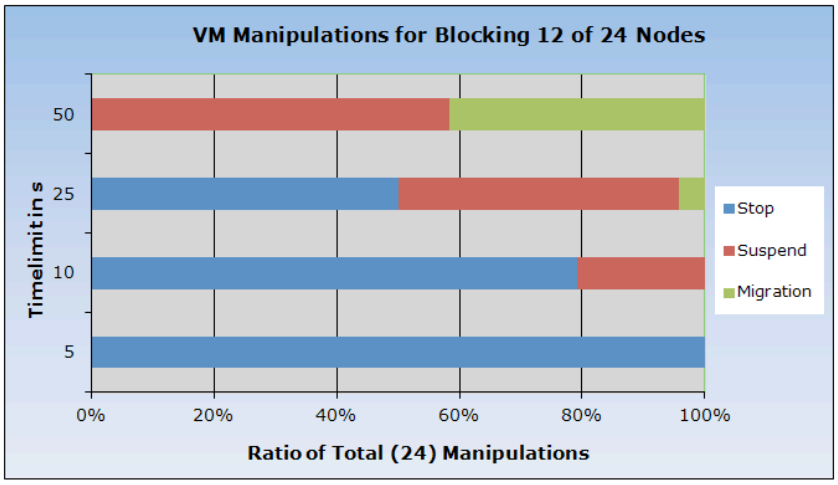


Figure 6.4.: VM manipulations chosen by GR when ordered to remove 24 running VMs from 12 nodes within a given timelimit.

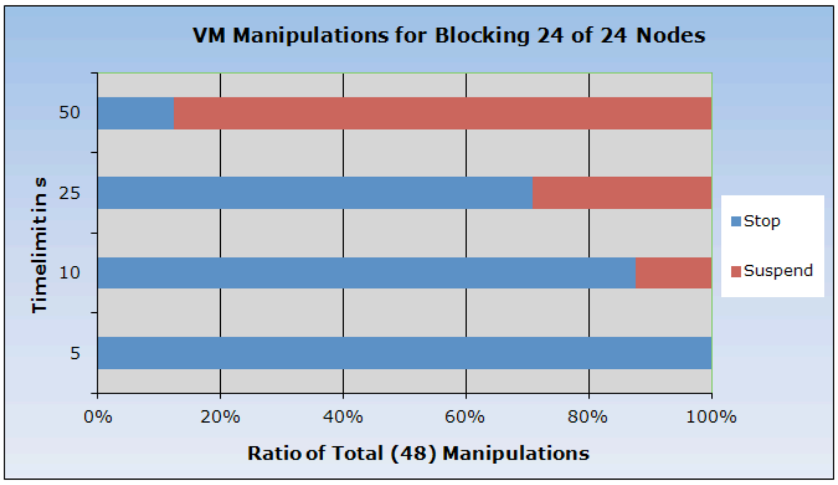


Figure 6.5.: VM manipulations chosen by GR when ordered to remove 48 running VMs from 24 nodes within a given timelimit.

6 suspends manipulations had to be aborted. This can be attributed to an insufficient estimation for the duration of these manipulations made by Feasibility Estimator (FE).
This sub-experiment demonstrates that modifications to static policies can be used to remove VMs from (sub-)clusters within a given timelimit. The chosen manipulations depend on the timelimit parameter and more gentle manipulations are chosen for higher timelimits. Timelimits were obeyed in every case even though it took the Local Controller Unit to abort manipulations in two setups.

The **second sub-experiment** uses a reconfiguration pattern which periodically changes the number of CPU cores available for VMs. Figures 6.6, 6.7 and 6.8 and show the number and types of manipulations chosen by the VM-Scheduler for experiment runs with different timelimits and reconfiguration intervals. The figures also contain the number of "Failed" manipulations, i.e. manipulations that failed to finish within timelimit and had to be aborted. In contrast to the first sub-experiment, where only manipulations for resolving policy violations were applied by GR, in this sub-experiment also the GP plays a role in provisioning leases to newly available CPU cores. Therefore, these figures also contain the manipulations start and resume chosen by GP. The figures show that with a rising timelimit the number of stops decreases and the share of suspends and migrations increases. For timelimits of 5 seconds, only stops are performed to resolve policy violations induced by reconfiguration events. This effect applies to any reconfiguration interval. Another effect visible for any reconfiguration in-

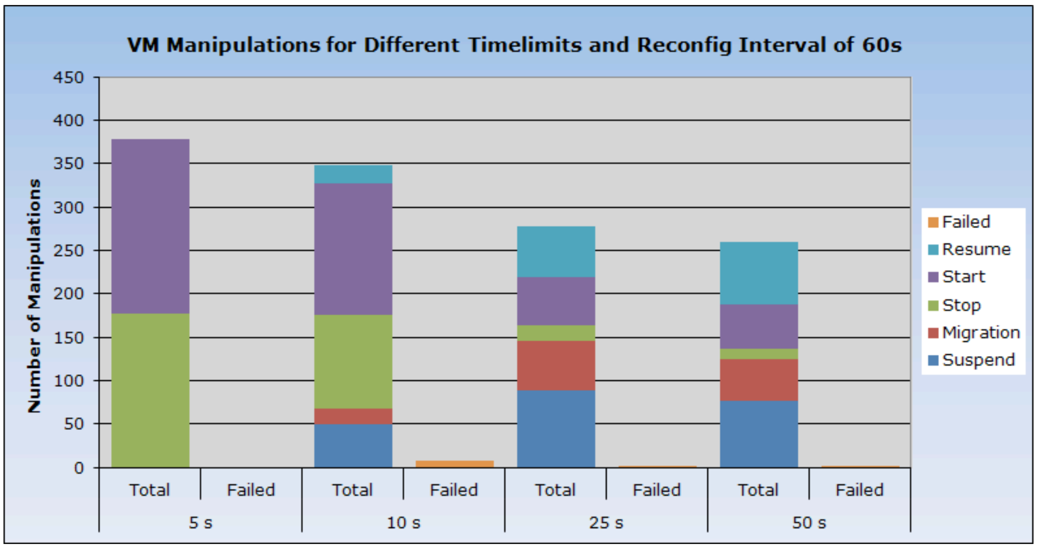


Figure 6.6.: Type and number of total executed and hereof failed VM manipulations for a reconfiguration pattern with $RI = 60s$ and experiment duration of $1800s$ for different policy timelimits.

terval is that the lower the timelimit, the more manipulations are carried out. This is explained easily by the fact that when policy violations have to be resolved quicker, then preempted leases can also be provisioned again earlier, resulting in a higher frequency of GP and GR initiated manipulations. Comparing the results for different reconfiguration intervals with each other is difficult since the experiment durations differ.[5] It is, nevertheless, obvious that the number of manipulations for a given timelimit is similiar for $RI = 40s$ and $RI = 60s$ (Figures 6.6, 6.7), but apparently lower for $RI = 20s$ in Figure 6.8. A possible explanation is that for a low reconfiguration interval of 20 seconds factors like duration of decision and execution phases play a greater role. These factors are independent from the reconfiguration interval and do not scale down with a decreased experiment duration. So probably the GP cannot keep up with provisioning of leases due to locked VMs, so less leases are provisioned and less manipulations are issued by GR in order to resolve policy violations.

While these findings are not remarkable per se, two aspects need to be mentioned: The overlapping of timelimit and reconfiguration interval and the impact of the reconfiguration interval on the efficiency of result computation. The overlapping of timelimit and reconfiguration interval, i.e. cases where the reconfiguration interval is shorter than the timelimit of policies, is of particular interest. An example

[5]The experiment durations differ because 30 reconfiguration events were used for every reconfiguration interval. The experiment duration therefore scales with the reconfiguration interval.

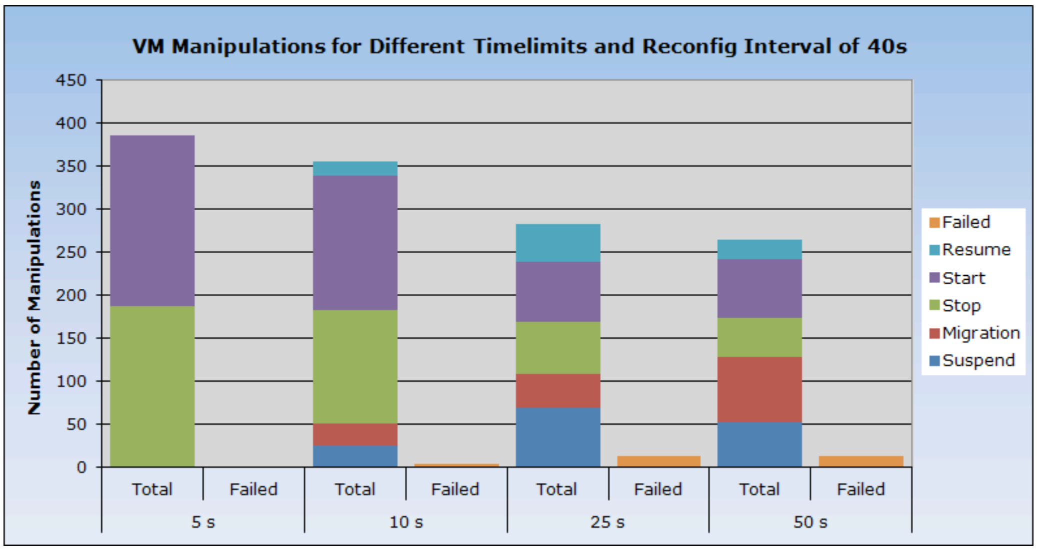


Figure 6.7.: Type and number of total executed and hereof failed VM manipulations for a reconfiguration pattern with $RI = 40s$ and experiment duration of $1200s$ for different policy timelimits.

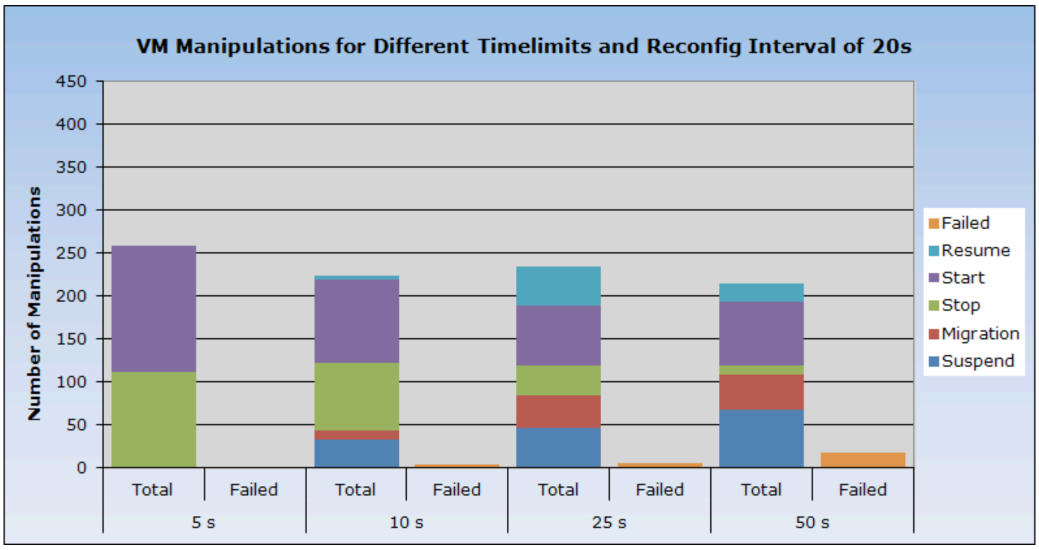


Figure 6.8.: Type and number of total executed and hereof failed VM manipulations for a reconfiguration pattern with $RI = 20s$ and experiment duration of $800s$ for different policy timelimits.

configuration is $timelimit = 50s$ and $RI = 20s$, shown in Figure 6.8. The overlapping of an ongoing manipulation by a shortly afterwards induced policy violation is something that may happen in a real life scenario as well: The administrator falsely applies a blocking of certain nodes, notices his error immediately and tries to revert it by modifying policies again. Also, the GP may decide to provision a lease and resume its VMs on nodes which are shortly after, during the execution of the resume manipulation, concerned by a policy violation. So what is happening in such a case?

First, the recognition of policy violations occurs in parallel to decision and execution phases of GR and GP. Nevertheless, during the execution phase a manipulated VM is locked by the mechanism explained in Section 4.4.5.1. This means that any subsequent manipulation of a single VM has too wait until the previous manipulation of the same VM finishes.[6] This can lead to two situations:

1. The duration of manipulation A (e.g. migration or resume) finishes within timelimit of a shortly after begin of manipulation A violated policy. In this case, the remaining time of this timelimit is used for the decision and execution phases needed for resolving the newly introduced policy violation. So, manipulations will be chosen which are shorter than otherwise possible with the given timelimit.
2. The duration of manipulation A exceeds the timelimit of a shortly after begin of manipulation A violated policy. In this case, the Local Trigger Unit will kill the according VMs and abort ongoing manipulations.

When comparing the manipulations chosen by GR for the three reconfiguration intervals, the ratios of manipulations can be interpreted in this sense: With a timelimit of 50 seconds, the GR should opt for similiar ratios of manipulations types when resolving policy violations. However, while for $RI = 60s$ and $RI = 20s$ about 60% suspends were chosen, this contrasts $RI = 40s$ with only 30% of suspend manipulations. With both $RI = 20s$ and $RI = 40s$ exhibiting an overlapping of reconfiguration interval and timelimit, for $RI = 40s$ the first situation seems to occur more often, i.e. previous manipulations decrease the time left for choosing and executing following manipulations. This results in choosing long-lasting suspends less often. This is backed by the observation that both $RI = 60s$ and $RI = 20s$ had only 8% resp. 9% stops while $RI = 40s$ has a ratio of 26% chosen stop manipulations. For $RI = 20s$ the second situation seems to occur: Previous manipulations do not finish within timelimit of successively violated policies and manipulations had to be aborted. This can be seen by looking at the ratio of kills where $RI = 20s$ has 13% of GR initiated manipulations aborted, while $RI = 40s$ has 7% and $RI = 60s$ has less than 1% aborted manipulations.

The effect of the reconfiguration intervals and timelimits towards the efficiency of SecApp result computation can be seen in Figure 6.9. Here not the absolute numbers of computed results are depicted but the relative efficiency with respect to the optimal number of computable results using the available CPU cores. The higher the reconfiguration interval, the more results can be computed for a specific timelimit. This comes at no surprise since for lower reconfiguration intervals the provisioning of preempted leases might not be able to keep up with the speed of the reconfiguration. This applies to resuming VMs, but also to starting VMs which might not even have fully booted before they are subject to manipulations again. So, for low reconfiguration intervals the overhead of starting/resuming VMs plays a greater role and therefore leads to lower efficiency in computing SecApp results. A second observation is that the longer the timelimit is, the more results can be computed for a specific reconfiguration interval. This is due to more gentle manipulations for higher timelimits. This effect is especially obvious when comparing the timelimit of 5 seconds to the timelimit of 25 seconds for any reconfiguration interval. This statement does, however, not hold for the reconfiguration interval of 20 seconds and a timelimit of 50 seconds. Here the aforementioned overlapping effect comes into play.

[6]As a reminder, a mechanism for avoiding self-sustaining scheduling behavior was introduced in Section 4.4.6.3. This mechanism blocks VMs and nodes for manipulations after a previous manipulation for a certain time frame. However, as stated in the mentioned section, the mechanism is not used by GR for static policies like the one used in this experiment. Therefore any subsequent VM manipulation by GR in this experiment can indeed start once a previous manipulation has finished.

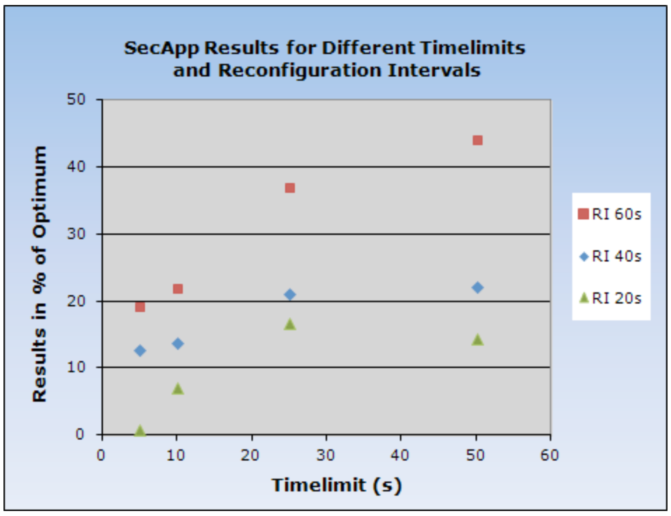


Figure 6.9.: Ratio between actually computed and optimally computable SecApp results for different reconfiguration intervals and policy timelimits.

6.3.3. Summary

In this experiment it was shown how the VM-Scheduling Framework copes with static policy modifications, i.e. with manually introduced policy violations. Parts of a cluster which was fully used by running VMs were suddenly blocked for these in the first sub-experiment. This resulted in removal of the respective VMs. The framework, specifically the GR, was able to obey to timelimits of the violated policies. Manipulations were chosen depending on the timelimit, with lower timelimits causing more stops, while higher timelimits allowed for more suspends and migrations.
In the second sub-experiment, a sequence of policy modifications was applied to the running VM Scheduling Framework. Again, the framework was able to change the allocation and state of VMs accordingly, obeying the timelimits and killing VMs in several cases where manipulations took too long. Shorter timelimits led to the preference of stops over more gentle manipulations. The longer the time between policy modifications, the more SecApp results could be computed by VMs. Both sub-experiments show that static policies and their runtime modification can be used to change the usage of a cluster by SecApps easily and conforming to policy specifications. Policy compliance is even achieved for a timelimit of 5 seconds. Giving more time to the VM-Scheduling Framework to react to policy violations will, however, increase the number of computed SecApp results considerably.

6.4. Efficiency Experiment

This experiment evaluates how the VM-Scheduling Framework copes with the changing uage of resources that are not under full control[7] of the framework. An example for such a resource is CPU, whose usage also depends on the MainApp. A dynamic policy that operates on this resource is associated with the cluster nodes. It is expected that the configuration of this policy, i.e. the settings for parameters timelimit and threshold have a significant effect towards the thesis goals of increased cluster utilization and SecApp result output. In order to evaluate this effect, a set of CPU load patterns, representing a wide range of MainApps, are generated and applied to the cluster environment. The resulting efficiency of using free CPU resources and computation of SecApp results is measured and discussed.

6.4.1. Metrics Discussion

To assess the benefit of the VM-Scheduler with respect to cluster utilization and SecApp result output, metrics need to be defined. The first aspect of interest is the generation of additional cluster resource usage. This metric, however, does not fully reflect the actual benefit of running SecApps in VMs. Three aspects influence the translation of resource usage by SecApps to actually computed results:

1. Running applications in a virtualized environment instead of natively on a node causes a decrease in application performance. This was discussed before in Section 4.3.2 and the argument was made that virtualizing SecApps is needed to provide flexibility and isolation. This overhead is therefore not considered for assessing the benefit of the VM-Scheduler, i.e. no comparison to running SecApps natively is made.
2. A SecApp runs on top of an Operating System (OS), which itself uses resources. This overhead can be neglected when an OS is up and running. However, when booting or shutting down an OS, considerable resource usage is generated which does not contribute to the computation of SecApp results.
3. The manipulation of VMs causes resource usage. This overhead does not contribute to the computation of SecApp results.

In order to account for points 2 and 3, i.e. to assess the overhead, variables and metrics for the resource CPU[8] are introduced. Let $SI = 1s$ be the sampling interval. The duration of an experiment run is given by the interval $[0, kSI] = k\ seconds$.

$$maxCPU_{node} = \sum_{i=1}^{k} maxUsage_{CPU}(k, 1) \tag{6.1}$$

is the cumulated maximal CPU utilization on node for an experiment run. Accordingly

$$vmCPU_{node} = \sum_{i=1}^{k} \sum_{j=1}^{|VM|} currentProcUsage_{CPU}(k, 1, VM_j) \tag{6.2}$$

$$mainCPU_{node} = \sum_{i=1}^{k} currentProcUsage_{CPU}(k, 1, MainApp) \tag{6.3}$$

[7]For comparison, the resources UsedCore and AllocVM are fully under control of the framework.

[8]The CPU usage is a commonly accepted metric to evaluate the utilization of a cluster. However, the introduced metrics can also be applied in the same manner to other resources.

are the cumulated CPU resource utilization of VMs (6.2) and of the MainApp (6.3) on a node for an experiment run.

$$buffer_{node} := \frac{maxCPU_{node} \times (100 - threshold\ of\ P_{CPU})}{100} \tag{6.4}$$

This variable $buffer_{node}$ reflects the non-usable CPU share which is given by the threshold parameter of the dynamic policy P_{CPU}. The variable $results_{vm}$ represents the total number of results calculated by a single VM for an experiment run. The variable $maxresults$ is the number of results that could be computed by SecApps if 100% of CPU resources in the cluster were dedicated to VMs.[9] Based on these definitions, the resulting metrics are:

$$ResUsage = \frac{\sum_0^{nodes} vmCPU_{node}}{\sum_0^{nodes}(maxCPU_{node} - mainCPU_{node} - buffer_{node})} \tag{6.5}$$

$$VmConv = \frac{\sum_0^{vms} results_{vm} \times \sum_0^{nodes} maxCPU_{node}}{maxresults \times \sum_0^{nodes} vmCPU_{node}} \tag{6.6}$$

$ResUsage$ as defined in Equation 6.5 describes the ratio between resources which actually have been used by VMs and theoretically usable resources in the cluster. The theoretically usable resources are limited by the threshold parameter of the policy which is reflected in the buffer variable.[10] The metric $ResUsage$ therefore describes the loss in resource usage due to non-optimal scheduling of VMs and local adaption of resource usage by the Resource Usage Adaptor (RUA) (see Section 4.4.4.1). $VmConv$ as defined in Equation 6.6 describes the ratio between actually computed results by consuming resources and the theoretical amount of results that could have been computed by SecApps using the same amount of resources. This metric therefore reflects the amount of VM-utilized resources that were not translated to results but were wasted by OS-overhead and VM manipulations.

6.4.2. Experiment Layout

The experiment is carried out in the environment mentioned in Section 6.1, consisting of 24 worker nodes. Every node runs a specific synthetical CPU utilization pattern ("load pattern"). Every such node is also used by the VM-Scheduling Framework, i.e. a static policy of $P_{AllocVM} : (2, 25)$ is applied. Additionally, a dynamic policy P_{CPU} is applied with a specific configuration $(threshold, timelimit) \in \{95, 90, 80, 70\} \times \{20, 15, 10, 8, 6\}$. Multiple load patterns are run against specific policy configurations in this experiment and the resulting cluster usage and SecApp results are measured. Three experiment runs are performed for every experiment configuration ($load\ pattern \times policy\ configuration$). Such a run takes 20 min and starts by applying the pattern, starting the VM-Scheduler and the measurements. A number of leases is submitted and is taken care of by the GP which tries to provision these leases. After the 20 minutes have elapsed, the measurements are stopped and the previoulsy submitted leases are canceled by explicit request to the framework. The leases, VMs and SecApp used in this experiment are identical to the ones used in the "Policy Modification Experiment" (see Section 6.3.1).

[9] In this experiment, the same VM configuration and SecApp is used as in the "Policy Modification Experiment". Therefore, a single result is computed by a single VM and a maximum for computable SecApp results can be defined.

[10] There is a certain fuzziness introduced by the buffer variable. There is the possibility that CPU usage extends the policy threshold for a period defined by the timelimit. Knowing this fact, $ResUsage$ is considered a synthetic metric, not describing what could be used under the best circumstances, but rather what is usable in plausible cases.

CPU Usage Pattern Generation For testing the effect of a specific policy configuration $(threshold, timelimit) \in \{95, 90, 80, 70\} \times \{20, 15, 10, 8, 6\}$ towards the defined metrics, load patterns are generated. The goal of using different load patterns is to show that the framework copes with different load scenarios representing arbitrary MainApps. For producing specific patterns, a program was implemented that generates a node CPU usage with a precision of 0.5% (standard error) at a given point in time within tens of a second. The two main load patterns used in this experiment are shown in Figure 6.10.

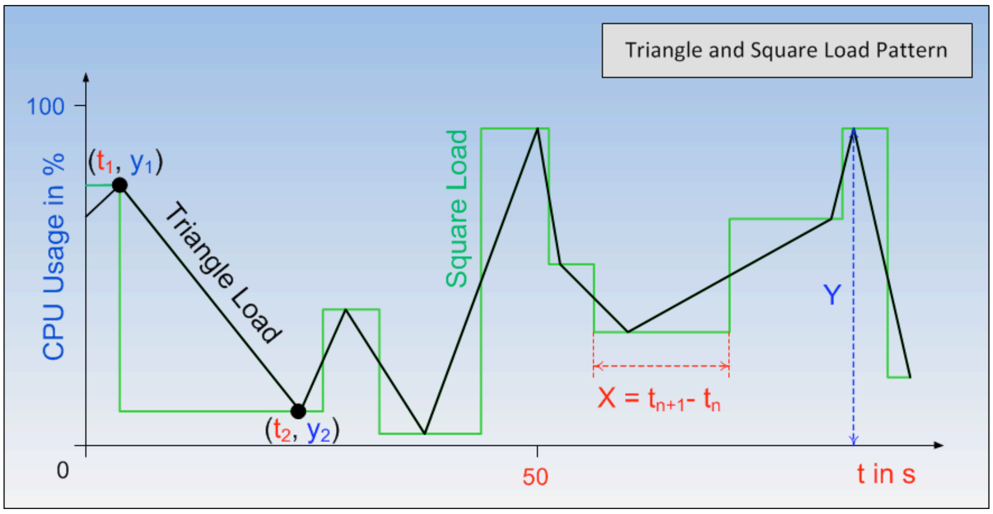


Figure 6.10.: Node CPU usage generation using the square (green-colored) and triangle (black-colored) pattern types. The CPU usage at a point in time t is a variate of the random variable Y and the frequency of usage changes is realized by the random variable X.

The load pattern drawn in green is called square load pattern. For this pattern the CPU usage on a node changes instantly. The pattern drawn in black is called triangle load pattern representing a steadily (and almost linearly) changing CPU usage. A load pattern is a tuple $LP := (Load, Type)$ with $Type \in \{square, triangle\}$, representing the type of pattern; and $Load$ with $5 \leq Load \leq 95$ representing the mean of the generated node CPU usage (in %) over an experiment run. So for a specific tuple $LP := (Load, Type)$ the same mean CPU usage is created for any node in the cluster. The actual development of the CPU usage, however, differs between nodes over an experiment run. For any such pattern, the frequency of load changes is randomly varied in an interval $[\frac{1}{5s}, \frac{1}{30s}]$ throughout an experiment run. More precisely, X is a random variable $X \in \{\triangle t \,|\, \triangle t = t_{n+1} - t_n \wedge \triangle t \in \mathbb{R} \wedge 5 \leq \triangle t \leq 30\}$, representing the difference between two points in time $(t_{n+1} - t_n)$ for which a target CPU usage is set. X follows a uniform distribution with variates in $[5, 30]$. The way the target CPU usage for t_{n+1} is reached from t_n is determined by the $Type$ parameter, i.e. either the CPU usage changes instantly or steadily.

The target CPU usage for a point in time t_n is defined as the variate of a continuos random variable $Y \in \{y \mid y \in \mathbb{R} \wedge 0 \leq y \leq 100\}$. The probability distribution for Y depends on the desired mean load for the specific load pattern. Figure 6.11 shows two different probability densities: the red function represents a distribution for random variable with expectation value $E[Y] = 50$, the blue function represents a distribution for a random variable with expectation value $E[Y] \approx 43$. So, a desired mean load for a particular load pattern over an experiment run (e.g. 43%) is achieved by realizing Y with

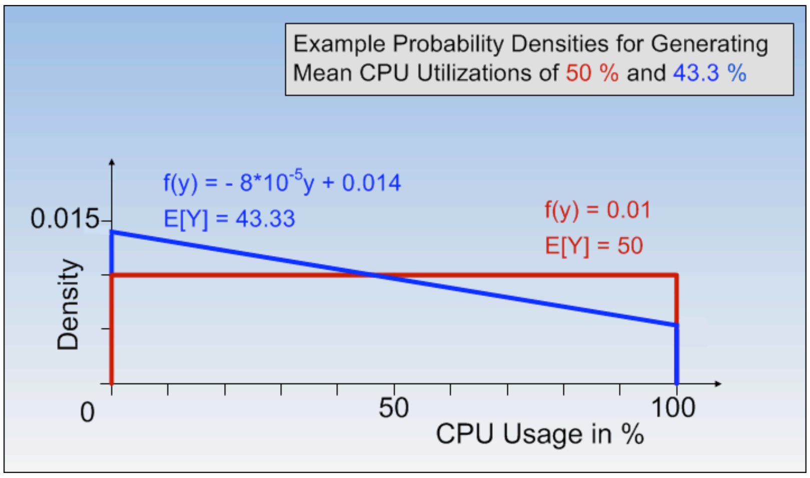


Figure 6.11.: Two example linear probability density functions $f(y)$ for a random variable Y. A repeated realization yields a mean CPU usage of 50% resp. 43.3%.

the according expectation value (e.g. $E[Y] \approx 43$). In order to cover the whole range of possible CPU usage values, the probability density is a linear function for any desired mean.[11] For a mean of 50% CPU usage it is exactly a uniform distribution, while for all other desired means it is a tilted uniform, i.e. a linear probability distribution calculated according to basic stochastical laws:

$$P(n \leq Y \leq m) = \int_n^m f(y)dy \text{ with } f(y) = ay + b \text{ and } E[Y] = \int_n^m y f(y)dy.$$

The experiment is conducted by running a specific policy configuration $(threshold, timelimit) \in \{95, 90, 80, 70\}\times\{20\} \cup \{90\}\times\{15, 10, 8, 6\}$ against a load pattern $(Load, Type) \in \{4, 8, 12, ..., 92, 96\}\times \{square, triangle\}$. This results in 368 runs in total with each lasting $20min$. However, the results show that the desired load means are not exactly met, which probably would require a longer experiment run duration. Nevertheless, the results are explicit enough to make arguments on how the framework copes with specific CPU utilization scenarios with respect to the defined metrics.

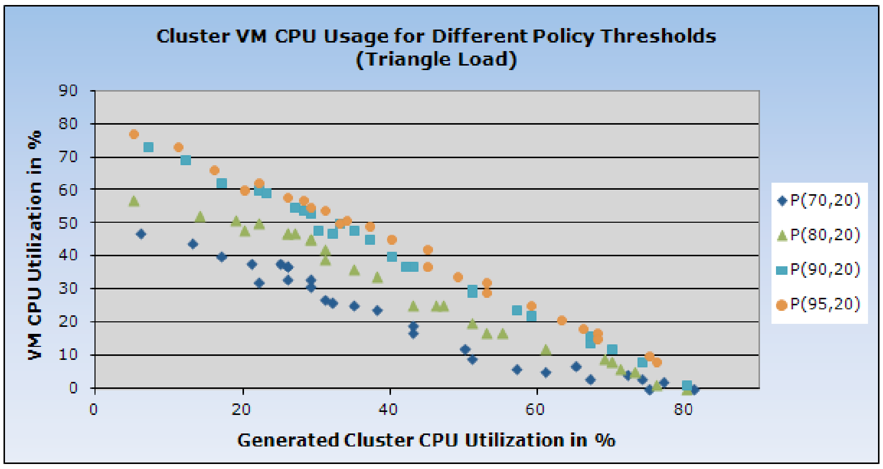
6.4.3. Results

For brevity reasons, the term "host load" is used to refer to the synthetically generated cluster CPU usage averaged over all nodes over an experiment run. The term "VM load" refers to the cluster CPU usage generated by VMs over an experiment run. First the impact of the threshold parameter is examined. Multiple run configurations with differing policy threshold (from 95% to 70%), but same timelimit ($20s$) have been set up and tested. The findings are now discussed. The timelimit parameter and its effect towards the efficiency metrics are examined in the second part of the section.

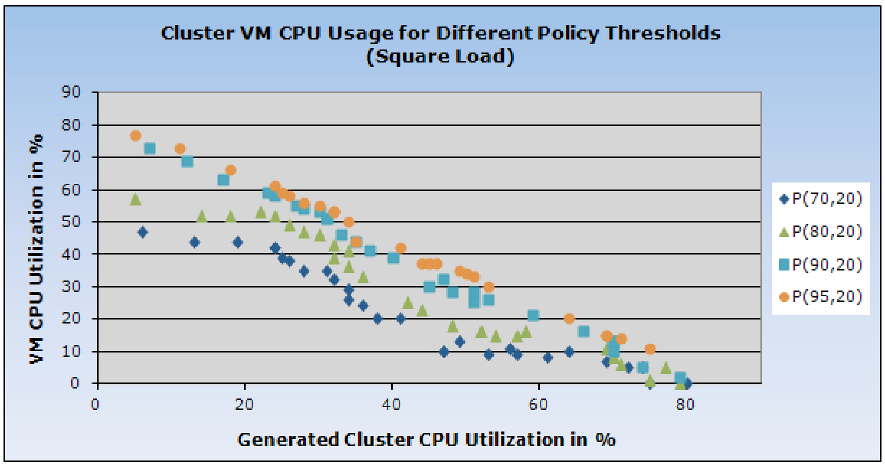
Threshold Parameter In Figure 6.12a the relation between host load and resulting VM load is shown for the triangle load pattern. Any measurement point represents a single experiment run. First,

[11] A normal distribution would favor variates close to the desired mean, potentially sparing occurrences of very low and very high load. On the other hand, a "bathtub curve" (weibull probability distribution) would do just the opposite, favoring high and low values at the expense of values in the middle of the domain. A linear density function is a reasonable compromise.

the amount of VM load clearly depends on the chosen policy threshold. This comes at no surprise since the threshold directly affects the resource share usable by VMs. Second, the generated host load has an recognizable effect towards the VM load. For thresholds of 95% and 90% this relation seems quite linear (with negative slope), i.e. the fewer load is generated on physical nodes, the more CPU resources are utilized by VMs. For thresholds of 80% and 70% the measurements follow the same basic pattern, but the curve looks somewhat shaky.[12] The same can be said for the square load pattern shown in Figure 6.12b . For any chosen threshold, the VM load increases with a decreasing host load.



(a) Triangle load pattern



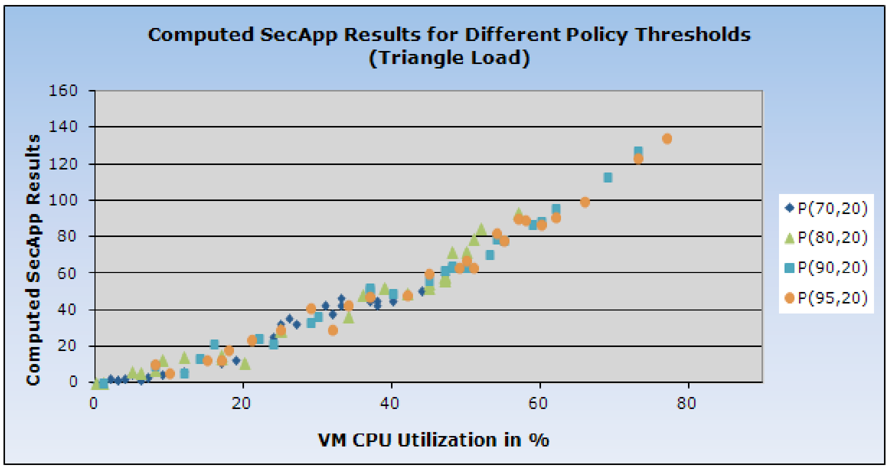
(b) Square load pattern

Figure 6.12.: Generated cluster CPU usage and resulting VM CPU usage for 20*min* experiment runs. Every dot represents a single run. Different policy thresholds are depicted in different colors.

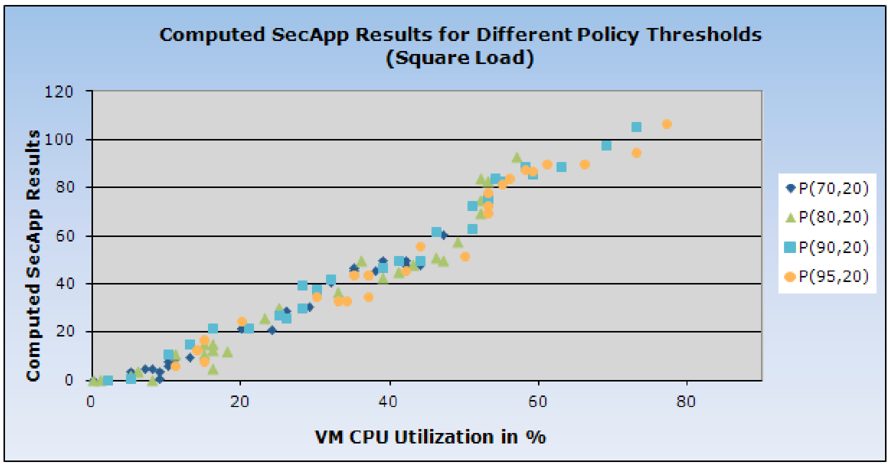
For the square load pattern, the shaky curves for thresholds of 70% and 80% are even more obvious. A possible explanation is that there are host load ranges (0%-20% and 40%-60%) for which the VM-Scheduler tries to start/resume two VMs (0%-20%) or one VM (40%-60%) on a node, but has to revert/modify these allocations due to repeated policy violations. This assumption is backed by the fact that for the range 40%-60%, the number of manipulations is 43% higher compared to the range

[12]The residuals for an applied linear regression are greater for 80% and 70% than for the 95% and 90% thresholds.

of 20%-40%.[13] The effect is more obvious for the square load pattern because the host load changes instantly, rather than gradually as for the triangle load pattern, and therefore is more likely to cause a policy violation. These shaky effects, i.e. the clear visual deviations from a linear functional relationship, may be regarded as a non-optimal configuration of the VM-Scheduler and can be avoided by improving the feasibility estimation and local resource usage adaption.
The following Figures 6.13a and 6.13b show the translation of VM load to computed SecApp results for the triangle and square load patterns. These figures therefore show the waste of CPU resources due to OS overhead and manipulations. At first glance, there is a monotonic, quite linear relation



(a) Triangle load pattern



(b) Square load pattern

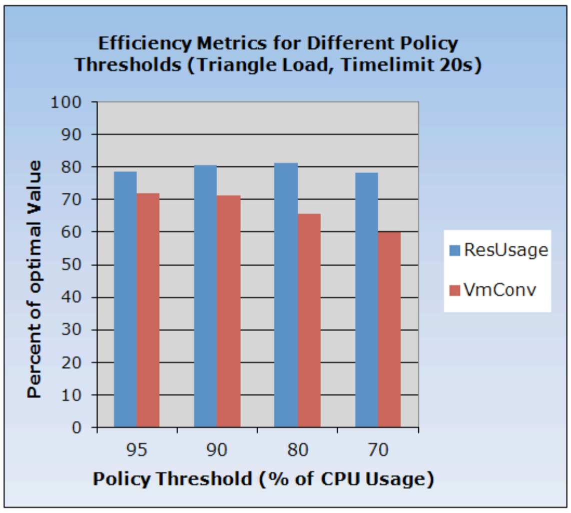
Figure 6.13.: VM CPU usage and resulting SecApp results for 20*min* experiment runs. Every dot represents a single run. Different policy thresholds are depicted in different colors.

between VM load and computed SecApp results for all thresholds for both load patterns. That is, the more VM load is generated, the more SecApp results are computed. Moreover, the VM load to SecApp results conversion, i.e. how many results can be computed with a given VM load, seems to be rather identical for all thresholds. This suggests that the conversion of VM generated CPU usage to benefitial SecApp results is independent from the chosen threshold. However, this can be only

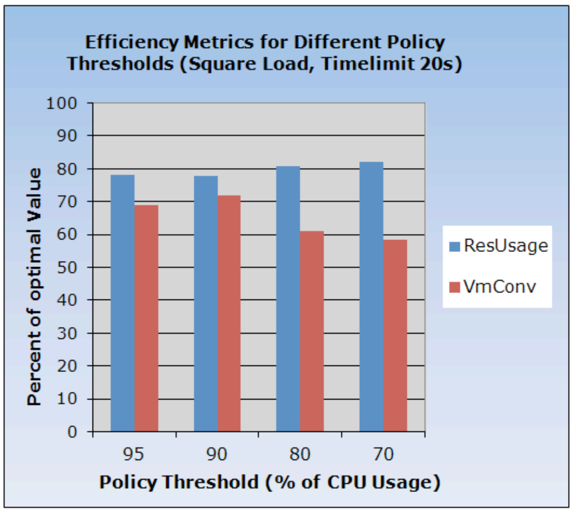
[13]Measured for the square load pattern and threshold 70%.

said for VM loads below 50%. Higher VM loads were not reached for, e.g., a policy threshold of 70%.

What could be seen so far by looking at the plots is that the threshold has an effect towards the VM load, but no visible effect towards the translation of VM load to SecApp results. There are, however, host load ranges which are better exploited by the VM-Scheduling Framework than others, which is likely to be caused by a non-optimal configuration of the VM-Scheduler. When factoring out the generated host load for the different thresholds, the picture becomes clearer. The efficiency metrics, calculated as explained in Section 6.4.1, are shown in Figures 6.14a and 6.14b.



(a) Triangle load pattern



(b) Square load pattern

Figure 6.14.: Efficiency metrics (see Section 6.4.1) for different policy thresholds.

For both load patterns, the VMs consume about 80 % of the usable CPU resources for every chosen threshold ($ResUsage$). While the threshold affects the absolute amount of VM load, there is no impact of the threshold towards the relative efficiency of exploiting usable resources. This observation is independent from the applied load pattern. Concerning the translation of VM load to results (*VmConv*), both load patterns also behave similiar: For thresholds of 90% and 95%, the efficiency lies at about 70% and declines for the other two thresholds to about 60%. This contradicts the tentative visual interpretation that the threshold has no impact on the translation of VM load to SecApp results. What is now the reason for this decrease at lower thresholds? Figure 6.15, which shows the average amount of manipulations executed for a certain threshold, clearly states that more manipulations have been conducted for lower thresholds.[14] Any manipulation and the OS overhead for booting a VM use CPU cycles. Therefore, an increased number of manipulations leads to a the lowered efficiency concerning VM load to SecApp results translation.

[14]These numbers have been obtained as an average over all experiment runs with square load pattern and timelimit 20 seconds. The averaged cluster CPU utilization was at about 47%.

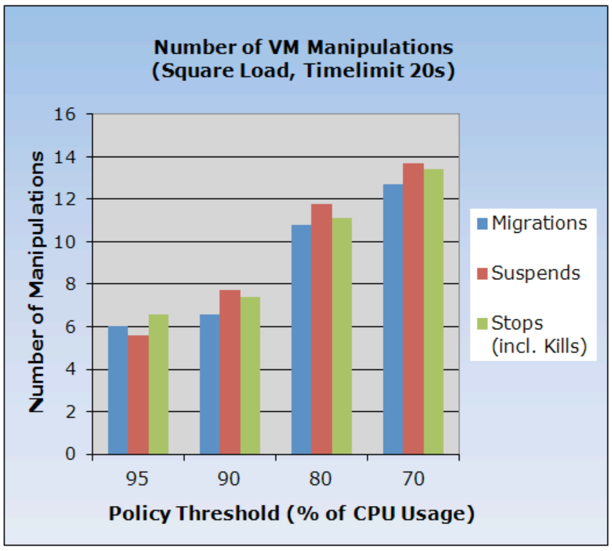
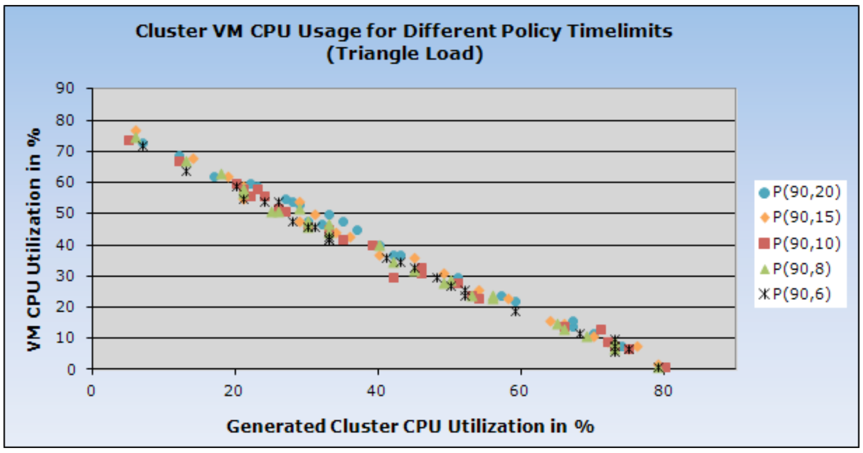


Figure 6.15.: Average number of VM manipulations for different policy thresholds with an applied square load pattern and timelimit of 20s.

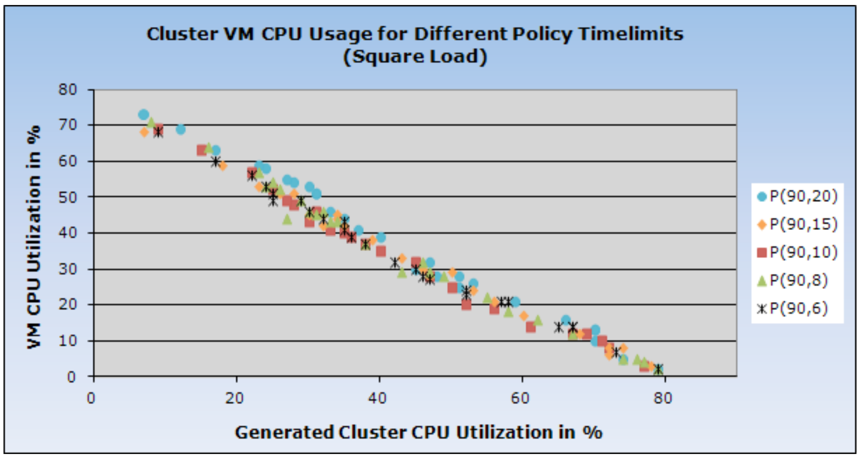
Summary for Threshold Parameter As a summary the following can be stated: There is an almost linear relation between host load and VM load for any threshold. The threshold parameter affects the total amount of resources used by VMs. The higher the threshold, the more CPU resources are used by VMs. The threshold does not have an impact towards the relative efficiency of exploiting usable CPU resources on nodes in the cluster: On average about 80% of usable CPU resources[15] are used by VMs. The parameter does affect the efficiency how VM-utilized CPU resources are translated to SecApp results.
While for higher thresholds (90%) about 70% of the optimally computable results were obtained this percentage drops to about 60% for lower thresholds (70%). The reason for this behavior is the increased number of manipulations for lower thresholds. The VM-Scheduler performs similiar for both tested load patterns.

[15]If the buffer given by the threshold is taken into account.

Timelimit Parameter Next the impact of the timelimit parameter is examined. Multiple policy configurations with a timelimit ∈ {20, 15, 10, 8, 6} and a threshold of 90% have been set up and tested. Figures 6.16a and 6.16b show the relation between host load and VM load for different timelimits. For both load patterns there is no obvious impact of the timelimit parameter on the CPU usage of



(a) Triangle load pattern

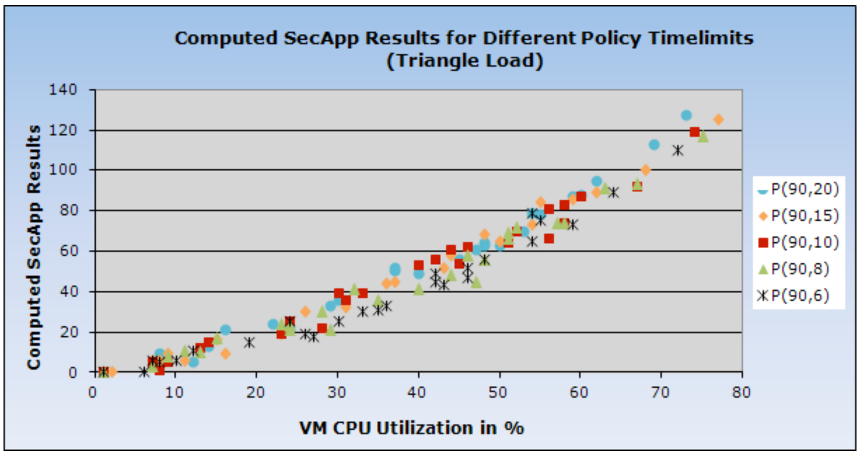


(b) Square load pattern

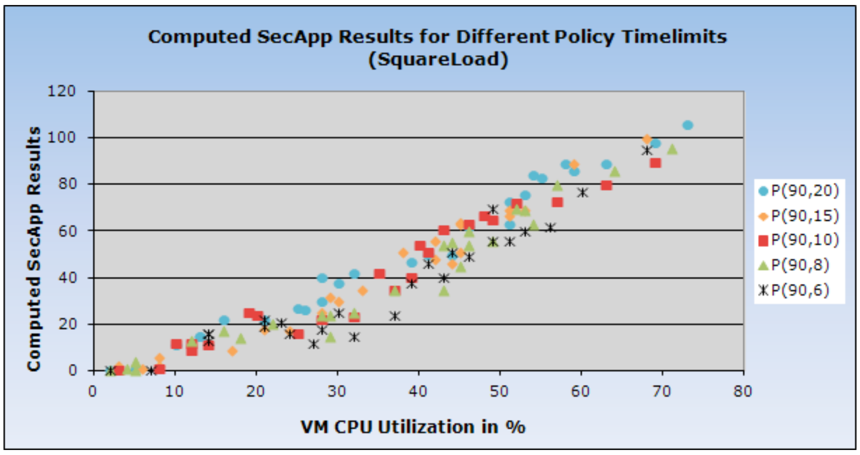
Figure 6.16.: Generated cluster CPU usage and resulting VM CPU usage for 20*min* experiment runs. Every dot represents a single run. Different policy timelimits are depicted in different colors.

VMs. Furthermore, there is a linear dependency with negative slope between generated host load and resulting VM load, i.e. the more host load is generated on nodes, the less CPU resources are used by VMs.

The picture is different for the translation of VM load to SecApp results. This effect is depicted in Figures 6.17a and 6.17b for the triangle and the square load pattern respectively. While at first glance both figures also show a similiar, quite linear dependency between VM load and SecApp results for all timelimits, a closer look reveals that *a*) lower timelimits seem to perform worse that higher timelimits and *b*) this effect is more obvious for the square load pattern. So how can this be explained?



(a) Triangle load pattern



(b) Square load pattern

Figure 6.17.: VM CPU usage and resulting SecApp results for 20*min* experiment runs. Every dot represents a single run. Different policy timelimits are depicted in different colors.

- Lower timelimits leave less time to the VM-Scheduler to make a decision and execute a VM manipulation. Stops are more likely to occur compared to higher timelimits, which decreases the amount of computable results. This effect is shown in Table 6.1, which shows the averaged number of manipulations for the square load pattern. For $timelimit = 20s$, 6.6 stops are performed, which accounts for about 36% of all manipulations performed in this configuration. For $timelimit = 6s$, 49.7 stops are performed, which corresponds to about 61% of all manipulations. This higher share of stops for lower timelimits leads to a decreased efficiency in translating VM load to computed SecApp results.

- According to the numbers given in Table 6.1, also the total number of manipulations rises with a decreasing timelimit. With an increasing number of manipulations, the OS and manipulation overhead leads to a decreased efficiency in translating VM load to computed SecApp results. The reason for the dependency between timelimit and number of manipulations is the "triggerSensitivity", which is discussed later in Section 6.4.3.

	VM Manipulations			Triggers Fired	
Timelimit (s)	Mig	Susp	Stop	Global	Local
6	12.5	18.8	49.7	23.9	17.4
8	13.9	11.3	33.8	25.3	7.7
10	15.4	12.7	22.3	29.5	4.3
15	7.2	8.2	15.9	17.1	2.1
20	6	5.6	6.6	12.9	0.25

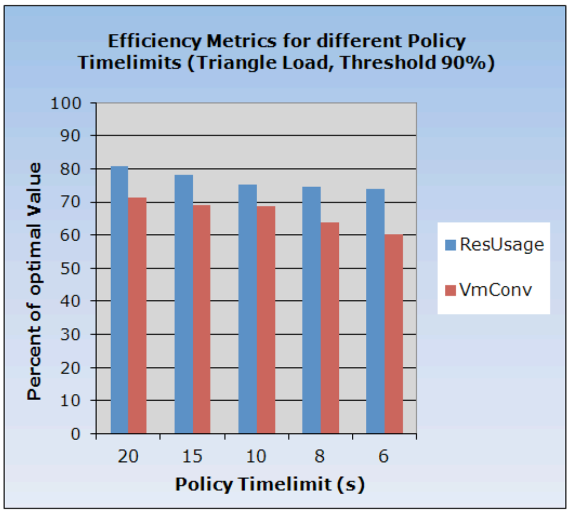
Table 6.1.: Number of VM manipulations and fired triggers averaged over all experiment runs for square load pattern and policy threshold of 90%.

- Another hint is given in Table 6.1: Not only the total amount of manipulations rises with decreasing timelimit, but also the number of locally triggered kills. While it seems natural that with more manipulations taking place also the amount of failed manipulations rises, the share of aborted manipulations is higher for *timelimit* = 6*s* (about 17%) than for *timelimit* = 20*s* (about 1%). This can be attributed to a bad feasibility estimation and leads to a decreased efficiency in translating VM load to computed SecApp results.
- The reason why this effect is more obvious for the square load pattern is that an instant and persistent host load change is more likely to cause a policy violation and therefore manipulations than the gradual host load change applied in the triangular load pattern. This is due to the global trigger mechanism and clearly backed by data given in Table 6.2. So, more manipulations for the square load pattern decrease the efficiency of translating VM load to computed SecApp results.

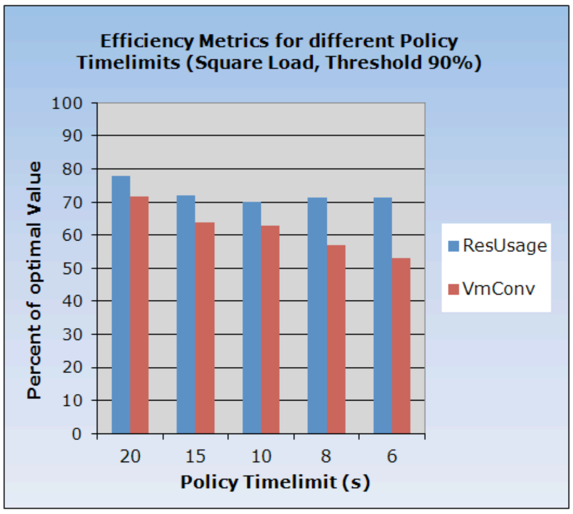
	VM Manipulations	
Timelimit (s)	Square Load	Triangle Load
6	81	40.9
8	59	31.8
10	50.4	26.2
15	31.3	15.1
20	18.2	9.6

Table 6.2.: Number of VM manipulations averaged over all experiment runs for policy threshold of 90%.

The shown plots indicate that there is no obvious impact of the timelimit towards the usage of free CPU resources, but a certain, not yet quantified effect towards the number of computed SecApp results. The efficiency metrics shown in Figures 6.18a and 6.18b make these effects more explicit. For the efficiency of exploiting usable CPU resources ($ResUsage$), the timelimit of 20 seconds performs best for both load patterns; about 80% of usable CPU resources are exploited by VMs. There is a slight decline for other timelimits, which all still reach at least 70% efficiency. So there is a certain, yet limited impact of the timelimit parameter towards the efficiency of exploiting usable resources. Concerning the translation of VM load to computed SecApp results (*VmConv*), there is a considerable decline from 71% for *timelimit* = 20*s* to 52% for *timelimit* = 6*s* for the square load pattern. For the triangle load pattern this efficiency drops from 71% for *timelimit* = 20*s* to 60% for *timelimit* = 6*s*. This decline reflects the relevant impact of the timelimit parameter towards the efficiency of translating VM load to SecApp results. Comparing this impact for the two load patterns shows that for any timelimit this efficiency is worse for the square load pattern which is due to the higher number of manipulations for this load pattern.



(a) Triangle load pattern



(b) Square load pattern

Figure 6.18.: Efficiency metrics (see Section 6.4.1) for different policy timelimits.

Summary for Timelimit Parameter As a summary it can be stated that the timelimit parameter has a minor impact towards the efficiency of exploiting usable resources. The efficiency is at about 80% for *timelimit* = 20*s* and declines to about 70% for *timelimit* = 6*s*. This holds for both tested load patterns. The timelimit parameter does, however, have a considerable impact on the efficiency of translating VM load to computed SecApp results. Here, the usage of low timelimits has an adverse effect on the number of computed SecApp results. This adverse effect exists for both load patterns but is more evident for the square load pattern.

Job Length Concerning the declining *VmConv* metric for lower timelimits, one additional aspect needs to be considered. So far, these metrics have been calculated using the assumption that the duration for computing a single result by a VM ("job length") is 2.5*min*. There are, however, applications (e.g. ALICE Offline batch jobs - AliEn [10]) that may take up to an hour to complete a result computation. For such a job length, the loss induced by stopping a running computation clearly is weighing in heavier, with an adverse effect towards the efficiency measure *VmConv*. A recalculation of this efficiency for an example job length of 5*min*, i.e. two successive, completed jobs count as one, demonstrates this effect: For the square load pattern, a timelimit of 6 seconds and a job length of 2.5*min*, a *VmConv* efficiency of 52% could be achieved. By doubling the job length this efficiency drops to 40%. The decline in efficiency for lower timelimits is relevant and becomes more important the longer an average result computation takes. This effect of course also applies to the efficiency metrics calculated for the threshold experiment.

Trigger Sensitivity As shown before (see Table 6.2), there is a dependency between the number of manipulations and the timelimit parameter. This parameter reflects the time span for which the resource usage is allowed to exceed the threshold with a VM using this resource. The earlier a global trigger fires, the more time is left for decision and execution phases of the GR. This means that more time-consuming but gentler manipulations can be chosen. However, not every exceedance of the threshold automatically leads, if not handled by GR and Local Trigger Unit, to an invalidation of the respective policy. Both the resource usage variation caused by MainApp and the RUA may cause the total resource usage to drop below threshold again. This means that an early global triggering, i.e. a high global triggerSensitivity[16], allows for gentler manipulations, but also increases the occurrence probability of globally fired triggers, thereby increasing the total number of manipulations. Late triggering, i.e. a low global triggerSensitivity, decreases the occurrence probability of globally fired triggers and leads to fewer manipulations. However, late triggering leaves fewer time for decision and execution phases of the GR, thereby leading to less gentle manipulations. So the choice of the triggerSensitivity is a trade-off. However, it is not reasonable to choose the same triggerSensitivity for all timelimit configurations. For any timelimit, the triggerSensitivity should be chosen such that upon triggering there is still enough time left to execute a set of manipulations. For a low timelimit of 6 seconds, the global trigger therefore must fire earlier than for a timelimit of 20 seconds. This is the reason why lower timelimits cause more manipulations. In Figure 6.19 the effects of different

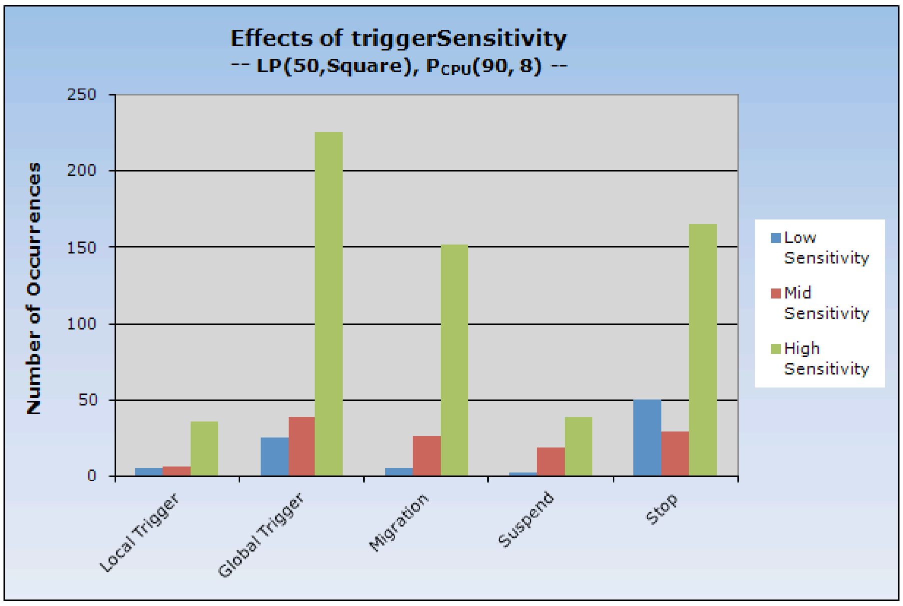


Figure 6.19.: Effect of triggerSensitivity (see Table 6.3) on VM manipulations and triggering. The effect is shown for a specific setup $LP(45, square)$ and $P_{CPU}(90, 8)$.

triggerSensitivity levels are shown for a specific experiment run.[17] The triggerSensitivity levels (Low, Mid, High) refer to the ones defined in Table 6.3. For a low sensitivity, few global triggers have fired (29) and most manipulations were stops (5 migrations, 2 suspends, 50 stops). Quite the opposite can be seen for the high sensitivity. Here, a large number of global triggers fired (226) and many manipula-

[16]As a reminder, the triggerSensitivity was defined in Section 4.4.3.1 by the number of subsequent, failed model-checks for a policy. A high number corresponds to a low triggerSensitivity.

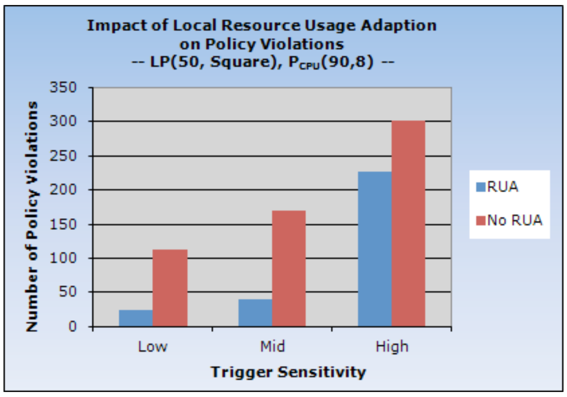
[17]Run configuration: $20min$, P_{CPU} : $(90, 8)$, Load Pattern: (square, 50).

tions (152 migrations, 39 suspends, 165 stops) were executed. The excessive amount of manipulations led to bottlenecks not sufficiently predicted by the Feasibility Estimator (FE) and consequently to a considerable amount of locally triggered kills (36). The reasonable compromise therefore is to make a trade-off, i.e. to use the medium sensitivity as given in Table 6.3. In the experiment this medium sensitivity allowed for both having less manipulations than with high triggerSensitivity, but also for choosing not only stop manipulations (26 migrations, 19 suspends, 29 stops). This contributes to a better SecApp result computation. This medium sensitivity was used in all experiments discussed so far.

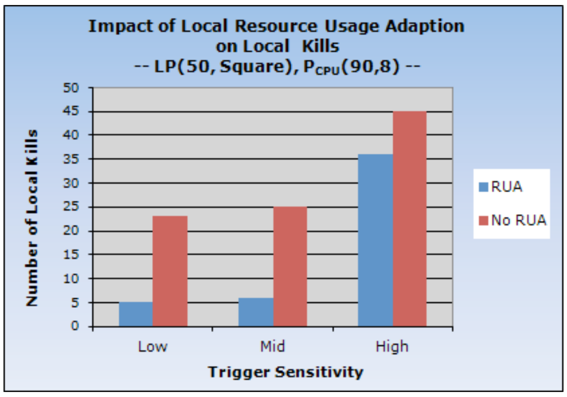
	Timelimit (s)					
Sensitivity	25	20	15	10	8	6
Low	6	5	4	3	2	1
Mid	7	6	5	4	3	2
High	8	7	6	5	4	3

Table 6.3.: TriggerSensitivity configurations (in s) for different timelimits.

Local Resource Usage Adaption In conceptual work, Section 4.4.4, it has been argued that a subsidiary framework design, i.e. to handle resource usage generated by SecApps locally as far as possible, helps to relieve the GR from making global changes of the allocation pattern. If resource utilization can be adapted locally, less globally triggered policy violations occur, thereby decreasing the computational effort needed by GR and reducing the number of executed manipulations. According to the proposed GR algorithm, this in general will lead to the choice of gentler manipulations and less locally triggered kills, which in turn increases the efficiency of computing SecApp results. An experiment has been conducted where the framework ran with disabled resource usage adaption.



(a) Policy violations



(b) Local kills

Figure 6.20.: Effect of local resource usage adaption by RUA on the number of locally triggered kills and globally triggered policy violations. The effect is shown for a specific setup $LP(45, square)$ and $P_{CPU}(90, 8)$.

Figure 6.20a shows the number of globally triggered policy violations depending on the applied triggerSensitivity for both enabled and disabled resource usage adaption for a specific experiment run. It can be seen that a local RUA decreases the amount of policy violations the GR has to handle, which in turn decreases the number of globally commanded manipulations. Figure 6.20b shows the effect of these setups for locally triggered kills of VMs and/or aborted VM manipulations. Again, it is clearly visible that an enabled local resource usage adaption decreases the number of locally triggered kills. The conceptual decision to have a locally acting functional unit (RUA) is thereby confirmed to be appropriate for the purpose of decreasing the number of globally handled policy violations.

6.4.4. Summary

In this experiment it has been examined how the VM-Scheduling Framework copes with a changing usage of resources over which the framework does not have full control.[18] The CPU was chosen as a prototypical resource whose usage also depends on the MainApp. It was evaluated what impact a policy configuration, i.e. settings for threshold and timelimit parameters, has towards the goals of increased cluster utilization and generated SecApp result output. In order to represent a wide range of potential MainApps, synthetic CPU usage patterns were generated and applied to a test environment. It could be shown that the amount of resources which were exploited by VMs has an almost linear relation (with negative slope) to the synthetically generated CPU usage. This property of linearity could be observed independently from the chosen threshold and timelimit.[19] Also, it was shown that a higher threshold corresponds to higher total resource usage by VMs. When factoring out the buffer, i.e. the amount of non-usable resources determined by the threshold, the share of exploitable resources used by VMs was between 70% and 80% for any chosen timelimit and threshold. Concerning the translation of exploited resources to actually computed SecApp results, the threshold parameter has a minor impact. While for higher thresholds (90%) about 70% of the optimally computable results were obtained, for lower thresholds (70%) this percentage drops to about 60%. The timelimit parameter also has an impact towards the translation of exploited resources to computed results. There is a decline from a relative efficiency of 71% for the timelimit of 20 seconds to 52% for the timelimit of 6 seconds. For both parameters threshold and timelimit, this decline depends on the average SecApp result computation time. The larger the computation time, the less efficient is the translation of used resources to actually computed results.
These findings suggest that using higher timelimits and higher thresholds will yield better results with respect to the thesis goals of increasing the cluster usage and generating additional result output. To see the full picture, it is now necessary to evaluate the effect of these parameters towards the amount of interference caused for a MainApp.

6.5. Interference Experiment

To evaluate the amount of interference caused by running SecApps in a cluster it is not sufficient to run a synthetic CPU utilization pattern that represents a MainApp. Instead, an application needs to be run which has a measurable performance metric. Using this metric the amount of interference caused by additionally run SecApps can be assessed. The MainApp used in this experiment not only generates a substantial amount of CPU usage but also considerable network traffic. This requires the usage of dynamic policies for the resources CPU and NetOut. Again it is evaluated what effect the

[18]For comparison, the framework has full control over the resource AllocVM.
[19]This statement has not been proven mathematically, but was derived visually. Also, only the property of linearity, not the exact equation and the residuals of a regression are independent of the policy parameters.

policy configuration, i.e. the parameters timelimit and threshold have towards the amount of generated interference. More specifically, the following questions are targeted in this experiment:

- Are there policy configurations for which no interference[20] exists?
- If interference can be avoided, will still additional cluster usage and SecApp results be generated?
- Can the amount of interference be tuned by adapting timelimit and threshold parameter of policies?

6.5.1. Experiment Layout

The experiment is executed in the cluster environment used for the previous experiments (see Section 6.1). Each of the 24 worker nodes is in the scope of the VM-Scheduling Framework, i.e. a static policy of $P_{AllocVM} : (2, 25)$ is associated with every node. Additionally, a dynamic policy P_{CPU}, which operates on the CPU resource, and another dynamic policy P_{NetOut}, which targets the outgoing network traffic, are associated with each node. Different configurations for these policies are applied in this experiment. The leases and VMs used in this experiment are identical to the ones used in the "Policy Modification Experiment" (see Section 6.3.1).

MainApp Design The MainApp used in this experiment consists of two independent tasks that run on each node, one that is CPU-bound and another one that is network-bound. The CPU-bound task is a GCC compiler benchmark running in a closed loop. The number of completed loops having compiled a specific amount of source code, summed over all nodes for a run, is used as the performance metric for this task. The metric is referred to as "CPUResults". The network-bound task is the copying of data from experiment worker nodes to target nodes which are not in the scope of the VM-Scheduling Framework.[21] The copying is realized by sending data from /dev/zero on a worker node to /dev/zero on the target host using out-of-the-box Linux tools "dd" and "netcat".[22] The amount of copied data, summed over all worker nodes for a run, is used as the performance metric for this task. The metric is referred to as "NetResults". So both tasks together are the MainApp. This MainApp has two separate performance metrics, CPUResults and NetResults.

In order to avoid a permanent full saturation of the CPU and NetOut resources (i.e. $Usage_{res}(k, k) = 100$) on the nodes, which would prevent any SecApps from running in the cluster, the following is done: The GCC compiler benchmark uses only one of the two available CPU cores. The copying task is configured such that the data rate, i.e. the usage of NetOut, varies over time. This also causes a variation of the CPU usage caused by the copying task. For each node the applied variation pattern is different. However, averaged over the duration of an experiment run, every node has a NetOut usage of about 50% and has copied the same amount of data.[23]. Using this scheme, both the CPU utilization and the NetOut utilization on every node in the cluster vary over time without inter-node correlation.

[20] Interference was defined as a statistically significant difference between performance metric values.

[21] The target nodes have not been mentioned yet in the experiment layout and are chosen such that all worker nodes can simultaneously saturate their outgoing network interfaces. Every copying worker node has its own target node and it is ensured that no contention occurs at switches.

[22] By using this method the disk resources on the nodes are not involved which otherwise could be a bottleneck limiting the data-rate.

[23] The variation of the NetOut utilization is realized using the method described in Section 6.4.2. Actually a NetOut load pattern $LP := (50, square)$ is applied. The Linux tool "pv" is used to cap the network utilization.

Nevertheless, the values obtained for the performance metrics are constant[24] for any experiment run without SecApps.

SecApp Design Any VM used in the experiment hosts a GCC compiler benchmark as described for the previous experiments (see Section 6.3.1). Additionally, each VM runs a network-stressing task, which is implemented using the same method (dd/netcat) like the one described for the MainApp. However, no variation of the data-rate is applied but any running VM pushes data over the network as fast as it can.[25] The metric used for assessing SecApp result computations is the number of completed compiler loops. This metric is referred to as SecApp results.

The experiment compares the metrics CPUResults and NetResults for experiment runs without running SecApps ("No-VM runs") with those where additional SecApps were run ("With-VM runs"). An experiment run takes 7 minutes. At first 8 No-VM runs are conducted. CPU utilization, CPUResults and NetResults are recorded as a baseline to be compared against With-VM runs. For each policy configuration $(threshold, timelimit) \in \{95, 90, 85, 80, 75\} \times \{25, 15, 8\}$ for the two policies P_{CPU} and P_{NetOut} three With-VM runs are performed, resulting in a total of 45 With-VM runs. The leases are submitted at the beginning of a With-VM run. For every such With-VM run, the CPU usage of the nodes, CPUResults, NetResults and computed SecApp results are recorded.

6.5.2. Results

As baseline for evaluating the impact of SecApps towards the MainApp, the statistics gathered for the No-VM runs are used. The mean and standard deviation of CPUResults and NetResults for 8 runs are shown in Table 6.4. These values translate to an average cluster CPU utilization of 54% and an average NetOut utilization of about 52%, i.e. 522 MBit/s per node, for a No-VM run.

	Mean	Standard Deviation	Resource Usage (%)
NetResults	666129 MB	1393 MB	52
CPUResults	344.25	4.1	54

Table 6.4.: Mean and standard deviation of performance metrics and resulting cluster resource usage for No-VM runs

Since CPUResults and NetResults for a single No-VM run are aggregated values over all nodes in the cluster, a normal distribution for these values can be assumed according to the central limit theorem. The specific normal distributions for CPUResults and NetResults are representatives for the performance of a non-interferred MainApp. To evaluate the interference caused by SecApps, the respective values gathered for With-VM runs need to be tested against these distributions. This test is done in two different ways: First, it is evaluated whether there is a statistically significant deviation for With-VM runs with a specific policy configuration. Second, it is evaluated whether there is a dependency between the amount of interference and policy parameters settings. A null hypothesis is formulated to test for statistically significant differences between the performance metric values

[24]Constant with a certain standard deviation when considering multiple runs. This deviation is provided later on in Table 6.4.

[25]The virtualization product used in all experiments (VirtualBox 3.14) showed a practical limitation of about $250 MBit/s$ throughput per ethernet interface. This was the network throughput created by a single VM in this experiment.

obtained for No-VM runs and With-VM runs :

$$\begin{aligned} H_0 &: \mu_0 = \mu_1 \\ H_1 &: \mu_0 \neq \mu_1 \end{aligned} \tag{6.7}$$

where H_0 represents the assumption that no statistically significant difference for a metric (CPUResults, NetResults) exists between With-VM and No-VM runs for a specific policy configuration. H_1 therefore represents the opposite assumption. This hypothesis H_0 now has to be tested for every policy configuration to find out whether there is a statistically significant interference with the MainApp. A significance niveau of 5% is chosen according to common scientifc practice.

An important aspect needs to be discussed before proceeding: By testing for H_0 a conclusion like "there is no interference for this policy configuration" cannot be drawn. By rejecting H_0 it can only be stated that there is interference, though it is not possible to prove that there is no interference for a policy configuration. This problem of hypothesis design is known [13, 34] and non-inferiority and equivalence trials were proposed to circumvent this issue. These, however, require an assumption on what is considered an acceptable difference of the metric means for the control groups. For the MainApp in this experiment no such assumption can be made. Therefore, the proposed hypothesis design does not provide statistical proof, but nevertheless a plausible hint on the existence of interference. It will not be shown that there is no interference for a policy configuration, but rather:
Is there enough statistical evidence to assume that there is interference for a specific policy configuration? If so, then the existence of interference is shown. If not, then the hypothesis ("there is no interference") could not be rejected using this statistical test. The retaining of H_0 then translastes to: The probability that the values found for With-VM runs also could have been obtained for No-VM runs is too high to reject H_0.

For testing the significance a "two-sample unpooled t-test for unequal variances" is used. Tables 6.5a and 6.5b show the results (t-values) for these significance tests referring to CPUResults and NetResults respectively. The values highlighted with grey color represent a retained H_0, the other values stand for a rejected H_0, i.e. an accepted H_1.

	Timelimit (s)		
Threshold (%)	8	15	25
75	47.28	20.9	9.28
80	4.4	0.74	0.2
85	1.8	0.1	0.01
90	0.01	0.01	0.01
95	0.01	0.01	0.01

(a) t-Values for CPUResults

	Timelimit (s)		
Threshold (%)	8	15	25
75	46.78	94.06	20.36
80	2.52	21.62	17.1
85	2.08	5.62	2.56
90	1.94	3.14	3.06
95	1.24	1.42	1.4

(b) t-Values for NetResults

Table 6.5.: Two-sided t-test values for different policy configurations and performance metrics.

For CPUResults given in Table 6.5a this means: For every policy configuration P_{CPU} except (75,8), (75,15) and (75,25) interference is shown. For (75,8), (75,15) and (75,25) the existence of interference could not be proven. For NetResults given in Table 6.5b this means: For every policy configuration P_{NetOut} except (75,8), (75,15), (75,25), (80,15), (80/25) and (85,15) interference is shown. For (75,8), (75,15), (75,25), (80,15), (80/25) and (85,15) the existence of interference could not be proven.

This leads to the interpretation: There are policy configurations for which the MainApp is interferred. There are other policy configurations (threshold of 75% for both P_{CPU} and P_{NetOut}) where no interference could be found. The latter can be regarded, despite of the hypothesis design aspect discussed above, as a strong hint that there are policy configurations ((75,8), (75,15), (75,25)) which do not lead to interference. The values for which H_0 could not be rejected are those where fewer resource shares were given to SecApps.
This observation brings up the question whether there is a dependency between the policy parameter settings and the amount of interference caused. The values for the amount of interference are given in Table 6.6 for the metrics CPUResults and NetResults. These values clearly state that for any given timelimit, the amount of interference strictly increases with the policy threshold.[26] This monotonic relation applies to both CPUResults and NetResults. So, choosing a lower threshold for a policy will decrease the amount of interference. This makes the threshold parameter an appropriate configuration item to control interference. It is clear that choosing a too low threshold might lead to a situation

	Timelimit (s)		
Threshold (%)	8	15	25
75	2.21	4.46	6.75
80	8.33	14.71	18.67
85	21.5	24.38	41.25
90	34.25	42.71	43.88
95	48.21	50.08	51.59

(a) CPUResults

	Timelimit (s)		
Threshold (%)	8	15	25
75	1060	27	1701
80	5205	2136	2257
85	9172	4600	6209
90	9901	8180	9508
95	14411	12425	14125

(b) NetResults (in MB)

Table 6.6.: Amount of interference for different policy configurations.

where no VMs are run at all. So one could suspect that for the policy configurations where no interference was observed ((75,8), (75,15), (75,25)) this could have happened. This, however, is not the case. In Figure 6.21 it can be seen that additional CPU utilization (7%) and SecApp results (24) were generated for the policy configuration (75,15). But this figure also clearly shows that using a higher threshold enhances the benefits of the VM-Scheduler, i.e. additional cluster utilization and generated SecApp results, considerably. When using the VM-Scheduler, there is a trade-off to be made between the amount of interference tolerated and benefit from VM-Scheduling expected. This trade-off is realized by choosing approriate policy configurations and should be geared towards avoiding strong interference, i.e. to keep a MainApp functional according to pre-defined Service-Level Targets (SLT) metrics.
For the timelimit parameter, such a clear conclusion ("interference and VM benefit scale with threshold") can not be drawn. As shown in Table 6.6, for metric CPUResults the amount of interference strictly increases with the chosen timelimit. Nevertheless, for NetResults this statement does not hold. For timelimits of 15 and 25 seconds the amount of interference also increases with the timelimit. The timelimit of 8 seconds stands out, however. It generates, except for threshold 75%, the highest amount of interference. A possible explanation is that for low timelimits there are more manipulations than for higher timelimits due to triggerSensitivity. Such manipulations like migration and suspend, but also successive starts and resumes carry considerable overhead towards network usage, which may be the cause for the high amount of interference for the timelimit of 8 seconds.

[26] A statistical test for significance or even curve fitting is omitted at this place. The observations are used only to demonstrate some basic insight into the effect of parameters.

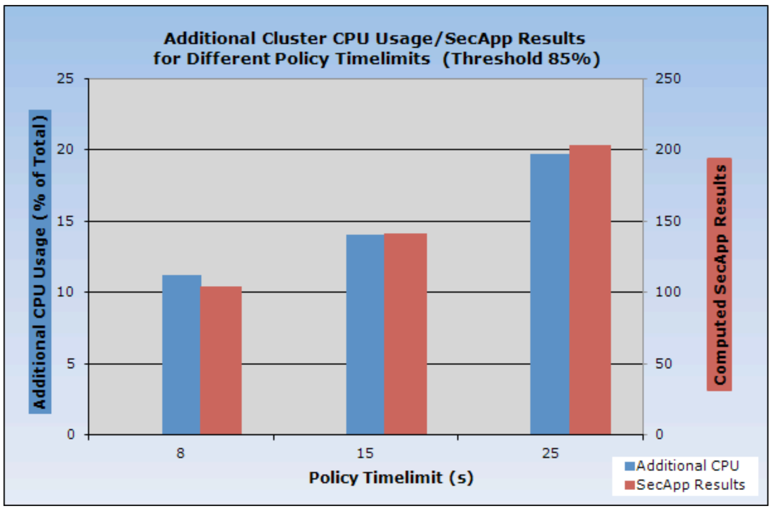


Figure 6.21.: Cluster CPU exploited by VMs and accordingly computed SecApp results for different policy thresholds.

Three conclusions can be drawn from the observations concerning the timelimit parameter:

- The impact of network manipulations has been underestimated and should be re-weighted in the Feasibility Estimator (FE). This should result in prefering stops over e.g. suspends of VMs more often.
- For the experiment layout, this means that more timelimit configurations (e.g. 10, 20, 40 seconds) should have been tested to provide more factual data to investigate. Identical configurations for P_{CPU} and P_{NetOut} had been chosen in the current experiment layout. Differing configurations for these (e.g. a lower threshold for P_{NetOut} than for P_{CPU}) could yield interesting results.
- The amount of interference increases with the timelimit for the CPUResults metric. For the metric NetResults, choosing a lower timelimit (15 seconds) yields a lower amount of interference than a higher timelimit (25 seconds). However, choosing a too low timelimit (8 seconds) comes with a considerable penalty towards the amount of interference concerning NetResults.

The effect of the timelimit parameter towards the benefit of the VM-Scheduler, i.e. computed SecApp results and cluster utilization, is exemplified in Figure 6.22, which depicts these values retrieved for a threshold of 85%. The number of SecApp results increases with rising timelimits. The same relation holds for the cluster utilization. Regarding the feasibility of the timelimit parameter as a configuration item to make a trade-off between interference and VM-Scheduler benefit, the following can be said: The higher the timelimit, the more SecApp results and additional cluster utilization is generated. For CPU-bound tasks a higher timelimit will also lead to a higher amount of interference. For network-bound tasks such a relation could not be shown. So, for CPU-bound MainApps the timelimit parameter can be used to make a trade-off between amount of interference and benefit of the VM-Scheduler such that no strong interference occurs. For network-bound tasks the timelimit parameter is not an appropriate configuration item for making this trade-off.

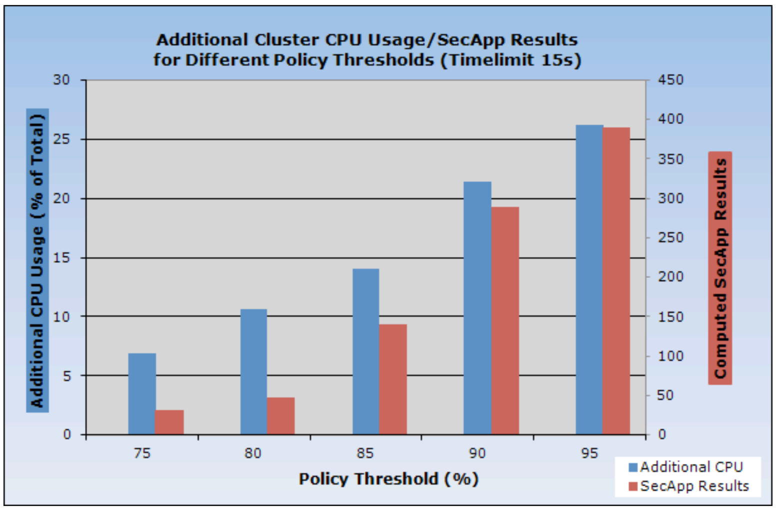


Figure 6.22.: Cluster CPU exploited by VMs and accordingly computed SecApp results for different policy timelimits.

6.5.3. Summary

It could be shown that there are policy configurations for which no (statistically significant) interference can be observed, which increase the cluster utilization and lead to the generation of SecApp results. The effect of the policy parameters timelimit and threshold towards the amount of interference was examined. A monotonic relation between the threshold value and the amount of interference was observed. This makes the policy threshold an appropriate configuration item to control interference and avoid strong interference for a MainApp. Results for the timelimit parameter are indifferent. Concerning the amount of interference generated towards a CPU-bound MainApp, there is a monotonic relation between timelimit and amount of interference. No such relation was found for the amount of interference towards a network-bound application. Based on these findings, it can be stated that interference can, in a controlled manner, be modified by the threshold parameter. The timelimit parameter can be used to control interference if the MainApp is not network-bound. More experiments and presumably a better feasibility estimation are needed to assess the effect of the timelimit parameter towards the interference of a network-bound MainApp.

6.6. HLT-Chain Experiment

In the previous experiments the chosen approach was evaluated for different host load patterns (see Section 6.4) and a specific, yet constructed MainApp (see Section 6.5). Such in-situ experiments, as good as they may be planned, can provide arguments but no certainty. The VM-Scheduler therefore has to be tested against a MainApp, whose correctness of functionality is specified using a performance metric. Only such a specification allows to make a statement on whether there are benefits in using the VM-Scheduler with a specific MainApp while at the same time preventing strong interference. In this experiment, the VM-Scheduler is evaluated against the HLT-Chain application.

The goal of this experiment is to show that:

1. SecApps can run in parallel to the HLT-Chain without causing strong interference.
2. The average cluster CPU usage thereby is improved.
3. Additional results can be computed by SecApps run inside of VMs.

In a first sub-experiment both the HLT-Chain and the VM-Scheduler are running in parallel for a certain time and appropriate metrics are evaluated with respect to the stated goals.
A second sub-experiment is made to assess whether a more demanding scenario with sudden cluster utilization changes can also be coped with by the VM-Scheduler. A typical case is the starting and stopping of the HLT-Chain, something that happens multiple times a day at A Large Ion Collider Experiment (ALICE) and should not be affected by running SecApps. Another typical scenario is the starting and stopping of the VM-Scheduler at runtime of the HLT-Chain. To evaluate the impact of such cluster-wide changes, a sequence of starting and stopping both the VM-Scheduler and the HLT-Chain is is tested in the second sub-experiment.

6.6.1. HLT-Chain Application

A reminder about the properties of the HLT-Chain is needed before taking a look at the sub-experiment layout. The HLT-Chain application is soft real-time application used for processing physics data gathered at the ALICE experiment in CERN. The HLT-Chain has a distributed, hierarchical structure. The structure makes up of components that are placed on physical nodes and communicate with each other. Data is inserted at components representing the leafs of the hierarchy and is subsequently processed and propagated throughout the hierarchy until fully processed and stored by Data Acquisition (DAQ). The hierarchy and several processing paths in distinct colors are shown in Figure 6.23.

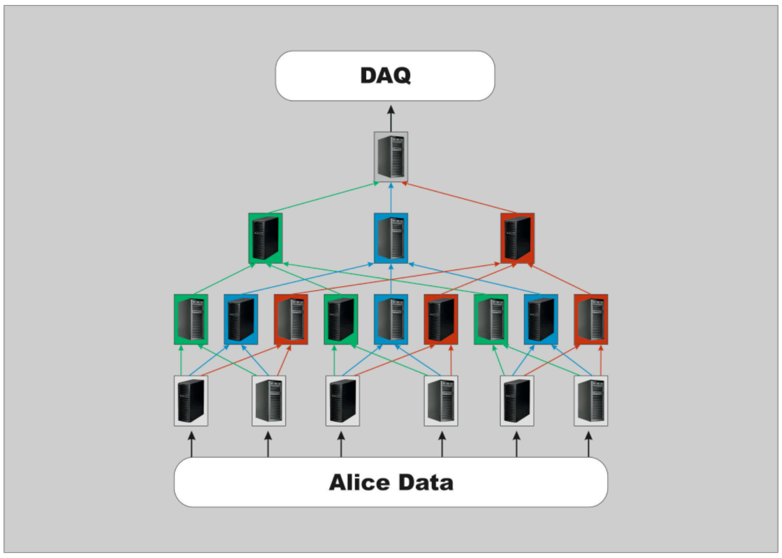


Figure 6.23.: Dataflow for the HLT-Chain application. Data arrives at leaf nodes, is processed towards the root node along (here: colored) data paths and is finally stored by Data Aquisition (DAQ).

According to the application's experts, the proper functioning of the HLT-Chain can be assessed by evaluating the number of events waiting to be processed during a HLT-Chain run. "Event" in this context is the basic unit of data to be processed by the HLT-Chain MainApp. The metric to be analyzed therefore is the queue-length of waiting events. The application status is ok unless the queue-length reaches 3000 at any point in time during an HLT-Chain run. The HLT-Chain is not tested in its native environment, the HLT-Cluster of the ALICE experiment. When the ALICE experiment entered production state in November 2010, rules became active that disallow software testing and evaluation in the HLT-Cluster. It is the recommended practice for software developers to test HLT-Chain components not in the HLT-Cluster, but in a test cluster. A software package was provided for eased installation of the HLT-Chain in test clusters. Additionally, a software module called "file-publisher" allows to replay recorded collision data with the original data rate. The file-publisher is the de-facto standard for testing the HLT-Chain. This guideline, using the file-publisher in a test cluster, was followed to evaluate the framework for this thesis.

6.6.2. Experiment Layout

For this experiment 25 nodes are used from the environment[27] presented in Section 6.1. Each node hosts a number of HLT-Chain components dedicated to specific physics computations like cluster finding, tracklet finding, tracking and merging. The HLT-Chain in this experiment is configured to process 100 events per second. This means that short term variations (on scale of seconds) for this frequency can occur but in the long run (on scale of minutes) the file-publisher seeks to adjust the input data rate such that for a $30min$ run an averaged frequency for incoming events of 100Hz is achieved. The precise configuration of HLT-Chain and the allocation of its components to physical nodes is given in Section A.1. Every node running the HLT-Chain application is also a target node for the VM-Scheduler. This is realized by associating each node with a static policy of $P_{AllocVM} : (2, 10)$ or $P_{AllocVM} : (0, 10)$ respectively, depending on whether running SecApps in an experiment run is desired or not. The leases, VMs and SecApp used in this experiment are identical to the ones used in the "Policy Modification Experiment" (see Section 6.3.1).

In the **first sub-experiment** two different setups are compared. The HLT-Chain is run for 30 minutes without SecApps ("No-VM run") in the first setup. In the second setup, the HLT-Chain is run again for 30 minutes, this time with additionally run SecApps ("With-VM run"). At $t_0 = 0s$ the VM-Scheduler is given access to the physical nodes by switching from $P_{AllocVM} : (0, 10)$ to $P_{AllocVM} : (2, 10)$ which causes leases to be provisioned. Policies $P_{CPU} : (95, 15)$, $P_{NetOut} : (75, 15)$ and $P_{NetIn} : (75, 15)$ are associated with every node. The metrics measured are the CPU utilization per node, the MainApp performance metric queue-length and the number of computed SecApp results.

In the **second sub-experiment**, the HLT-Chain and the VM-Scheduler are alternatingly started and stopped. A run consists of a sequence of actions $\in$ {StartChain, StartVM, StopChain, StopVM}. The "StartChain" and "StopChain" actions are realized by using the HLT-Chain command-line interface. The "StartVM" and "StopVM" actions are realized by switching from $P_{AllocVM} : (0, 10)$ to $P_{AllocVM} : (2, 10)$ and vice versa for all nodes of the cluster. A run takes about 40 minutes. The metrics measured are the CPU utilization per node and the MainApp performance metric queue-length. The number of SecApp results is not recorded because the repeated starting and stopping of the VM-Scheduler does not allow a reasonable interpretation of this metric.

[27]An identically configured physical node was added because of HLT-Chain requirements, thus having 25 nodes in this experiment.

6.6.3. Results

In the **first sub-experiment**, the queue-length averaged over an experiment run was measured.[28] For the first setup (No-VM run), the average queue-length was $\mu = 8.93$ with a variance of $\sigma^2 = 9.09$. The queue-length did not reach 3000 for any single measurement. So the HLT-Chain performed according to its specification.[29] For the second setup (With-VM run) the average queue-length was $\mu = 10.72$ with a variance of $\sigma^2 = 28.71$. This difference, especially the increased variance, suggests that SecApps have an influence on the MainApp for the With-VM run. The important question is, however, whether this is relevant, i.e. whether there is strong interference. Did the additional running of SecApps render the application's performance inacceptable? In Figure 6.24 the queue-length is shown

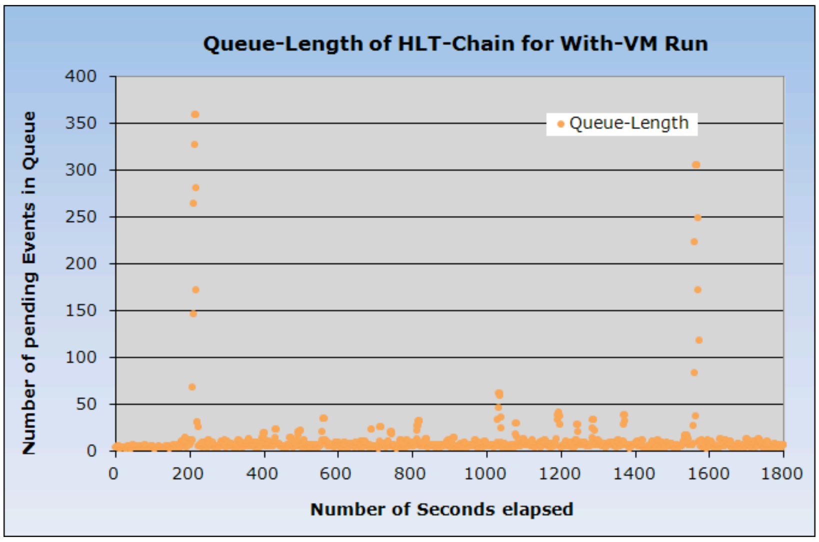
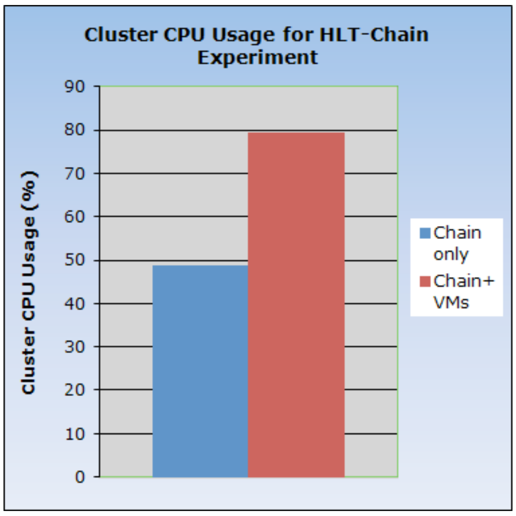


Figure 6.24.: Queue-length of HLT-Chain in 30*min* With-VM run. Displayed resolution is one queue-length value per second.

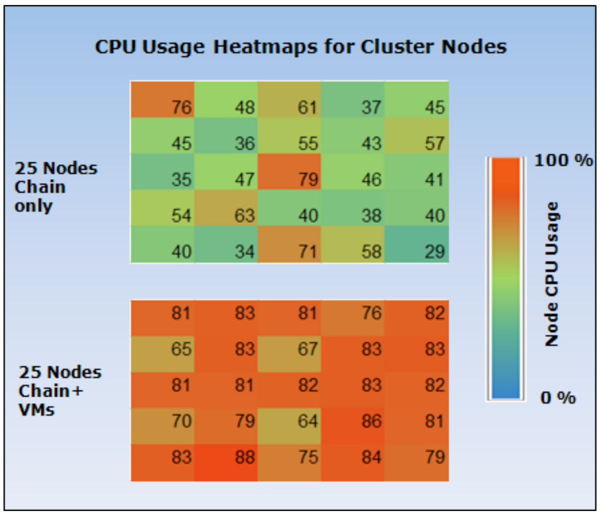
for the duration of the With-VM run. The highest value observed for the queue-length was about 350. Therefore, no strong interference occurred according to the criterion "queue-length < 3000". In Figure 6.25a the change in cluster CPU utilization comparing both setups is depicted. The average utilization in a No-VM run was 48.69 % and rises to 79.27% for the With-VM run. This effectively improves the total cluster load by about 63%. Figure 6.25b shows two heatmaps for the CPU usage on single nodes averaged over the experiment duration, thereby detailing the change in the cluster usage between the two setups. Each box in these figures represents a physical node and its coloring shows the average node CPU utilization over an experiment run. The upper heatmap shows the first setup where no SecApps were run. It can be seen that the CPU usage is not spread evenly over all nodes. There are nodes which have a higher CPU usage and others with a lower one. This is due to the HLT-Chain's component topology. The lower heatmap in Figure 6.25b represents the average node CPU usage for the second setup (With-VM run). It can be seen that not only a higher load could be obtained for any single node and the cluster in total, but also that the differences in CPU utilization on nodes

[28]The measurement interval was $\Delta t = 1s$. The queue-length is treated in a unitless form.

[29]Multiple runs were carried out with this very configuration beforehand to test the stability of the HLT-Chain (see Section A.1). With an inter-run variance of $\sigma^2 = 5.47$ for five runs and all runs performing according to specification, the HLT-Chain was considered to run stabily.



(a) Cluster CPU usage



76	48	61	37	45
45	36	55	43	57
35	47	79	46	41
54	63	40	38	40
40	34	71	58	29

81	83	81	76	82
65	83	67	83	83
81	81	82	83	82
70	79	64	86	81
83	88	75	84	79

(b) Node CPU usage

Figure 6.25.: Average CPU usage in 30*min* HLT-Chain run with and without additional VMs.

in the upper heatmap could be levelled. This additional cluster usage generated by running SecApps allowed an additional 158 results to be computed by SecApps. With ($2.5min \times 1VM$) needed for a single result, this corresponds to the amount of results computable by 14 VMs running at full scale for 30 minutes. So assuming that the full cluster could have been used by VMs this is an efficiency of $14/50 = 28\%$.[30] Taking into account that HLT-Chain utilized about 50% of the total available

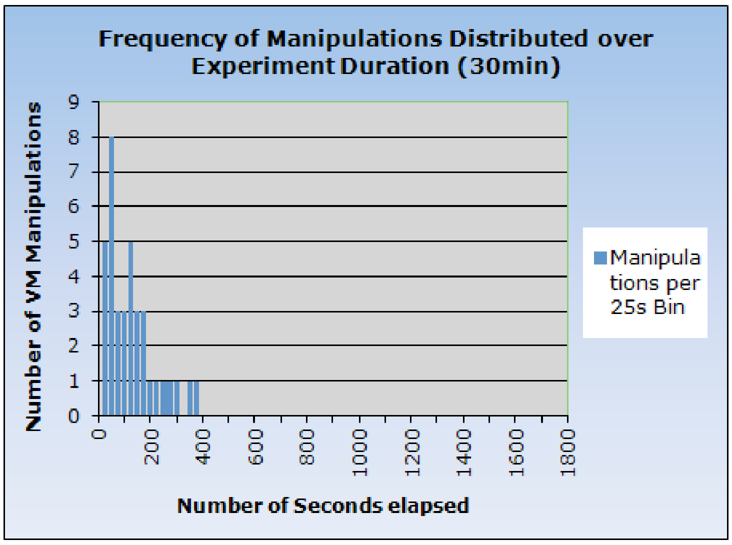


Figure 6.26.: Number of VM manipulations distributed over 30*min* HLT-Chain run, accumulated per 25*s* time bin.

[30] A VM uses one CPU core. With 25 nodes having two CPU cores in total there are 50 CPU cores.

CPU resources, the efficiency of SecApp result computation[31] even rises to about $14/25 = 56\%$. The behavior of the VM-Scheduler in this experiment, i.e. the distribution of VM manipulations over the experiment runtime, is shown in Figure 6.26 using aggregations for time bins of 25 seconds. The figure shows that the biggest amount of manipulations was executed in the first 300 seconds of the experiment, declining to zero after 400 seconds. This set of manipulations consists of starts (34) and very few migrations (3). This means that in this experiment there was a certain lead-in phase where the scheduler tried to find suitable allocations for VMs and provisioned the respective leases. After this lead-in phase, the local resource usage adaption was sufficient to guarantee policy compliance and no further manipulations were needed.

In the first sub-experiment it was shown that there is a policy configuration which enables the running of SecApps without causing strong interference to the HLT-Chain, which increases the cluster CPU usage and allows the computation of additional SecApp results. The **second sub-experiment** now evaluates the VM-Scheduler in a more demanding scenario with a sequence of actions $\in$ {StartChain, StartVM, StopChain, StopVM} applied in an experiment run taking $40min$. The sequence of actions is given in Table 6.7. Figure 6.27 shows the development of the queue-length

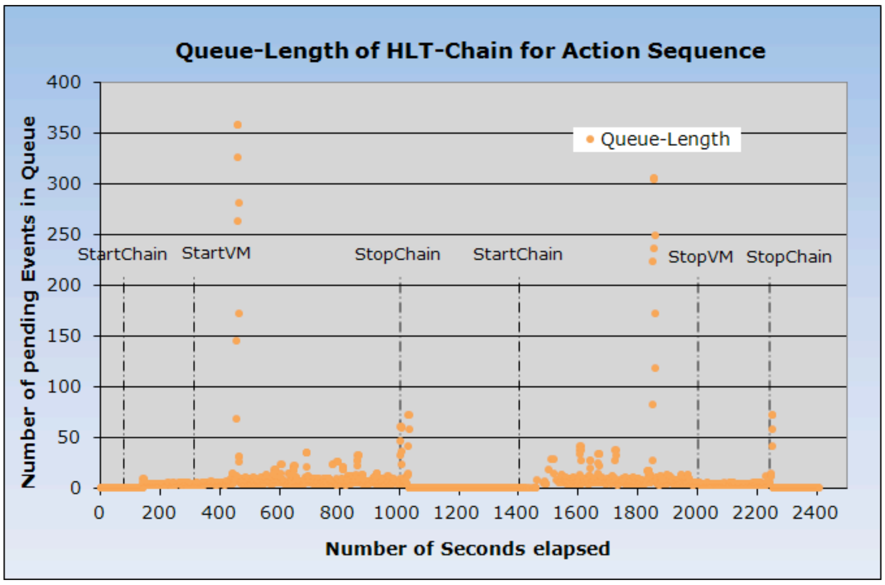


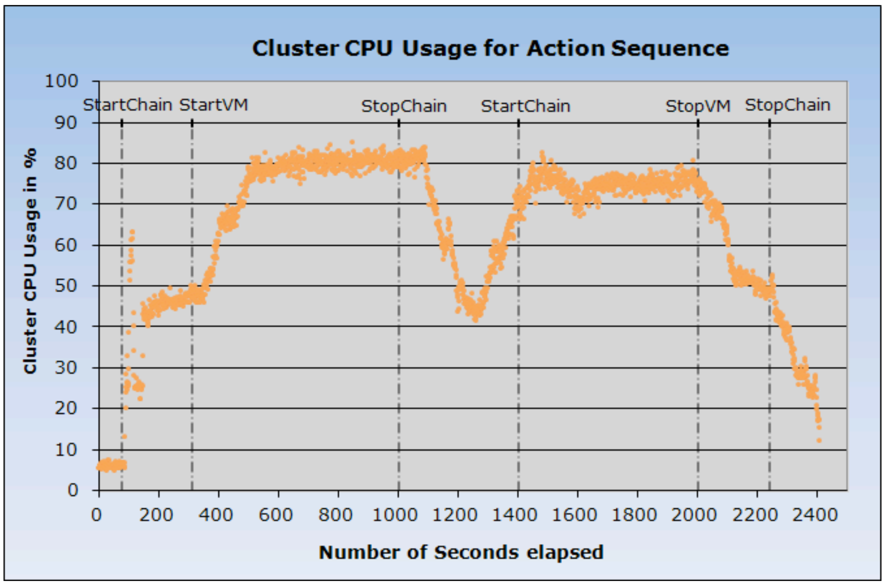
Figure 6.27.: Queue-length of HLT-Chain in $40min$ experiment with a sequence of actions applied (see Table 6.7). Displayed time resolution is one queue-length value per second.

Action	StartChain	StartVM	StopChain	StartChain	StopVM	StopChain
Time(s)	75	310	1000	1400	2000	2250

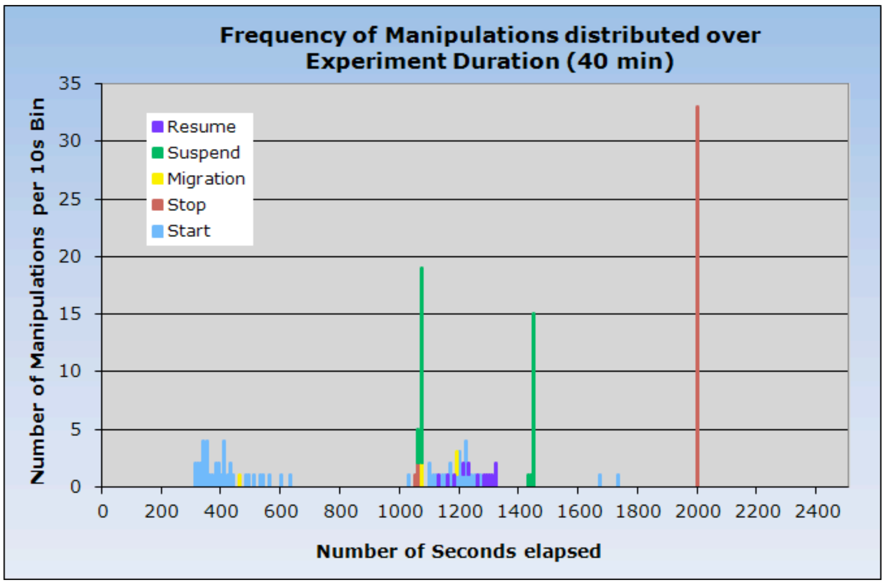
Table 6.7.: Sequence of actions for $40min$ HLT-Chain experiment.

[31]This efficiency does not take into account the buffer defined by the threshold setting. If the efficiency is calculated according to the metrics defined in Section 6.4.1 then $ResUsage = 66\%$ and $VmConv = 42\%$. However, these efficiencies do not consider the additional restriction set up by P_{NetOut} and P_{NetIn}.

metric for this sub-experiment. The action sequence is also shown in this figure together with the point in time when an action was applied. It can be seen that the queue-length metric never exceeds 400. This means that according to the criterion mentioned before ("queue-length < 3000"), there is no strong interference towards the HLT-Chain. Figure 6.28a shows the development of the cluster CPU usage over the duration of the sub-experiment. The usage rises from close to zero to about 45% at start of HLT-Chain at $t = 75s$.[32] The additional start of SecApps beginning at $t = 310s$ improves the



(a) Cluster CPU usage (one value/0.5s)



(b) VM manipulations (accumulated per 10s time bin)

Figure 6.28.: Cluster CPU usage with the sequence of actions (above) and VM manipulations (below) in 40min HLT-Chain experiment.

[32]The starting of the HLT-Chain consists of multiple successive phases. The starting time of the final phase is used as the reference point "StartChain".

CPU usage to about 81% until the HLT-Chain is stopped at $t = 1000s$. At this point, a fast drop of CPU usage to about 40% occurs. This comes at no surprise since the CPU usage generated by the HLT-Chain vanishes. In Figure 6.28b the number and type of according VM manipulations can be seen. When the chain is stopped at $t = 1000s$ several suspends (19) occur. This is presumably due to that fact that stopping of HLT-Chain itself poses an additional short-term spike in CPU or network usage which causes policy violations and resulting manipulations. The VM-Scheduler recovers the CPU usage drop by provisioning queued leases, i.e. starting and resuming VMs. This can be seen in Figure 6.28b that details the manipulations at that point in time ($t > 1000s$) and by looking at the cluster usage Figure 6.28a: The usage rises up to about 70% again, this time only by running SecApps with a still stopped HLT-Chain. This shows that the framework achieves a high cluster CPU usage whether HLT-Chain is running or not and automatically adapts to changes in HLT-Chain run state. At $t = 1400s$ the HLT-Chain is started again. The Scheduler reacts with a greater number of suspends (17) to free resources for the HLT-Chain. The queue-length does not reach the critical watermark of 3000 during that phase, so the starting of the HLT-Chain is not interferred by the manipulations executed by the VM-Scheduler. After the HLT-Chain has been started, the suspended VMs remain in that state, only two VMs are started. This is because the average CPU utilization is already at a quite high level (> 70%), leaving few resources for additional VMs. The average CPU utilization then remains at about 75% until VM-Scheduling is stopped at $t = 2000s$ by switching $P_{AllocVM} : (2, 10)$ to $P_{AllocVM} : (0, 10)$ for every node, which causes all VMs to be stopped and CPU usage to decline. Again, no harmful effects towards the performance metric queue-length could be found.

In this second sub-experiment it could be seen that even such a demanding scenario like the stopping and starting of the VM-Scheduler does not produce relevant, i.e. strong interference to the HLT-Chain. Also, the stopping and starting of the HLT-Chain could be handled gracefully by the VM-Scheduler without causing strong interference.

6.6.4. Summary

In the sub-experiments it was shown that by using the implemented approach, SecApps can be run in addition to the HLT-Chain application without causing strong interference to it. By running SecApps in a 30 minutes **first sub-experiment** the cluster CPU utilization was raised from 49% to 79% and 158 additional SecApp results were computed. This corresponds to about 56% of optimally computable SecApp results if all unused CPU resources would have been translated to SecApp results. With a **second sub-experiment** the usability of the VM-Scheduler in demanding scenarios like HLT-Chain start-up could be shown. First a running VM-Scheduler could adapt to sudden changes in cluster resource usage without strongly interfering a starting or stopping HLT-Chain. Second the VM-Scheduler was able to achieve a high cluster CPU utilization (> 70%) independent from the run state of the MainApp. Third a running HLT-Chain was not strongly interferred by starting or stopping of the VM-Scheduler.

6.7. Summary of Empirical Results

Four experiments have been conducted. In the "Policy Modification Experiment" discussed in Section 6.3 a user interface to the VM-Scheduling Framework was used to modify active policy configurations at runtime. Using a sequence of such modifications, the amount of cluster resources available to SecApps was repeatedly changed and the framework needed to adapt the VM allocations accordingly. It was shown that the VM-Scheduling Framework copes with such changes, i.e. newly available resources are used for provisioning VMs and running VMs are manipulated on time if needed to preserve policy compliance. It was shown that the effect of these modifications towards the number of computed SecApp results depends on the frequency of modifications and the timelimit of modified policies. In the "Efficiency Experiment" of Section 6.4 the effect of policy parameters timelimit and threshold on SecApp result computation and additionally exploited cluster CPU resources was examined. It could be shown that the greater policy threshold and timelimit, the more SecApp results can be computed and the more cluster CPU utilization can be generated. Measurements indicate that this observation holds for arbitrary existing MainApp CPU utilization patterns. In the "Interference Experiment" presented in Section 6.4 the effect of policy parameters timelimit and threshold on the amount of interference towards a MainApp was examined. It could be shown that the amount of interference rises with the applied policy threshold. Concerning the policy timelimit it was shown that the amount of interference towards a CPU-bound MainApp rises with the applied timelimit parameter. For a network-bound MainApp such a relation could not be found. The aforementioned experiments 6.3, 6.4 and 6.5 indicate that policies are appropriate configuration items to control the amount of interference towards a MainApp, the additionally generated cluster CPU utilization and the amount of computed SecApp results. In the "HLT-Chain Experiment" discussed in Section 6.6 it was demonstrated that a policy configuration setup exists that allows to run the VM-Scheduling Framework in a cluster environment with a time-critical MainApp, the HLT-Chain application. It was shown that no strong interference occurred, the cluster CPU utilization could be increased from 49% to 79% and additional 158 SecApp results could be computed. This number of SecApp results corresponds to 56% of the optimally computable SecApp results. The ability of the framework to cope with the demanding scenario of starting and stopping of the HLT-Chain application without causing strong interference was demonstrated as well.

7. Summary and Conclusion

In this chapter the thesis approach is summed up and a conclusion is given. As a reminder, the goal was to develop the concept and implementation of a software framework ("VM-Scheduler") for running additional applications ("SecApps") in a dedicated cluster with a time-critical application ("MainApp"). This goal was constrained by: *a*) The MainApp must not be affected in its proper functionality, *b*) this has to be achieved without controlling the MainApp and *c*) cluster usage has to be increased and additional computational results have to be generated. An examination of current state-of-the-art research showed that both the increase of resource usage / result output and the achievement of performance goals for applications are topics covered by researchers and developers. However, no existing approach satisfies the goal stated for this thesis. Either researchers assume controllability of the MainApp, provide insufficient isolation between MainApp and SecApps or lack in flexibility of methods applicable for maintaining proper functionality of the MainApp. Based on these findings, the concept and prototypical implementation of a VM-Scheduler has been developed. The prototypical implementation was evaluated empirically and found to be appropriate for achieving the thesis goal.

In the first section of this chapter the thesis work is summed up. In the second part a conclusion is drawn and the contribution of this thesis is given.

7.1. Summary

The challenge of this thesis is that by running SecApps in a dedicated cluster, the proper functionality of the MainApp can be affected. According to the given goal this needs to be prevented. As a first step it was needed to define what "affecting the proper functionality of a MainApp" means. A statement on whether a MainApp is affected by SecApps requires to have a predefined performance metric (e.g. throughput) for the MainApp. Based on this performance metric, the terms "interference" and "amount of interference" were introduced. These allow for an assessment of the effect that SecApps have towards the performance of a MainApp. It was argued that not every statistically significant effect (i.e. interference) is relevant. A statement on relevance can only be made if for the performance metric of a MainApp a target value or range exists, which serves as a criterion for the proper functionality of the MainApp. Using this criterion, the term "strong interference" was introduced to capture the notion of "SecApps affect the proper functionality of a MainApp.

Based on these definitions, the question was approached how SecApps can be given access to cluster resources such that no strong interference occurs, but cluster usage is increased and additional SecApp results can be computed. It was argued that potential reasons for interference have to be found and counteracted. The found reasons are: *a*) Software incompatibility, *b*) security holes, *c*) software bugs and *d*) competition for sparse resources between SecApps and the MainApp. By using Virtual Machines, a specific environment for every single SecApp is provided. Thereby any negative effects arising from software incompatibilities are prevented and the scope of runnable SecApps is broadened. VMs also efficiently isolate SecApps from the MainApp, thereby avoiding the effects of potential security issues and software bugs. Regarding the fourth reason (resource competition), it was argued that a decision instance is required to evaluate the usage of resources at runtime, to recognize potential resource competition and to make decisions on the amount of resources given to VMs hosting SecApps accordingly. To enable a decision instance to prevent or limit the duration of resource competition, explicit contraints on the resource usage of SecApps ("policies") were introduced. A policy defines a maximum time span ("timelimit") for which SecApps are allowed to use a resource if the total usage of that resource exceeds a certain value ("threshold"). The basic idea is that policies are configurable ("configuration items") and that the amount of interference (i.e. the effect towards the MainApp) scales with the settings for the policy parameters threshold and timelimit.

Based on a set of configured policies, a decision instance decides on when and where to run SecApps in the cluster. If necessary, the decision instance also modifies the resource usage of running SecApps in order to satisfy the policy constraints. A single SecApp is represented by an abstraction called lease. A lease consists of at least one VM hosting the application logic of the SecApp. The lease abstraction and the VM encapsulation of SecApps provide standardized methods to the decision instance to manipulate arbitrary SecApps and to modify their resource usage. The decision instance has a two-layered architecture and comprises of four main functional units. One layer is the Global Scheduler that resides on a global management node and hosts the functional units Global Provisioner (GP) and Global Reconfigurator (GR). The second layer is the Local Controller that runs on every worker node[1] and hosts the functional units Resource Usage Adaptor (RUA) and Global Trigger Unit.

The GP takes care of leases that are not running[2] and makes scheduling decisions on when and where to provision a lease by starting or resuming its VMs. It uses a non-preemptive, run-to-completion approach and employs a priority-scheduling algorithm with backfill. The other functional unit of the global layer, the GR, was developed as a runtime scheduler that decides on how and when to manipulate a running lease[3] by suspending, stopping or migrating its VMs. The GR makes these decisions based on an informed depth-first search heuristic with backtracking. The purpose of the GR is to modify the resource usage of running leases if needed to satisfy the policies constraints. It uses the method of manipulating VM states. Since such manipulations carry considerable overhead concerning resource usage and duration, the GR is complemented in its purpose by the functional units of the local layer. The RUA is part of the Local Controller and continuously modifies the share of resources like CPU or NetOut usable by a locally running VM. The adaption is realized by a closed loop integral controller per single resource. The RUA provides a fine-grained, minimal-overhead charged method to modify the resource usage of leases, acts locally on a node and independent from the GR. Thereby it decreases the relative number of cases where the GR is required to initiate VM manipulations to satify policy constraints. The fourth functional unit is the Local Trigger Unit which also is part of the Local Controller. It is an actor-of-last-resort that realizes policy compliance forcefully by stopping and aborting VMs, unless policy compliance can be maintained by GR and RUA.

This conceptual proposal was implemented in a platform-independent manner using the Python programming language. The only platform-specific code is that which is responsible for interfacing the virtualization platform, capping local VM resource usage and querying resource statistics. However,

[1]A worker node is a physical node in the cluster that is intended to run SecApps.

[2]According to the terminology used in this thesis, these leases are in queued state and wait for access to computational resources.

[3]According to the terminology used in this thesis, these leases are in provisioned state and are currently using cluster resources.

due to object-oriented software design these functionalities are encapsuled using classes with clearly defined interfaces and can be replaced easily.

Several experiments were carried out to test the implemented VM-Scheduler. In two experiments the effects of policy parameter settings for timelimit and threshold on the SecApp result output, the additional cluster CPU usage and the amount of interference caused towards a prototypical MainApp were evaluated. The results indicate that the threshold parameter setting scales monotonically increasing with all three target metrics. The results for the timelimit parameter were inconclusive. While the timelimit setting also scales monotonically increasing with the additional cluster CPU usage and SecApp result output, this does not hold for the amount of interference: The monotonically increasing relation only holds if a CPU-bound MainApp is present. If a network-bound MainApp runs in the cluster, then a lower timelimit setting can also lead to a higher amount of interference. It was shown that the amount of interference can be controlled by configuring the threshold parameter of policies. This means that an upper limit for the effect that running SecApps have towards a MainApp can be set. Policies therefore are appropriate configuration items to customize the VM-Scheduler towards a cluster and MainApp environment and to specify a trade-off between tolerated amount of interference and the benefit gained from SecApps. The conceptual approach of adapting the resource usage of SecApps at runtime in accordance with configured policies is thereby shown to be feasible for avoiding strong interference. While these experiments examined the general idea and adaptability of the VM-Scheduler to arbitrary environments, the appropriateness of the thesis approach was clearly demonstrated for the HLT-Chain application in KIP cluster. In this experiment, the VM-Scheduler increased the cluster CPU utilization from 49% to 79%, the aimed at data-rate of 100Hz for HLT-Chain was achieved, hence no strong interference occurred and additional SecApp results (code compilation) were computed. The theoretically exploitable CPU share of $100\% - 49\% = 51\%$ was translated to SecApp results with an efficiency[4] of 56%.

7.2. Conclusion

The thesis provides the concept and an implementation for a framework ("VM-Scheduler") that is able to run additional applications ("SecApps") in a dedicated cluster with a time-critical application ("MainApp"). The pivotal challenge is that of preventing the MainApp from being affected by running SecApps. The driving ideas to meet this goal are: *a*) The isolation of SecApps by running them in Virtual Machines and *b*) the dynamic resource allocation for VMs, i.e. runtime adaption of the resource share used by SecApps. This adaption is realized by using both globally initiated VM manipulations (migration, suspend/resume, start/stop) and decentralized, feedback-controller based capping of resource usage using OS interfaces. An informed depth-first search heuristic was developed to choose appropriate VM manipulations in a time-constrained manner. The proposed concept uses application-agnostic, OS provided metrics only. No interface for manipulating a MainApp is needed. The implementation is platform-independent and only requires a cluster environment to provide a Python interpreter, TCP/IP network communication, a shared storage facility and a virtualization platform. It was shown empirically that the framework provides reasonably tunable configuration items ("policies") that allow to make a trade-off between benefit gained from running SecApps and tolerated amount of interference for many different MainApp scenarios. These arguments hint for the general applicability of the concept. Fulfilment of the thesis goal was directly demonstrated for a specific

[4]Efficiency is the ratio between actually computed results and optimally computable results if all the free CPU cycles were used.

time-critical application, the HLT-Chain application. This application processes experimental physics data with soft real-time conditions. Applying the VM-Scheduler to a cluster running this application allowed to increase the cluster CPU utilization from 49% to 79%, to compute additional SecApp results and to maintain the proper functionality of the HLT-Chain application. Fulfilment of the tree goal constraints is therefore considered achieved: First it is possible to customize the framework such that the MainApp is not affected. Second, no control is exercised over the MainApp. Third, the cluster resource usage is increased and additional results are computed.

Contribution The main contribution of this thesis is that it allows to exploit unused CPU resources and to generate additional SecApp results in a dedicated cluster running the HLT-Chain application without affecting the proper functionality of this application. The novel and distinctive property of the provided VM-Scheduler is that it assumes no runtime knowledge or active control of the time-critical application in order to ensure its proper functionality. The framework only monitors the amount of unused resources and dynamically modifies the resource share consumed by SecApps. A second aspect distinguishes this thesis: The VM-Scheduler uses multiple methods for the dynamical modification of the SecApp resource share. While single methods have been studied separately before, no existing solution provides the here presented conceptual flexibility by combining the multiple methods of VM migration, suspend and stop as well as feedback-control based resource usage adaption.

8. Future Work

During the work for this thesis, two not fully covered challenges were encountered. The first one arises from a theoretical consideration - how to make sure that the framework does not exhibit self-sustaining behavior. The second one concerns runnable SecApps: The scope of SecApps was constrained to best effort schedulable applications and no empirical evaluation was made for VMs hosting tightly coupled parallel SecApps. These aspects are now discussed and suggestions are given how to advance these aspects in future work.

Self-Sustaining Behavior In this thesis the fact was mentioned that VM scheduling by GR may cause a self-sustaining behavior: The handling of policy violations causes manipulations which in turn again cause new policy violations. Basic heuristics were introduced to decrease the occurrence probability of such a phenomenon: *a*) Handling of VM resource usage locally by RUA to decrease the number of VM manipulations, *b*) the preference of manipulations that are unlikely to cause new policy violations, *c*) the exclusion of being-manipulated VMs and *d*) the exclusion of nodes with ongoing manipulations for a specific time frame after a manipulations was issued.

An informal description for self-sustaining behavior was proposed: Given that running applications (MainApp, SecApps) do not change in resource usage, the frequency of manipulations should tend to zero over time, i.e. VM manipulations have to be justified by changes in the resource usage of running applications. The framework exhibits self-sustaining behavior if this this does not hold. As a first indication, in the HLT-Chain experiment (see Section 6.6.3) it could be seen that after a lead-in phase where many manipulations occurred, the frequency of manipulation dropped to zero. This is a hint that the applied heuristics work, but cannot be regarded a proof. In future work the qualitative description of self-sustaining behavior requires a more formal or quantitative definition. Based on a such a definition, the behavior of the framework needs to be evaluated. Inspiration for a theoretical stance on this phenomenon may come from different research fields: Signal theory and Control theory both use Bounded-Input Bounded-Output (BIBO) and Nyquist stability criteria for linear, time-invariant systems. For nonlinear systems the circle stability criterion is proposed [42]. In dynamical systems theory, which basically abstracts the aforementioned theories to nonlinear, time-variant systems described by differential equations with few parameters, the Lyapunov stability criterion is used for assessing the development of a system over time [35]. Complex systems theory extends dynamical systems theory by making statements on systems that have a very large number of parameters, e.g. interdependent or interacting entities in biological organisms, actors in economy or solid state physics. This, for instance, leads to other criteria like criticality in cellular automata or chaotic behavior in random boolean networks [15, 50]. Game theory, used for describing the interaction between decision-making agents in economist models, proposes the nash equilibrium as a stability criterion[69]. A cluster with network-connected nodes and running distributed applications can be understood as a system evolving over time. Whether a formalization of the behavior of the framework is sufficiently possible and whether the mentioned criteria meet the notion of self-sustaining behavior has to be evaluated. Apart from a theoretical approach of defining and examining the behavior of the framework, it might also be needed to improve the practical measures taken to counteract self-sustaining behavior. In the authors view, the application of inhibitors, i.e. feedback control mechanisms on multiple layers (per VM, lease, node, sub-cluster, cluster) is a feasible way to control the number of manipulations, and thereby the occurrence probability of self-sustaining behavior. Further research work and empirical studies are required to examine this assumption.

SecApp Types In the scope discussion of that thesis (see Section 4.1) it was stated that only best effort schedulable applications should be used as SecApps. Source code compilation was chosen as a prototypical representative for the empirical framework evaluation. This evaluation left open the question how tightly coupled, parallel applications fare as SecApps. If no explicit deadlines or other performance constraints are given, then such applications, e.g. Message Passing Interface (MPI) based fluid dynamics simulations, can be scheduled in a best effort manner. This case was not explicitely evaluated. In this thesis it was assumed that suspending a lease consisting of multiple VMs is possible for any distributed SecApp. So, in the current implementation no special focus was laid on the avoidance of synchronization issues. A lease is simply suspended by parallely initiating the suspend manipulation for all VMs of a lease. However, for tightly coupled SecApps, like MPI applications, such an approach might cause an application to enter an erroneous state after resuming the lease. Researchers propose different methods [121, 2, 36, 78] for system-level checkpointing of parallel applications, trying to avoid synchronization issues. The current approach (parallel suspending) and the made proposals need to be evaluated and and the most reliable solution has to incorporated into the framework.
Beside of evaluating the suitability of the framework for tightly coupled, parallel SecApps, it could also be tried to broaden the scope of runnable SecApps to SLT-constrained applications. A reason was given for not considering SLT-constrained SecApps: There is no guarantee that there will be enough available resources for a SecApp at a future point in time because of not sufficiently predictable MainApp resource usage. In order to extend the range of runnable SecApps, a number of physical nodes could be reserved for VM scheduling. Such nodes then are completely under control of the VM-Scheduling framework and can be used as emergency migration targets for VMs of SLT-bound or interactive SecApps if no other resources are available. Thus, guarantees could be given for a subset of SecApps, extending the scope of runnable SecApps. Another option is the analysis of resource usage patterns for the MainApp. Time series techniques [16] could be used to identify phases with a reliably predictable amount of unused resources, which in turn could be used for, e.g., running interactive applications. Such a paradigm shift would have to be accompanied by a modification of the scheduling algorithm used by GP. Currently for GP a basic priority-based scheduling algorithm with backfill is employed and priorities are calculated upon lease request submittance. In order to satisfy deadlines, to avoid starvation or to enable fair-share scheduling for leases, dynamic priority calculation, that takes into account past queue- and computation times, should be considered for future framework releases.

A. Appendix

Additional information on topics covered in this thesis is given in this chapter. First, the HLT-Chain application configuration used for the experiments (6.6) is described. Second, the derivation of the formulas for the computational complexity of the GR algorithm is detailed. Third, the terms cloud computing, Everything as a Service (EaaS) stack and its constitutive layers are explained. Fourth, an introduction to platform virtualization is given and several products are mentioned. Fifth, general best-practice is elaborated for customizing the VM-Scheduling Framework towards a cluster and its MainApp.

A.1. HLT-Chain Application Setup

The HLT-Chain application used for the "HLT-Chain Experiment" (6.6) was configured to process a continuous replay of raw data that was recorded for a specific ALICE experiment run. The respective run has the ID 146616. Due to limited availability of resources in the test cluster only data for a specific sub-detector (Time Projection Chamber (TPC), 2 Slices) was processed. The data was propagated to the HLT-Chain application using the file-publisher method with a configurable replay frequency. The replay frequency reflects the number of events to be processed per second. The replay was tested for the frequency range between 80 Hz and 120 Hz. It showed that for frequencies greater than 100 Hz the application did not run sufficiently stable for at least 30 minutes without reaching or exceeding the maximum number of pending (queued) events. Therefore the highest acceptable frequency (100Hz) was chosen for the "HLT-Chain Experiment". The precise layout of the application, i.e. which components of the HLT-Chain were placed on which physical nodes, is given in table A.1. For more detailed information on the HLT-Chain application please refer to [104] .

A.2. Derivation of Complexity Formulas

The equation $T(V, L, N) = 1 + \sum_{i=0}^{L-1} 2^i \binom{L}{i} (N-1)^{V-Vi/L}$

was given for the computational time complexity of a **brute-force search** for finding applicable VM and lease manipulations. The equation's derivation is now detailed.

The variables $N = |\{node\}|$, $L = |\{lease\}|$ and $V = |\{VM\}|$ are used in the equation, reflecting the number of nodes, provisioned leases and running VMs known to the VM-Scheduler. Every lease has the same number of associated VMs and every such VM violates a policy. There are $(N-1)^V$ possibilities to migrate violating VMs without preempting a lease. There are $\binom{L}{i}$ possibilities to choose i leases for preemption. There are two ways to preempt a single lease (suspend/stop). After preempting i leases, $(V - Vi/L)$ running VMs remain that can be tried to be migrated. Preempting all leases means no further migration can be tried.

Node	Component	Component	Component
ti019	TPCCATracker		
ti020	TPCCATracker		
ti027	TPCCATracker		
ti037	TPCCATracker		
ti038	TPCCATracker		
ti039	TPCCATracker		
ti040	TPCCATracker		
ti041	TPCCATracker		
ti047	GMRelay1	ClusterFinder	
ti049	GMRelay2	ClusterFinder	
ti050	TPCCAGlobalMerger	ClusterFinder	
ti051	TPCCAGlobalMerger	ClusterFinder	
ti052	TPCCAGlobalMerger	ClusterFinder	
ti053	TPCCAGlobalMerger	ClusterFinder	
ti054	GlobalEsdConverter		
ti055	GlobalEsdConverter		
ti056	GlobalEsdConverter		
ti057	GlobalEsdConverter		
ti058	GlobalTrigger	ClusterFinder	
ti059	GlobalTrigger	ClusterFinder	
ti060	BarrelMultiplicityTrigger		
ti061	BarrelMultiplicityTrigger		
ti062	BarrelMultiplicityTrigger		
ti064	OutputRelay	ClusterFinder	ClusterFinder
ti067	BarrelMultiplicityTrigger		
ti075	OutputRelay	ClusterFinder	ClusterFinder

Table A.1.: Allocation of HLT-Chain components to physical nodes for the HLT-Chain experiments.

So the total number of choosable sets of manipulations is calculated by:

$$(N-1)^V + 2^1\binom{L}{1}(N-1)^{V-V/L} + 2^2\binom{L}{2}(N-1)^{V-V2/L} + ... + 2^{L-1}\binom{L}{L-1}(N-1)^{V-V(L-1)/L} + 1$$
$$= 1 + \sum_{i=0}^{L-1} 2^i \binom{L}{i} (N-1)^{V-Vi/L}$$

For a brute-force search every possible result (choosable set of manipulations) is checked against policies. The number of model-checks therefore is equal to the best- and worst-case time complexity, which is precisely the equation given above.

For the proposed **search heuristic** a worst-case time complexity of

$$T(V, L, N) = L + \frac{(N-1)V}{L}\left(L + \sum_{i=1}^{L-1}\sum_{j=i}^{L-1} 2^i \binom{j}{i}\right)$$

was given. This equation was derived as follows: There a L violation-contributing leases. For the first violation-contributing lease in *HIGH* it is tried to migrate its violating VMs. Since this is done using a first-fit approach there are at most $(N-1)V/L$ model-checks. If after tried migrations the current lease is still contributing to a violation, then it is subsequently tried to preempt a set of other leases and again tried to migrate violating VMs of the current lease. If all combinations of lease preemptions and successive migrations fail, the lease itself is preempted using suspend or finally stop. For any such possibility a model-check is conducted. So the total number of checks for the first violation-contributing lease is calculated as:

$$\frac{(N-1)V}{L} + \frac{(N-1)V}{L}\left(2^1\binom{L-1}{1} + 2^2\binom{L-1}{2} + ... + 2^{L-1}\binom{L-1}{L-1}\right) + 1$$

For the second lease in *HIGH* the same is tried. Assuming the worst-case that only the preemption of the previous lease was successful, then still $L-1$ violation-contributing leases exist. For the current lease accordingly the number of checks is:

$$\frac{(N-1)V}{L} + \frac{(N-1)V}{L}\left(2^1\binom{L-2}{1} + 2^2\binom{L-2}{2} + ... + 2^{L-2}\binom{L-2}{L-2}\right) + 1$$

For any lease in *HIGH* such a summand exists. So L summands exit. For the last lease in *HIGH* the potential preemption of other leases is redundant:

$$\frac{(N-1)V}{L} + 1$$

The total number of checks in the worst-case therefore is:

$$L + \frac{(N-1)VL}{L} + \frac{(N-1)V}{L}\left(\sum_{i=0}^{L-1}\sum_{j=1}^{L-i-1} 2^{L-i-j}\binom{L-i-1}{L-j-1}\right) = L + \frac{(N-1)V}{L}\left(L + \sum_{i=1}^{L-1}\sum_{j=i}^{L-1} 2^i\binom{j}{i}\right)$$

This is precisely the equation given above for the worst-case time complexity of the heuristic search.

A.3. Cloud Computing and EaaS Stack

The concepts of cloud computing and EaaS are briefly portrayed in this section.

Cloud Computing Cloud computing is a currently adopted metaphor to describe a class of technologies that make software and hardware remotely available to customers in a transparent manner. An important aspect is the service orientation of these approaches, i.e. to provide customer value on-demand and to make the service provider accountable for the negotiated service level. The most intuitive approach to cloud computing is by referring to the Everything as a Service (EaaS) stack, which describes a range of services that are delivered by cloud service providers. This stack is explained in the following.

EaaS Stack Everything as a Service (EaaS) is an industry-inspired taxonomy of services that extends the Service Oriented Architecture (SOA) paradigm to arbitrary, billable services in distributed computing. The most common layers of the EaaS stack are now given:

Humans as a Service For the sake of completeness, the Humans as a Service (HaaS) layer is mentioned. Personal (humans) can be regarded as a service because they provide value to customers, are billable and service terms can explicitely be stated. In general, software support and hotlines can be regarded as HaaS, though rarely referred to this way.

Software as a Service With the ubiquituos availability of internet, the business model of software companies nowadays turns to strategies different from selling products and licenses. Today, software can also be leased. Herein the software itself is not installed on a customer's computer infrastructure but installed in a computing facility outside of the customer's scope. The customer forms a contract with the software hoster and leases a time frame during which the software can be used but not owned. This layer of the EaaS stack is called Software as a Service (SaaS). Typical examples for software that is provided as a service are Customer Relationship Management (CRM) and Collaboration Frameworks. Please see [79] for an overview of current SaaS projects.

Platform as a Service The Platform as a Service (PaaS) strategy can be understood as a structural extension to the classical web hosting services. Like for the latter, a service provider offers computer infrastructure and an installed platform (OS, web server) to be used by customers to run their software. These customers pay a fee or renting amount. Such services nowadays not only cover web hosting, but a whole range of other platform services like development and runtime environments in which a customer can run its own software. Typical examples are MS Azure [125], which provides a leasable runtime environment for .NET applications and Google Apps [48], which provides a runtime environment for applications based on the Google App engine.

Infrastructure as a Service In recent years, the Infrastructure as a Service (IaaS) strategy has become more and more important. Companies do rely on using Information Technology (IT), but do not want to invest in installment, management and maintenanace of their own computing facilities. From an economical point of view, IT infrastructure is the perfect business object for outsourcing since high Total Cost of Ownership (TCO) can be avoided. A typical scenario for such outsourcing is that a company leases infrastructure in a remote computing facility (cluster) which is operated by external service providers. IaaS describes this aspect of making remote infrastructure leasable on a timely base. Such infrastructure ranges from single computers to a whole Local Area Network (LAN) consisting of a large number of servers. In addition, also server management applications like Domain Name System (DNS), Dynamic Host Configuration Protocol (DHCP), directory services and additional infrastructure items like Virtual Local Area Network (VLAN), gateways and proxies can be leased. Furthermore, backup and storage facilities are offered. Even though servers and other infrastructure items can be real hardware, the actual enabling technology of the IaaS layer is platform virtualization. This technology gives IaaS providers the flexibilty to offer services in a fine temporal, functional and structural granularity. Typical examples adopting the IaaS paradigm are Amazon EC2 [4], Amazon S3 [5] and Nebula [77]. While Amazon EC2 is the brand for offering server hardware, S3 is the term used by Amazon to refer to their storage services. Nebula offers both computing and storage facilities.

Other terms in relation to the EaaS paradigm are High Performance Computing as a Service (HPCaaS) and Data Intensive Computing as a Service (DICaaS). Both refer to large scale scientifc computing and its outsourcing to external service providers. While the first targets scientific number-crunching[1] applications, the second focuses on infrastructure and applications which process and store huge amounts of data. When linking the thesis to one of these fuzzy-defined business terms, then IaaS is the term that applies best.

[1]Number-crunching is a common, but slightly colloquial term for describing computing-intensive tasks that require a large amount of CPU resources.

IaaS, SaaS and PaaS typically are referred to as cloud services. A cloud therefore is a set of computing facilities (clusters) that hosts a service which belongs to these layers and is transparently offered to customers. Transparency refers to the fact that the customer only knows and uses the interfaces (Application Programming Interface (API), rich client, web browser), but is not aware of the location or back-end technology of the hosting environment.

A.4. Virtualization Technologies and Products

The term virtualization encompasses a number of technologies. These have in common that they provide an abstracted environment for software, e.g. OS or applications. Please refer to Smith and Nair [97] for an introduction and taxonomy. In this thesis, the term virtualization is used for hardware (platform) virtualization, more specifically same Instruction Set Architecture (ISA) hardware virtualization[2], which provides an environment for a specific OS. Such an environment is called a Virtual Machine (VM).

Technologies

Platform virtualization adds an intermediate layer between a virtualized operating system and the actual hardware. Such an intermediate layer is called hypervisor or Virtual Machine Monitor (VMM), depending on the product used for virtualization. Typically, the distinction between VMM type I and VMM type II is made. A VMM type I ("bare metal hypervisor") runs directly on the native hardware, i.e. it carries its own basic routines to manage the hardware resource access. In contrast, a VMM type II runs on top of a conventional, non-virtualized operating system ("host operating system"), either integrated into the kernel/core of the host OS or residing in user space. A VMM type II therefore uses the routines provided by the host operating system to access hardware resources. In this section, the term hypervisor is used instead of VMM. Beyond this basic classification of hypervisor technologies, three main types of platform virtualization techniques can be distinguished:

Full Virtualization The crux in virtualizing a platform and its operating system is that the OS core (Windows) or kernel (Unix/Linux) of a virtualized operating system ("guest operating system") is unaware of being virtualized. Therefore kernel-level commands of the guest OS try to access hardware directly. The virtualized kernel assumes full privileges to access the hardware, but lacks these privileges because it does not operate on the lowest layer ("ring-0") of an actual OS architecture[3] but on a virtualization layer. Guest kernel calls which try to access the hardware will cause an access violation and exceptions will be thrown by the host OS kernel. The basic principle of full virtualization is to catch such exceptions via the intermediate virtualization layer (hypervisor), to translate the request into privileged calls to the real hardware and to return a proper result to the caller (instead of an exception). This mechanism is called trap-and-emulate and is the core functionality used in full virtualization. This strategy is expensive, as every ring-0 call of the guest causes additional overhead. A refinement has been proposed for specific VM products: Binary translation tries to circumvent trap-and-emulate by looking at the instruction registers before instructions are being executed. Potential ring-0 access instructions are replaced beforehand with appropriate calls to the hypervisor. This mechanism has two purposes: First, this way access violations and exception handling can be avoided and a faster

[2]The virtual hardware provides the same CPU architecture (instruction set) like underlying real hardware CPUs.

[3]This layer, ring-0, is a privilege domain in OS architecture. Commands executed in this domain are not restricted in their access to hardware. Ring-0 corresponds to the kernel space of the host OS.

execution of virtualized systems is possible. Second, on x86 architectures commands exist which do have different semantics depending on which level (ring-0 or ring-2) they are issued from. Since the virtualized system is unaware of being virtualized, such calls will have faulty semantics. This is captured by the virtualization layer using a beforehand binary translation.

Paravirtualization Paravirtualization is a technique targeted at circumventing the performance issues of full virtualization. With paravirtualization the guest operating system is made aware of the fact that it is virtualized. This means specific drivers (network, disk) and the virtualized kernel do not try to access hardware directly via ring-0 calls, but contact the hypervisor via a specific Application Binary Interface (ABI) instead. This way the overhead of binary translation and trap-and-emulate is gone. However, this technique requires the virtualized OS to be modified, which has a negative practical relevance for production roll-out cycles and updates.

Hardware-assisted Virtualization This type of virtualization uses an approach similiar to full virtualization. While for full virtualization the trap-and-emulate/binary translation is carried out software-wise, in hardware-assisted virtualization the binary translation is done in hardware instead. Modern x86 processors have extensions (Intel-VT, AMD-V) that natively support such behavior. These extension allow virtualized Input/Output (I/O) drivers the direct access to real hardware.

Products

There are multiple mature products in the market that provide platform virtualization for x86 desktop and server environments. The following list is not exhaustive, neither concerning the number of products mentioned nor the features described. A more comprehensive overview of virtualization products and further references can be found here [29].

KVM Kernel-based Virtual Machine (KVM) [62] is a virtualization product for Linux OS derivates. It relies on hardware-assisted virtualization and partially uses binary translation. It provides a type II hypervisor, running as a loadable module of the Linux kernel. It supports checkpointing, suspend/resume and migration. KVM is open source and part of many Linux distribution repositories. Paravirtualized I/O drivers for the virtualized OS are available. KVM in its current release (1.2.0) supports the dedication of hardware Peripheral Component Interconnect (PCI) devices[4] to virtual guest systems. It allows different virtual disk types to be used, such as cow, cow2, raw images and vmdk.

VMware ESX VMware ESX [123] is a product targeted at server-based computing clusters. It relies on a type I hypervisor. This bare-metal hypervisor is a minimalistic Linux kernel ("vmkernel") with integrated virtualization support. To date, ESX uses full virtualization with binary translation, in addition hardware virtualization extensions are exploited. The product line splits up into a freely available ESXi edition with reduced functionality and the fully featured commercial ESX product. VMWare ESX is market leader in commercially deployed virtualization setups for computing clusters (in 2011, see [128]). ESX supports Storage Area Network (SAN)/Network-Attached Storage (NAS) based shared storage for virtual disks and provides its own proprietary cluster file system, Virtual Machine File System (VMFS).

[4]PCI is a bus used for connecting I/O devices with the processor.

VMware Workstation VMware Workstation [112] is a desktop and small business server virtualization product. In comparison to the ESX product line, VMware Workstation is a type II hypervisor product, running on top of a host operating system (Linux, Windows). Binary translation is the main technology used for virtualization. Hardware-assistance via Intel-VT or AMD-V can additionally be enabled if the hardware supports these features. VMware Workstation does not support migration and checkpointing, however suspend/resume is provided.

Xen Xen [11] is an open source virtualization product for Linux using the paravirtualization approach. It uses a hybrid type I+II hypervisor. Even though there is a native host operating system ("dom0"), the hypervisor is part of the host operating system's kernel (rather than a loadable module) and therefore can be regarded as preceeding the host OS kernel with prioritized access to hardware resources. Due to the use of paravirtualization, a guest OS needs to be modified such that ring-0 calls are redirected to the hypervisor rather than letting them try to access the hardware directly. The drawback is that closed source OS (like Windows) cannot be virtualized unless explicitely supported by the OS vendor. The claimed advantage is improved performance [11]. The architecture of Xen not only requires the guest system to be adapted but also needs a modification to the dom0 host OS. This is of practical relevance for maintenance and roll-out cycles. Xen offers several distinctive features. Xen not only allows migration of VMs, but also offers the runtime adaption of guest properties like number of virtual CPU cores, capping of CPU usage per VM and attribution of Random Access Memory (RAM) to VMs.

Oracle VirtualBox Virtualbox is a virtualization platform based on a type II hypervisor. It targets both desktop and server environments and is freely usable for personal and educational use. Full virtualization (binary translation) and optionally hardware-assisted virtualization are supported. VirtualBox runs on a multitude of operating systems like Windows, Mac OS X and Linux. It supports checkpointing, suspend/resume and migration and allows the usage of multiple virtual disk types like vdi, vmdk and cow. Block-level access to virtual disks on shared storage media using iSCSI is supported.

Virtual Machine Manipulation Primitives

The introduced platform virtualization products use different naming conventions for denoting the manipulation of VMs. These naming conventions are given in table A.2. The first column lists the manipulations by their name used in this thesis, while the other columns represent the corresponding product-specific terms.

Naming	Technique	KVM	VirtualBox	XEN	VMware ESX
start	start	start	start	start	start
stop	stop	stop	poweroff	shutdown	shutdown
suspend	suspendToDisk	savevm	suspend	save	suspend
resume	resumeFromDisk	loadvm	resume	restore	resume
migration	hot migration	live-migration	teleportation	live-migration	live-migration

Table A.2.: Naming of VM manipulation primitives for different platform virtualization products.

A.5. Feasibility Estimator Customization

An important aspect, the tuning of the Feasibility Estimator (FE), has to be considered when customizing the framework *a*) to a new MainApp and *b*) to a new cluster environment. The goal of this tuning is to enable the FE to make reasonably good estimations of the costs of running VMs and of executing VM manipulations. Bad estimations will not break the framework functionality, but may lead to an increased rate of globally triggered policy violations and locally triggered kills. This would decrease the benefit of the framework, i.e. the computation of additional SecApp results and improved cluster resource usage. The steps taken for adjusting the FE to the experimental environment, the KIP cluster, are now described and serve as a proposal for later FE tuning for other clusters:

The Estimation Equation The duration of a single VM manipulation can be described by its starting point t_a and its ending point t_b with $duration = t_b - t_a$. For any ongoing VM manipulation $\in \{start, stop, suspend, resume, migrate\}$ there are manipulation processes[5] on the concerned nodes that realize these manipulations:
$manipProc \in \{startProc, stopProc, suspProc, resumeProc, migOutProc, migInProc\}$.[6] The manipulaton processes use resources. While all these processes cause a certain CPU overhead, some processes like migInProc or resumeProc mainly cause overhead towards the resource NetIn and others like migOutProc and suspProc cause overhead towards the resource NetOut on a node. The FE is called at a point in time t_k and given a resource res, a future point in time t_{k+i} and a set of manipulations. Its purpose is to return the resource usage of res at point in time t_{k+i} given that the set of manipulations is applied at t_k. Equation A.1 shows the approach used by FE to estimate the future usage of a resource at t_{k+i}.

$$\begin{aligned} Usage_{res}(k+i,1) < Min\Bigg(100, \Bigg(&ProcUsage_{res}(k+i,1,MainApp) \\ &+ \sum_{n=1}^{N} ProcUsage_{res}(k+i,1,VM_n) \\ &+ \sum_{r=1}^{R} ProcUsage_{res}(k+i,1,VM_r) \\ &+ \sum_{m=1}^{M} ProcUsage_{res}(k+i,1,manipProc_m)\Bigg)\Bigg) \end{aligned} \tag{A.1}$$

The meaning of the $ProcUsage_{res}$ terms in equation A.1 is now explained:

- The $ProcUsage_{res}$ term in the first line of equation A.1 represents the projected future resource usage of a running MainApp using its averaged usage over the past five measurement intervals at t_k.
- The summand in the second line incorporates the resource usage of every VM_n that at t_k is in running state, contributes to resource usage and at t_{k+i} still runs or is being manipulated. The future resource usage of every such VM_n is estimated using its averaged resource usage over the past five measurement intervals at t_k.

[5]These manipulation processes are abstractions and do not necessarily exist as separate OS processes

[6]The migration primitive consists of two manipulation processes, migOutProc on the source node and migInProc on the target node.

- The summand in the third line reflects the summed resource usage of every VM_r that at t_k does not contribute to resource usage, but at t_{k+i} is expected to be running and contributing to resource usage (e.g. newly started, resumed or migrated VMs).
- The summand in the last line of equation A.1 reflects the summed resource usage of every ongoing manipulation represented by $manipProc_m$ at t_{k+i}.

Several values are needed for solving equation A.1 for a given resource, a set of manipulations and point in time t_{k+i}:

1. Current resource usage of MainApp and already running VMs (lines one and two). This information is available by querying the cluster model.
2. VM manipulations that take place in the interval $[k, k+i]$. Ongoing manipulations at t_k are known to the FE. To be started, i.e. proposed manipulations are parameters that are passed to the FE when queried by a functional unit like GP or GR.
3. Resource usage of newly started, resumed and migrated VMs on the target node (third line). An estimation for these values is given later in this section.
4. Duration and resource usage of VM manipulations ongoing at t_{k+i}, i.e. duration and resource usage of their corresponding manipulation processes $manipProc_m$ (fourth line). An estimation for these parameters is described next.

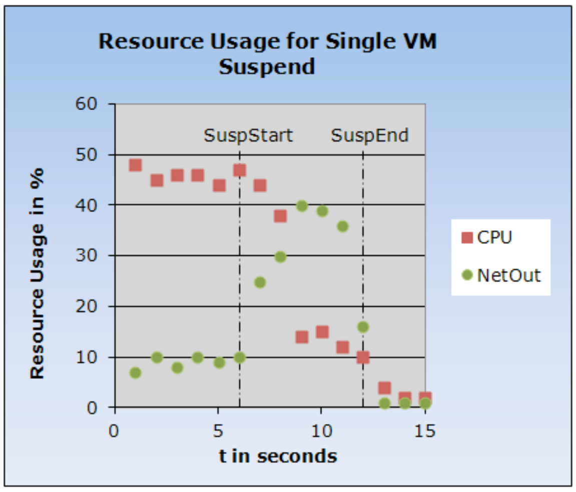
Resource Usage of VM Manipulations To obtain an estimation on the resource usage of VM manipulations, the resource usage of single manipulations in an idle environment[7] is measured first. It is assumed that this measured resource usage is the maximum of resource usage a manipulation can generate for node resources. Additional cluster activity, e.g. a running MainApp, other running VMs and manipulations, only lead to competition for resources, prolong the duration of a single manipulation and thereby decrease a manipulation's average resource usage over time. Figures A.1a and A.1b show the results of such an experiment for suspending and migrating a single VM in the idle KIP cluster (see 6.3.1, 6.1). For suspending a single VM (A.1a) it can be seen that once the manipulation starts, a drop in CPU usage occurs. This is due to the fact that a suspend consists of two phases: First the VM is halted, hence the CPU usage of the VM itself drops. Then the VM-related memory and register contents, which are required for a later resume, are transfered to the shared storage. The latter phase requires some CPU, but in most cases less than a previously running VM. For the NetOut resource an obvious increase in usage can be seen for the duration of the suspend manipulation. The variable $ProcUsage_{res}(k+i, 1, manipProc)$[8] is calculated from these experiments by substracting the latest known resource usage of a running, not yet manipulated VM from the maximum resource usage observed for a single interval during the manipulation of this very VM. For a manipulation taking i seconds this means:

$$\begin{aligned}&ProcUsage_{res}(k+i, 1, manipProc)\\&= max(Usage_{res}(k+1, 1), Usage_{res}(k+2, 1), ..., Usage_{res}(k+i, 1)) - ProcUsage_{res}(k, 1, VM)\end{aligned}$$

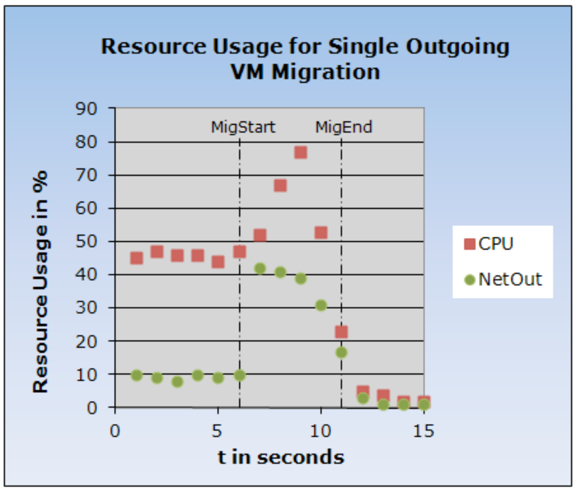
For a conservative estimation only the highest observed value is used instead of taking into account the varying resource usage during a manipulation. For manipulation processes startProc, resumeProc

[7]Nodes, storage and network are idle except for the single VM and its manipulation. Node configuration is homogeneuos.

[8]This variable describes the resource usage of a manipulation process $manipProc$ at t_{k+i} and contributes to equation A.1 in line four.



(a) Suspend resource usage



(b) Outgoing migration resource usage

Figure A.1.: Resource usage for NetOut and CPU plotted for a single VM suspend (left) and VM migration (right). The VirtualBox VM has one virtual CPU core, 512 MB RAM and runs GCC. The node has two CPU cores and a 1GBit network interface. An NFS server is used as shared storage. At $t = 0s$ the VM is up and running, no other activity is on the node or in the cluster, at $t = 6s$ the manipulation starts. Start and end of manipulations are indicated by vertical bars.

and migInProc the subtrahend is redundant since no previous resource usage at t_k by the manipulated VM on the target node exists. For the suspend manipulation in figure A.1a this means:

$$ProcUsage_{CPU}(k+6, 1, suspProc) = 47 - 45 = 2$$
$$ProcUsage_{NetOut}(k+6, 1, suspProc) = 40 - 9 = 31$$

The previous resource usage of a VM determines the outcome of this calculation, and in real life this usage is subject to changes. So for very a low CPU usage (e.g 7%) of a running VM, the observed maximum of additional CPU usage during a suspend may be higher than 2%. Nevertheless, the basic statement can be made that a suspend lowers the CPU usage and increases the NetOut usage during the manipulation in most cases. These values, $ProcUsage_{CPU}(k+6, 1, suspProc) = 2$ and $ProcUsage_{NetOut}(k+6, 1, suspProc) = 31$, are considered an estimated maximum percentage for the additional resource usage caused during a suspend manipulation. Accordingly the overhead of other manipulations is retrieved. An further example is the migration of a VM. The resource usage for this manipulation on the source node is shown in figure A.1b. Here the resource usage pattern considerably differs from the suspend scenario. This is due to the pre-copy technique used for migrations. Simply said, unless not the all memory content of a being-migrated VM is correctly transferred to the target node, the VM keeps running on the source node.[9] That results in a considerable overhead for the CPU and NetOut resources during a migration. The resulting values for the *migOutProc* are:

$$ProcUsage_{CPU}(k+6, 1, migOutProc) = 79 - 49 = 30$$
$$ProcUsage_{CPU}(k+6, 1, migOutProc) = 41 - 10 = 31$$

[9]For more precise information on techniques for hot migrations and their properties please refer to Akoush et al. [3].

These values are regarded as an estimation for the maximum percentage for the overhead of a process *migOutProc* on a source node. Accordingly the values for stopProc, migInProc, startProc and resumeProc have been obtained. These can be seen in table A.3.

manipProc	CPU(%)	NetIn(%)	NetOut(%)
suspProc	2	0	31
migOutProc	30	0	31
stopProc	0	0	0
migInProc	12	30	0
resumeProc	15	25	0
startProc	20	20	0

Table A.3.: Additional resource usage caused by different manipulation processes. These values are specific for the KIP cluster and a VirtualBox VM with one virtual CPU core, 512 MB RAM and running GNU C Compiler (GCC).

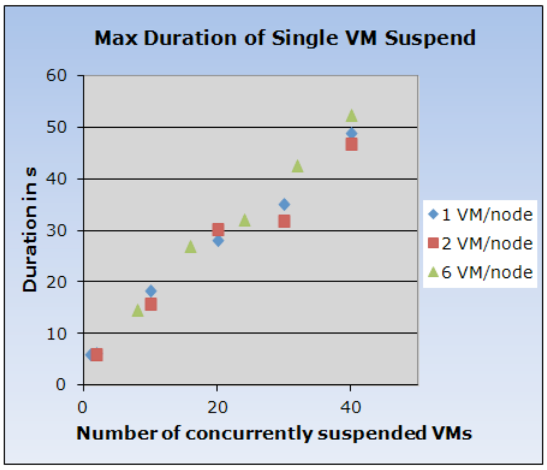
Several values in this table are equal to zero. This either means that the specific manipulation does not use a resource (e.g. NetIn for suspProc) or that starting a manipulation immediately causes a drop in the resource usage (e.g. CPU for stopProc).[10] A peculiarity is the start manipulation. By definition the start manipulation ends once the VM begins to boot, which with an irrelevant delay occurs after the command has been issued.[11] However, during the boot process such a VM stresses the NetIn resource by loading parts of the virtual disk image from shared storage. On the other hand, such a booting VM is already subject to local resource usage adaption by RUA, capping the resource usage of a VM. For the prototypical implementation, the duration of a start manipulation is the time from issueing the command until the virtualized SecApp starts calculating. While the VM is booting it is already under the realm of RUA and uses values of 20% for NetIn and CPU.

Duration of VM Manipulations For solving equation A.1 the duration of single manipulations needs to be known. The durations seen in figures A.1a and A.1b in the last paragraph are no valid representatives for real life durations, they rather represent a minimal duration. When there is activity in the cluster, e.g. multiple manipulations are executed concurrently, bottlenecks may occur, presumably for the network and shared storage resources. This prolongs the duration of manipulations. Figures A.2a and A.2b show the duration of the longest-lasting single manipulation when up to 40 VMs are concurrently[12] stopped and suspended respectively. It can be seen that a rising number of concurrently manipulated VMs prolongs this maximum duration for a single manipulation. In the figure representing the suspend manipulation it is also visible that having six VMs on a node does not have a considerable impact on the duration of the longest-lasting suspend. Most probably the bottleneck is not on the local node, but rather the route and access to the shared storage. The same can be said for the stop scenario. A linear regression for the stop manipulation (two VMs/node) gives $y = 0.08x + 0.35$. A linear regression for the suspend manipulation (two VMs/node) yields $y = x + 5.3$.

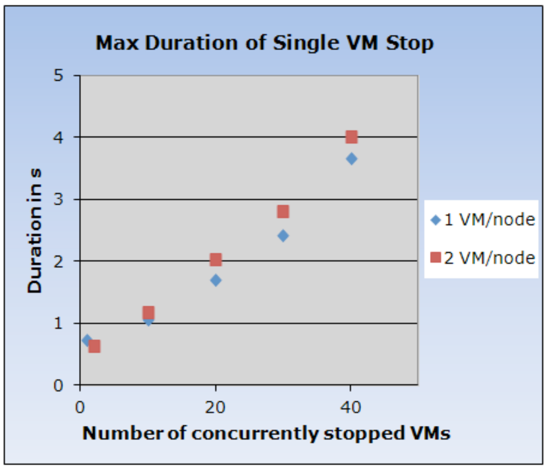
[10]For instance, in the case of stopping a VM, the VM instantly stops consuming CPU cycles and the overall usage of the CPU drops. Even though the stopping of the VM itself also uses CPU cycles, this amount is relatively small in comparison the CPU usage of a running VM. Therefore, the CPU overhead of a stopProc is considered to be zero.

[11]Given the time resolution of one second, which is used here.

[12]The start of manipulation for any VM is at the same point in time t_a.



(a) Maximum duration of VM suspend



(b) Maximum duration of VM stop

Figure A.2.: Maximum duration of single suspend and stop manipulations when up to 40 VMs are concurrently manipulated. The measurements are separated by the number of VMs running on each node. The VirtualBox VM has one virtual CPU core, 512 MB RAM and runs GCC. The node has two CPU cores and a 1GBit network interface. An NFS server is used as shared storage.

Given that both manipulation types, suspend and stop, stress the same resources (outgoing network route, shared storage write access), the simplistic assumption is made that these manipulation types and only these interfer with each other. Suspends slow down the duration of stops and vice versa. No other manipulation type (resume, start, migrate) interfers with these because *a*) local node resources do not seem to be the bottleneck and *b*) global network routes and shared storage access have different directions.[13] With this approach the duration of a single stop or suspend manipulation is calculated such:

$$\begin{aligned}
SuspAndStops &= < total\ number\ of\ ongoing\ and\ proposed\ suspends\ and\ stops > \\
duration(suspProc) &= \lceil 5.3 + SuspAndStops \rceil\ s \\
duration(stopProc) &= \lceil 0.35 + SuspAndStops/12.5 \rceil\ s
\end{aligned}$$

Figure A.3 shows the duration of the longest-lasting migration when up to 20 VMs are concurrently migrated to target nodes connected to the same switch.[14] The total number of concurrently executed migrations has no visible impact towards the duration. For a rising number of VMs/node there is an extended duration visible. Probably there is a bottleneck for the network or CPU resources of the concerned nodes. When only looking at the number of VMs/node, then a linear regression over the values for 20 (18 for 6 VMs/node) concurrently migrated VMs gives $y = 0.85x + 3.75$ with x being the number of outgoing migrations on the same node. In analogy to the method used above, it is simplistically assumed that any manipulation that stresses the same resources like a migration (suspend, stop on a source node; resume and start on a target node) has the same impact on the

[13]These assumptions are problematic for some scenarios. However for the protoyical implementation and evaluation they yielded sufficiently satisfying results, which was assessable using the local trigger rate. It is however advised to use a more sophisticated experimental setup with concurrent, potentially staged manipulations of different types when tuning the FE for another environment in order to retrieve more reliable and theoretically founded estimations.

[14]Target nodes are idle. All VMs on a source node are migrated to the same target node.

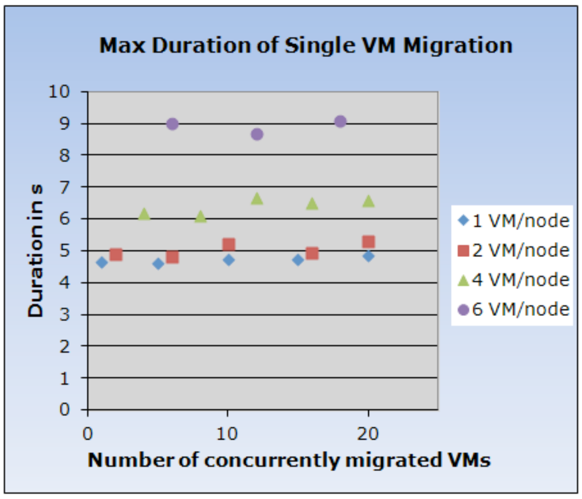


Figure A.3.: Maximum duration of a single migration when up to 20 VMs are concurrently manipulated. The measurements are separated by the number of VMs running on the source node. Target nodes are idle. The VirtualBox VM has one virtual CPU core, 512 MB RAM and runs GCC. The node has two CPU cores and a 1GBit network interface.

duration like a migration. With this approach the duration of a migration is calculated such:

$$\begin{aligned}
&ManipsSrcNode = <number\ of\ ongoing\ and\ proposed\ suspends, stops\ and\ outgoing\\
&\qquad migrations\ on\ source\ node>\\
&ManipsTrgNode = <number\ of\ ongoing\ and\ proposed\ resumes, starts\ and\ incoming\\
&\qquad migrations\ on\ target\ node>\\
&duration(migOutProc) = \lceil 0.85 + 3.75 \max(ManipsSrcNode, ManipsTrgNode)\rceil\ s\\
&duration(migInProc) \ \ = \lceil 0.85 + 3.75 \max(ManipsSrcNode, ManipsTrgNode)\rceil\ s
\end{aligned}$$

Finally, the same procedure is carried out for concurrent start and resume manipulations.[15] Like for suspend and stop, the total number of concurrently executed manipulations, i.e. the network route and shared storage read access seems to be the relevant bottleneck that determines the duration of these manipulations. In analogy to the method described above, the duration of starts and resumes is calculated such:

$$\begin{aligned}
&StartsAndResumes \ = <total\ number\ of\ ongoing\ and\ proposed\ starts\ and\ resumes>\\
&duration(resumeProc) = \lceil 9.2 + StartsAndResumes\rceil\ s\\
&duration(startProc) \quad = \lceil 21 + StartsAndResumes\rceil\ s
\end{aligned}$$

Resource Usage of New VMs Having an estimation for the duration and resource usage of single manipulations, only the resource usage of a newly started, resumed or to a target node migrated VMs is missing to solve equation A.1.

For a **newly started VM**, the usage of a resource like AllocVM is given by the resource's definition. Once a new VM is started, the VM uses one allocation slot and the overall usage of that resource is

[15]The duration of a start manipulation is determined by t_a = start of manipulation and t_b = start of SecApp.

incremented by one. The usage of resources like UsedCore and Mem is given by the VM's configuration file. If, e.g., a VM using two virtual CPU cores and 1GB memory is started, then these values are regarded as the VM's usage of the respective resources and the overall resource usage is adapted accordingly. For the resource CPU, the VM's resource usage is capped by $100 \times |cores_{VM}| \,/\, |cores_{node}|$, which for the KIP cluster and the used VM configuration is 50%. However, the resource usage of any running VM is controlled by RUA and therefore in most cases much smaller. The value of 20% was chosen for NetIn, NetOut and CPU as an estimation of the inital resource usage values. This choice is a compromise between a too low estimation, which would consequently increase the probability that a policy violation occurs and a too high value, which would reduce the probability that a VM is started, hence leaving resources unused and leases waiting for provisioning:

$$ProcUsage_{res}(k+i, 1, VM) = 20$$

For a **migrated VM**, the usage of a resource like AllocVM is given by the resource's definition. Once a VM is migrated to a target node, the VM uses one allocation slot on this node and the overall usage of this resource is incremented by one. The usage of resources like UsedCore and Mem is given by the VM's configuration file. If, e.g., a VM using two virtual CPU cores and 1GB memory is started, then these values are regarded as the VM's usage of the respective resources and the overall resource usage is adapted accordingly. For resources like CPU, NetIn and NetOut, the VM's resource usage on the source node prior to starting the migration is used as the base for estimating the resource usage on the target node. However, since a migration is often used to resolve policy violations it is reasonable to assume that there was resource shortage on the source node, which also decreased the share of resources usable by this very VM. Therefore the resource usage of a VM on the target node is the minimum of the VM's resource usage on the source node and the value of 20% for newly started VMs:

$$ProcUsage_{resTarget}(k+i, 1, VM) = \min\left(ProcUsage_{resSource}(t, 5, VM), 20\right)$$

For a **resumed VM**, the same like for migrated VM applies. For resources like AllocVM, Mem and UsedCore the predefined values are used. For resources like CPU, NetIn and NetOut, the resource usage of a VM is the minimum of the resource usage prior to its suspending and the value of 20% for newly started VMs:

$$ProcUsage_{res}(k+i, 1, VM) = \min\left(ProcUsage_{res}(t_{suspend}, 5, VM), 20\right)$$

Summary With these estimations, the resource usage of newly started, resumed or migrated VMs, the resource overhead of VM manipulation processes and the duration of manipulations can be calculated. Using these, the equation A.1 can be solved by FE for a given resource, given proposed manipulations and a future point in time t_{k+i}. The approach taken to estimate the duration and the resource usage generated by manipulations can be critizised for two reasons. First, it lacks a sufficient theoretical foundation which limits its applicability to general environments. However, for this prototypical implementation such a specific prediction model is sufficient to show the functionality of the framework for the given experimental environment. Second, since several parameters are left out in the estimation[16] and the used parameters require a more sophisticated modelling, the calculated

[16]Other parameters probably having an influence are VM memory size, network bandwidth and page dirty rate. The page dirty rate is relevant for a migration and represents the probability distribution for the number of memory pages that at a point in time are marked dirty (modified) on the source node after having already been transferred to a target host. Such

resource usage at a future point in time can be way off the actual resource usage. This means that the framework bases its global manipulation decisions on possibly bad assumptions. However, this is no problem since the framework provides mechanisms, RUA and local triggering, to decrease the resource usage of VMs and to enforce policy compliance even for bad estimations.

To sum it up, the proposed best-practice for making estimations for the future resource usage in a cluster requires to do the following:

1. Determine a good estimation for the inital resource usage of a started, resumed or migrated VM.
2. Determine the additional resource usage for manipulations of VMs.
3. Measure the impact of the number of concurrent write access manipulations (suspend, stop) on the duration of these manipulations and apply curve fitting.
4. Measure the impact of concurrent manipulations for a node on the duration of migrations and apply curve fitting.
5. Measure the impact of the number of concurrent read access manipulations (resume, start) on the duration of these manipulations and apply curve fitting.
6. Use the retrieved values and functions with equation A.1.

a page requires re-transmission. Please see [75, 27, 119] for more information.

B. Abbreviations

ABI	Application Binary Interface
ALICE	A Large Ion Collider Experiment
API	Application Programming Interface
BLO	Business-Level Objective
BMC	Board Management Controller
CERN	Conseil Européen pour la Recherche Nucléaire
CLI	Command-line Interface
CPU	Central Processing Unit
CRM	Customer Relationship Management
DAQ	Data Acquisition
DICaaS	Data Intensive Computing as a Service
DNS	Domain Name System
DHCP	Dynamic Host Configuration Protocol
EaaS	Everything as a Service
FE	Feasibility Estimator
GP	Global Provisioner
GR	Global Reconfigurator
GUI	Graphical User Interface
GCC	GNU C Compiler
HLT	High Level Trigger
HPCaaS	High Performance Computing as a Service
IaaS	Infrastructure as a Service
I/O	Input/Output
IT	Information Technology

ITIL	Information Technology Infrastructure Library
ISA	Instruction Set Architecture
HaaS	Humans as a Service
KIP	Kirchhoff Institute for Physics
KPI	Key Performance Indicator
KVM	Kernel-based Virtual Machine
LHC	Large Hadron Collider
LAN	Local Area Network
LSF	Load Sharing Facility
MPI	Message Passing Interface
MainApp	Main Application
NAS	Network-Attached Storage
NFS	Network File System
OS	Operating System
OLA	Operational-Level Agreement
OGE	Oracle Grid Engine
PCI	Peripheral Component Interconnect
PID	Proportional-Integral-Derivative
PaaS	Platform as a Service
RAM	Random Access Memory
RUA	Resource Usage Adaptor
SaaS	Software as a Service
SAN	Storage Area Network
SecApp	Secondary Application
SLA	Service-Level Agreement
SLT	Service-Level Targets
SOA	Service Oriented Architecture
TCO	Total Cost of Ownership
TCP/IP	Transmission Control Protocol/Internet Protocol

TPC	Time Projection Chamber
SMP	Symmetric Multiprocessing
SNMP	Simple Network Management Protocol
QoS	Quality of Service
VMFS	Virtual Machine File System
VLAN	Virtual Local Area Network
VM	Virtual Machine
VMM	Virtual Machine Monitor
XML	eXtensible Markup Language

Bibliography

[1] The ALICE Collaboration, "The ALICE Experiment at the CERN LHC," *J. Inst 3:S08002, 259pp*, (2008)

[2] Agarwal S. "Distributed Checkpointing of Virtual Machines in Xen Framework," *Master Thesis, Indian Institute of Technology*, Kharagpur, (2008)

[3] Akoush S., Sohan R., Rice A., Moore A.W., and Hopper A., "Predicting the Performance of Virtual Machine Migration," *Proc. International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems 2010*, Miami, (2010)

[4] Amazon Elastic Compute Cloud (Amazon EC2), `http://aws.amazon.com/ec2/`

[5] Amazon Simple Storage Service (Amazon S3), `http://aws.amazon.com/s3/`

[6] Anderson D. P., "BOINC: A System for Public-Resource Computing and Storage," *Proc. 5th IEEE/ACM International Workshop on Grid Computing*, Washington, (2004)

[7] Anderson D.P., Cobb J., Korpela E., Lebofsky M., and Werthimer D., "SETI@home: An Experiment in Public-Resource Computing," *Journal Commun. ACM, Vol. 45, No. 11. (November 2002), pp. 56-61*, (2002)

[8] Andreolini M., Casolari S., Cola M., and Messori M., "Dynamic Load Management of Virtual Machines in a Cloud Architectures," *Proc. International Conference on Cloud Computing 2009*, Beijing, (2009)

[9] Ansel J., Cooperman G., and Arya K., "Scalable User-Level Transparent Checkpointing for Cluster Computations," *Technical Report, CERN, arXiv:cs/0701037*, (2008)

[10] Bagnasco S., Betev L., Buncic P., Carminati F., Cirstoiu C., Grigoras C., Hayrapetyan A., Harutyunyan A., Peters A.J., and Saizet P., "AliEn: ALICE Environment on the GRID," *J. Physics*, Conf Ser. 119, (2008)

[11] Barham P., Dragovic B., Fraser K., Hand S., Harris T., Ho A., Neugebauer R., Pratt I., and Warfield A., "Xen and the Art of Virtualization," *Proc. SOSP2003*, Bolton Landing, (2003)

[12] Bestavros A., and Spartiotis D., "Probabilistic Job Scheduling for Distributed Real-time Applications," *Proc. First IEEE Workshop on Real-Time Applications*, New York, (1993)

[13] Blackwelder W.C., "Proving the null hypothesis," *In Clinical trials, Controlled Clinical Trials 3.4 (1982) : 345-353*, (1982)

[14] Bobroff N., Kochut A., and Beaty K., "Dynamic Placement of Virtual Machines for Managing SLA Violations," *Proc. Integrated Network Management*, Munich, (2007)

[15] Boccara N., "Modeling Complex Systems," *Springer Publishing Company, Inc., (1st ed.), 2004*, (2004)

[16] Box G.E.P., Jenkins G.M., and Reinsel G.C., "Time Series Analysis: Forecasting and Control," *Prentice-Hall, Inc., (3th ed.), Upper Saddle River, NJ, USA*, (1994)

[17] Buisson J., Andre F., and Pazat J.L., "A Framework for Dynamic Adaptation of Parallel Components," *Proc. Parallel Computing 2005: 65-72*, Malaga, (2005)

[18] Buyya R., Yeo C.S., and Venugopal S., "Market-Oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities," *Proc. International Conference on High Performance Computing and Communications*, Dalian, (2008)

[19] Byoungjip K., "Comparison of the Existing Checkpoint Systems," *Technical report, IBM T.J. Watson, October 2005*, (2005)

[20] Casavant T.L., and Kuhl J.G., "A Taxonomy of Scheduling in General-purpose Distributed Computing Systems," *Journal IEEE Transactions on Software Engineering, 14, 2 (February 1988)*, (1988)

[21] Chakode R., Yenke B.O., and Mehaut J.F., "Resource Management of Virtual Infrastructure for On-demand SaaS Services," *Proc. International Conference on Cloud Computing and Service Science 2011, 352-361*, Noordwijkerhout, (2011)

[22] Chase J., Irwin D., Grit L., Moore J., and Sprenkle S., "Dynamic Virtual Clusters in a Grid Site Manager," *HPDC 2003*, Seattle, (2003)

[23] Checconi F., Cucinotta T., and Stein M., "Real-Time Issues in Live Migration of Virtual Machines," *Proc. Euro-Par 2009, Parallel Processing Workshops*, Delft, (2009)

[24] Chen P.P-S., "The Entity-Relationship Model: Toward a Unified View of Data," *ACM Transactions on Database Systems, Vol. 1, 9-36*, (1976)

[25] Chen Q., Mehrotra R., Dubeyy A., Abdelwahed S., and Rowland K., "On State of The Art in Virtual Machine Security," *In Proc. IEEE Southeastcon, 2012*, Orlando, (2012)

[26] Choi H. W., Kwak H., Sohn A., and Chung K., "Autonomous Learning for Efficient Resource Utilization of Dynamic VM Migration," *Proc. International Conference on Supercomputing 2008*, Cairo, 2008

[27] Clark C., Fraser K., Hand S., Hansen J.G., Jul E., Limpach C., Pratt I., and Warfield A., "Live Migration of Virtual Machines," *Proc. 2nd Conference on Symposium on Networked Systems Design and Implementation, 273-286*, Berkeley, (2005)

[28] CloudZoom Virtual Appliance Marketplace, `http://cloudzoom.com/`

[29] Comparison of Platform Virtual Machines, `http://en.wikipedia.org/wiki/Comparison_of_platform_virtual_machines`

[30] The Comprehensive TeX Archive Network, `http://www.ctan.org/`

[31] Condor Job Checkpointing Limitations, `http://research.cs.wisc.edu/condor/manual/v7.3/1_4Current_Limitations.html`

[32] Condor World Map, `http://research.cs.wisc.edu/condor/map/`

[33] DMTF Common Information Model, `http://dmtf.org/standards/cim`

[34] Denis D.J., "Alternatives to Null Hypothesis Significance Testing," *Theory And Science 4.1, ISSN 15275558*, (2003)

[35] Devaney R., "An Introduction to Chaotic Dynamical Systems," *American Institute of Physics, (2nd ed.), 2003*, (2003)

[36] Emeneker W. and Stanzione D., "Increasing Reliability through Dynamic Virtual Clustering," *Proc. High Availability and Performance Computing Workshop*, Santa Fe, (2006)

[37] Enomalism Elastic Computing Platform, `http://www.enomalism.com/`

[38] Eukalyptus Cloud, `http://www.eucalyptus.com/eucalyptus-cloud`

[39] Fausett L. (Ed.), "Fundamentals of Neural Networks: Architectures, Algorithms, and Applications," *Prentice-Hall, Inc., (1st ed.), Upper Saddle River, NJ, USA*, (1994)

[40] Feitelson D.G., Rudolph L., and Schwiegelshohn U., "Parallel Job Scheduling - A Status Report," *In Proc. 10th Int. Conference on Job Scheduling Strategies for Parallel Processing (JSSPP'04)*, New York, (2004)

[41] Frachtenberg E., and Schwiegelshohn U., "New Challenges of Parallel Job Scheduling," *In Proc. 13th Int. Conference on Job Scheduling Strategies for Parallel Processing (JSSPP'07)*, Seattle, (2007)

[42] Franklin G.F., Powell D.J., and Emami-Naeini A., "Feedback Control of Dynamic Systems," *Prentice-Hall PTR, (4th ed.), Upper Saddle River, NJ, USA*, (2001)

[43] GNU C Compiler, `http://gcc.gnu.org/`

[44] Ganglia Monitoring System, `http://ganglia.info/`

[45] Ganis G., Iwaszkiewicz J., and Rademakers F., "Data Analysis with PROOF," *Proc. ACAT2008*, Erice, (2008)

[46] Gmach D., Rolia J., Cherkasova L., Belrose G., and Turicchi T., "An Integrated Approach to Resource Pool Management: Policies, Efficiency and Quality," *Proc. DSN 2008*, Anchorage, (2008)

[47] Gmach D., Rolia J., Cherkasova L., and Kemper A., "Resource Pool Management: Reactive versus Proactive or let's be Friends," *Comput. Netw. 53, 17 (December 2009), 2905-2922*, (2009)

[48] Google Apps for Business, `http://www.google.com/enterprise/apps/business/`

[49] Grit L., Irwin D., Yumerefendi A., and Chase J., "Virtual machine hosting for networked clusters: building the foundations for autonomic orchestration," *Proc. VTDC2006*, Tampa, (2006)

[50] Gros C., "Complex and Adaptive Dynamical Systems: A Primer," *Springer Publishing Company, Inc., (2nd ed.), 2009*, (2009)

[51] Gross D., Shortle J.F., Thompson J.M., and Harris C.M., "Fundamentals of Queueing Theory," *Wiley-Interscience (4th ed.), 2008, New York, NY, USA*, (2008)

[52] Hasselmeyer P., Schubert L., Koller B. and Wieder P., "Towards SLA-supported Resource Management," *Proc. International Conference on High Performance Computing and Communications*, Munich, 2006

[53] He L., Jarvis S.A., Bacigalupo D.A., Spooner D.P., and Nudd G.R., "Performance-Aware Load Balancing for Multiclusters," *Proc. 2nd International Symposium on Parallel and Distributed Processing and Applications*, Hong Kong, (2004)

[54] Hermenier F., Lorca X., Menaud J.M., Muller G., and Lawall J., "Entropy: A Consolidation Manager for Clusters," *Proc. International Conference on Virtual Execution Environments 2009*, Newport Beach, (2009)

[55] Hu L., Jin H., Liao X., Xiong X., and Liu H., "Magnet: A Novel Scheduling Policy for Power Reduction in Cluster with Virtual Machines," *Proc. Cluster 2008*, Tsukuba, (2008)

[56] Hungershoefer J., Streit A., and Wierum J-M., "Efficient Resource Management for Malleable Applications," *Technical Report PCTR-003-01, December 2001*, Paderborn, (2001)

[57] IEEE 802 LAN/MAN Standards Commitee, `http://www.ieee802.org/`

[58] Information Technology Infrastructure Library (ITIL) Homepage `http://www.itil-officialsite.com/`

[59] Irwin D., Chase J., Grit L., Yumerefendi A., Becker D., and Yocum K. G., "Sharing Networked Resources with Brokered Leases," *Proc. USENIX Technical Conference*, Boston, (2006)

[60] JBoss Application Server, `http://www.jboss.org/`

[61] Jackson D.B., "Maui Scheduler: A Multifunction Cluster Scheduler," *Beowulf Cluster Computing with Linux*, MIT Press, Cambridge, MA, pp. 351-368, (2008)

[62] Kernel Based Virtual Machine, `http://www.linux-kvm.org/page/Main_Page`

[63] Khanna G., Beaty K., Kar G., and Kochut A., "Application Performance Management in Virtualized Server Environments," *Proc. Network Operations and Management Symposium 2006*, Vancouver, (2006)

[64] Kiyanclar N., Koenig G. A., and Yurcik W., "Maestro-VC: A Paravirtualized Execution Environment for Secure On-Demand Cluster Computing," *Proc. CCGrid 2006*, Singapore, (2006)

[65] Kochut A., and Beaty K., "On Strategies for Dynamic Resource Management in Virtualized Server Environments," *Proc. International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Istanbul, (2007)

[66] Lara, C.E.M., "The SysMES Framework: System Management for Networked Embedded Systems and Clusters," *PhD Thesis, University of Heidelberg, 2011*, (2011)

[67] "The Large Hadron Collider," *Nature Insight, Vol. 448 Issue No. 7151 / 19 July*, (2007)

[68] Lee C., Lehoczky J., Siewiorek D., Rajkumar R., and Hansen J., "A scalable solution to the multi-resource QoS problem," *Proc. Real-Time Systems Symposium 1999*, Phoenix, (1999)

[69] Leyton-Brown K., and Shoham Y., "Essentials of Game Theory: A Concise, Multidisciplinary Introduction," *Morgan & Claypool Publishers, (1st ed.), San Rafael, CA, USA*, (2008)

[70] libvirt Virtualization API, `http://libvirt.org/`

[71] Linux Documentation, `http://http://linux.die.net/`

[72] Liu C., Zhao Z., and Liu F., "An Insight into the Architecture of Condor - A Distributed Scheduler," *Proc. CNMT2009*, Wuhan, (2009)

[73] Maghraoui K.E., Desell T.J., Szymanski B.K., and Varela C.A., "Malleability, Migration and Replication for Adaptive Distributed Computing over Dynamic Environments," *Parallel Processing and Applied Mathematics*, Gdansk, (2007)

[74] Meng X., Pappas V., and Zhang L., "Improving the Scalability of Data Center Networks with Traffic-Aware Virtual Machine Placement," *Proc. INFOCOM 2010*, San Diego ,(2010)

[75] Nagarajan A.B., Mueller F., Engelmann C., and Scott S.L., "Proactive Fault Tolerance for HPC with Xen Virtualization," *Proc. International Conference on Supercomputing, 23-32, 2007*, New York, (2007)

[76] Nagios Monitoring System, `http://www.nagios.org/`

[77] Nebula, `http://www.nebula.com/`

[78] Nicolae B., "BlobCR: Efficient checkpoint-restart for HPC applications on IaaS clouds using virtual disk image snapshots," *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Orsay, (2011)

[79] OpenCrowd Cloud Taxonomy: Software-As-A-Service, `http://cloudtaxonomy.opencrowd.com/taxonomy/software-as-a-service/`

[80] Oracle Cloud Resource Model API, `http://www.oracle.com/technetwork/topics/cloud/oracle-cloud-resource-model-api-154279.pdf`

[81] Oracle Grid Engine, `http://www.oracle.com/us/products/tools/oracle-grid-engine-075549.html`

[82] Oracle Virtual Assembly Builder, `http://www.oracle.com/us/products/middleware/application-server/virtual-assembly-builder-067878.html`

[83] Oracle Virtual Box, `http://www.virtualbox.org/`

[84] Padala P., Hou K. Y., and Shin K. G., "Automated Control of Multiple Virtualized Resources," *Proc. European Conference on Computer systems and Cloud Computing 2009*, Nuremberg, (2009)

[85] Pascual J.A., Navaridas J., and Miguel-Alonso J., Effects of Topology-Aware Allocation Policies on Scheduling Performance *Springer-Verlag, Job Scheduling Strategies for Parallel Processing, 138-156, ISBN 978-3-642-04632-2*, (2009)

[86] Pepple K., "Deploying OpenStack (Real Time Bks)," *O'Reilly Media, Inc., ISBN 9781449311056*, (2011)

[87] Pinedo M.L., "Scheduling: Theory, Algorithms, and Systems," *Springer Publishing Company, Inc., (3rd ed.), 2008*, (2008)

[88] Platform Load Sharing Facility, `http://www.platform.com/workload-management/high-performance-computing`

[89] Raghavendra R., Ranganathan P., Talwar V., Wang Z., and Zhu X., "No Power Struggles: Coordinated Multi-Level Power Management for the Data Center," *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems 2008*, Seattle, (2008)

[90] Real Time Scheduling Theory: A Historical Perspective, *Journal Real-Time Systems, Vol. 28, Issue 2-3, 101 - 155, Nov-Dec 2004*, (2004)

[91] Red Hat Enterprise Linux, `http://www.redhat.com/products/enterprise-linux/`

[92] Rolia J., Cherkasova L., Arlitt M., and Andrzejak A., "A Capacity Management Service for Resource Pools," *Proc International Workshop on Software and Performance 2005*, New York, (2005)

[93] rPath Cloud Engine, `http://www.rpath.com/product/rpath-cloud-engine.php`

[94] Ruth P., Rhee J., Xu D., Kennell R., and Goasguen S., "Autonomic Live Adaptation of Virtual Computational Environments in a Multi-Domain Infrastructure," *Proc. International Conference on Autonomic Computing 2006*, Dublin, (2006)

[95] Sha L., Abdelzaher T., Arzen K.E., Cervin A., Baker T., Burns A., Buttazzo G., Caccamo M., Lehoczky J., and Mok A.K., "Real Time Scheduling Theory: A Historical Perspective," *Real-Time Syst. 28, 2-3 (November 2004), 101-155*, Norwell, (2004)

[96] Shoykhet A., Lange J., and Dinda P., "Virtuoso: A System For Virtual Machine Marketplaces," *Technical Report NWU-CS-04-39, July, 2004*

[97] Smith, J. E., and Nair, R., "The Architecture of Virtual Machines" *Journal IEEE Computer, Vol. 38 Issue 5, 32-38, May 2005*, (2005)

[98] Sonnek J., Greensky J., Reutiman R., and Chandra A., "Starling: Minimizing Communication Overhead in Virtualized Computing Platforms Using Decentralized Affinity-Aware Migration," *CSE, University of Minnesota, Tech. Rep. TR09-030*, Minnesota, (2010)

[99] Sotomayor B., Keahey K., and Foster I., "Combining Batch Execution and Leasing Using Virtual Machines," *Proc. CCA'08*, Hagen, (2008)

[100] Sotomayor B., Montero R. S., Llorente I.M., Foster I., "Resource Leasing and the Art of Suspending Virtual Machines," *Proc. International Conference on High Performance Computing and Communications 2009*, Seoul, (2009)

[101] Sotomayor B., Montero R.S., Llorente I.M., and Foster I., "Virtual Infrastructure Management in Private and Hybrid Clouds," *IEEE Internet Computing, vol. 13, no. 5, pp. 14-22, Sep./Oct. 2009*, (2009)

[102] Stage A., and Setzer T., "Network-Aware Migration Control and Scheduling of Differentiated Virtual Machine Workloads," *Proc. International Conference on Software Engineering*, Vancouver, (2009)

[103] Stallings W., "Operating Systems Internals and Design Principles (fifth international edition)," *Prentice Hall ISBN 0-13-147954-7*, (2004)

[104] Steinbeck T.M., "The ALICE High Level Trigger Data Transport Framework," *Ph.D. Thesis, University of Heidelberg, arXiv:cs.DC/0404014*, Heidelberg, (2004)

[105] Steinbeck T.M., Lindenstruth V., and Tilsner, H., "New Experiences with the ALICE High Level Trigger Data Transport Framework," *Proc. CHEP04*, Interlaken, (2004)

[106] Stoess J., Klee C., Domthera S., and Bellosa F., "Transparent, Power-Aware Migration in Virtualized Systems," *Proc. GI/ITG Fachgruppentreffen Betriebssysteme*, Karlsruhe, (2007)

[107] Sun M.H., and Blough D.M., "Fast, Lightweight Virtual Machine Checkpointing," *Technical Report, Georgia Institute of Technology, GIT-CERCS-10-05*, (2010)

[108] TORQUE Resource Manager, `http://www.adaptivecomputing.com/products/open-source/torque/`

[109] Tesauro G., Jong N.K., Das R., and Bennani M.N., "A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation," *Proc. International Conference on Autonomic Computing 2006*, Dublin, (2006)

[110] TurnKey Linux Virtual Appliance Library, `http://www.turnkeylinux.org/`

[111] Turner A., Sangpetch A., and Kim H.S., "Empirical Virtual Machine Models for Performance Guarantees," *Proc. Large Installation System Administration 2010*, San Jose, (2010)

[112] VMware Workstation, `http://www.vmware.com/products/workstation/overview.html`

[113] VMware vSphere Feature Comparison, `http://www.vmware.com/virtualization/why-choose-vmware/vsphere-feature-comparison/most-trusted-virtual-infrastructure.html`

[114] VMware vSphere, `http://www.vmware.com/products/vsphere/mid-size-and-enterprise-business/overview.html`

[115] Van H. N., Tran F. D., and Menaud J.M., "Autonomic virtual resource management for service hosting platforms," *Proc. ICSE Cloud 09*, Vancouver, (2009)

[116] Van H.N., Tran F.D., and Menaud J.M., "SLA-Aware Virtual Resource Management for Cloud Infrastructures," *International Conference on Computer and Information Technology 2009*, Dhaka, (2009)

[117] Venkatesha S., Sadhu S., and Kintali S., "Survey of Virtual Machine Migration Techniques," *Workshop Advanced Operating Systems, University of California*, Santa Barbara, (2009)

[118] Verma A., Ahuja P., and Neogi A., "Power-Aware Dynamic Placement of HPC Applications," *International Conference on Supercomputing*, Island of Kos, (2008)

[119] Voorsluys W., Broberg J., Venugopal S., and Buyya R., "Cost of Virtual Machine Live Migration in Clouds: A Performance Evaluation," *Proc. CloudCom 2009 Proceedings of the 1st International Conference on Cloud Computing, 254 - 265*, Beijing, (2009)

[120] Walters J. P., Bantwal B., and Chaudhary V., "Enabling Interactive Jobs in Virtualized Data Centers," *Proc. CCA 2008*, Hagen, (2008)

[121] Walters J. P., and Chaudhary V., "FT-OpenVZ: A Virtualized Approach to Fault-Tolerance in Distributed Systems," *Proc. International Conference on Parallel and Distributed Computing Systems 2007*, Las Vegas, (2007)

[122] WebSphere Application Server, `http://www-01.ibm.com/software/webservers/appserv/was/`

[123] Website of VMWare ESX, `http://www.vmware.com/de/products/vi/esx/`

[124] Weng C., Li M., Wang Z., and Lu X., "Automatic Performance Tuning for the Virtualized Cluster System," *Proc. ICDCS 2009*, Montreal, (2009)

[125] Windows Azure, `http://www.windowsazure.com/en-us/`

[126] Wood T., Shenoy P., Venkataramani A., and Yousif M., "Black-Box and Gray-Box Strategies for Virtual Machine Migration," *Proc. NSDI '07*, Cambridge, (2007)

[127] Wood T., Tarasuk-Levin G., Shenoy P., Desnoyers P., Cecchet E., and Corner M.D., "Memory Buddies: Exploiting Page Sharing for Smart Colocation in Virtualized Data Centers," *Proc. International Conference on Virtual Execution Environments 2009*, Newport Beach, (2009)

[128] Worldwide Quarterly Server Virtualization Tracker, `http://www.idc.com/tracker/showproductinfo.jsp?prod_id=39`

[129] Xiong P., Zhikui W., Gueyoung J., and Calton P., "Study on Performance Management and Application Behavior in Virtualized Environment," *Network Operations and Management Symposium 2010*, Osaka, (2010)

[130] Xu J., Adabala S., and Fortes J., "Towards Autonomic Virtual Applications in the In-VIGO System," *Proc. International Conference on Autonomic Computing 2005*, Seattle, (2005)

[131] Zhang J., and Figueiredo R. J., "Application classification through monitoring and learning of resource consumption patterns," *Proc. IEEE IPDPS 2006*, Rhodes Island, (2006)

[132] Zhao M. and Figueiredo R.J., "Experimental Study of Virtual Machine Migration in Support of Reservation of Cluster Resources", *Proc. International Workshop on Virtualization Technologies in Distributed Computing*, Reno, (2007)

[133] Zhikui W., Yuan C., Gmach D., Singhal S., Watson B.J., Rivera W., and Xiaoyun Z., "AppRAISE: Application-Level Performance Management in Virtualized Server Environments," *IEEE Transactions on Network and Service Management 2009*, (2009)

[134] Zhu X., Wang Z., and Singhal S., "Utility-Driven Workload Management using Nested Control Design," *Proc. American Control Conference 2006*, Minneapolis, 2006

[135] Zhu X., Young D., Watson B. J., Wang Z., Rolia J., Singhal S., McKee B., Hyser C., Gmach D., Gardner R., Christian T., and Cherkasova L., "1000 Islands: An Integrated Approach to Resource Management for Virtualized Data Centers," *Proc. ICAC-09*, Tiruchirappalli, (2009)

Virtual Machine Scheduling in Dedicated Computing Clusters

Problemstellung Computercluster dienen u.a. der Ausführung verteilter Anwendungen. Zu diesen zählen auch Anwendungen mit zeitkritischem Verhalten. Diese verarbeiten einen kontinuierlichen Strom an Eingangsdaten und haben hierbei bestimmten Zeitbedingungen zu genügen. Beispiel für solche Bedingungen sind maximale Antwortzeit bei Nutzeranfragen in Web- und Cloudanwendungen oder anvisierte on-line Verarbeitungfrequenz in Experimenten der Teilchenphysik. Eine übliche Herangehensweise zur Sicherstellung dieser Zeitbedingungen ist „Over-Provisioning“. Diesem Ansatz zufolge wird eine zeitkritische Anwendung in einer speziell auf sie zugeschnittenen Hard- und Softwareumgebung, einem dedizierten Cluster ("Dedicated Cluster"), betrieben. Dies garantiert, daß die angestrebte Anwendungsleistung für jede im Vorhinein spezifizierte Dateneingangsrate erreicht wird. Dieses Vorgehen hat jedoch einen Nachteil: Ist die Eingangsdatenrate niedriger als die spezifizierte Maximalrate, werden Hardwareressourcen nicht zufriedenstellend ausgelastet. Ein Beispiel ist die HLT-Chain Applikation [8], welche Daten des ALICE Experiments am CERN [1] zu dessen Laufzeit verarbeitet. Hier führen Schwankungen in der zu verarbeitenden Datenrate zu einer Unteraulastung der Clusterumgebung. Aus Kosten- und Effizienzgründen ist jedoch eine Ausnutzung zeitweise unausgelasteter Ressourcen wünschenswert.

Dies läßt sich durch die zusätzliche Nutzung der dedizierten Hard- und Softwareumgebung durch Drittanwendungen realisieren. Dies bringt jedoch verschiedene **Herausforderungen** mit sich:

- Drittanwendungen benötigen eine geeignete Software- und Betriebssystemumgebung. Bei Inkompatibilität mit der vorhandenen Umgebung (Bibliotheken, OS, Architektur) ist eine mit Aufwand verbundene Anpassung der bestehenden Umgebung oder der Drittanwendung selbst nötig, wenn deren Einsatz gewünscht ist. Jedoch sind sowohl die Kosten[1] einer solchen Anpassung als auch die alternative Einschränkung auf ohne Anpassung lauffähige Drittanwendungen nicht erstrebenswert.

- Durch Installation und Betrieb von Drittanwendungen in einer dedizierten Umgebung kann es zu Sicherheitsproblemen kommen: Sowohl explizit platzierter Schadcode als auch Lücken in Betriebskonzept und Softwareimplementierung von Drittanwendungen können zu Angriffen führen. In deren Folge sind der Verlust sensibler Daten, die leistungsmäßige Beeinträchtigung oder sogar der Absturz der zeitkritischen Anwendung möglich.

- Durch den Betrieb von Drittanwendungen können Softwarefehler („Bugs“) in die dedizierte Umgebung eingeführt werden: Durch Fehler wie Memory-Leaks in Drittanwendungen kann die Betriebssystemumgebung und damit auch der spezifikationsgemäße Betrieb der zeitkritischen Applikation beeinträchtigt werden.[2]

[1]Unter Kosten ist hier die Gesamtheit aller Kosten (Total Cost of Ownership) zu verstehen. Diese umfaßt neben den Anschaffungskosten u.a. auch Betriebs-, Personal- wie auch Opportunitätskosten.

[2]Beispielsweise kann ein Memory-Leak unter Linux die Notfallroutine „Out of Memory Killer“ [6] auslösen. Diese erzwingt den Abbruch von Userspace-Prozessen um Hauptspeicher freizugeben. Der Ausschluß von für die zeitkritische Anwendung notwendigen Prozessen ist konfigurierbar. Dies ist jedoch nicht ausreichend um den ordnungsgemäßen Betrieb der zeitkritischen Anwendung sicherzustellen: Zum einen bietet dieses heuristisches Verfahren keine Garantie daß kein notwendiger Prozess beendet wird, zum anderen geht die Ausführung der Routine mit erheblichen Swappingaufwand einher und reduziert hierdurch die Systemleistung.

- Zur Laufzeit nutzt eine Drittanwendung Ressourcen (z.B. CPU, Netzwerk, RAM), welche für die alleinige Nutzung durch die zeitkritische Anwendung dimensioniert wurden. Dies kann zu einem Konkurrenzverhalten zwischen Anwendungen führen und bei Ressourcenknappheit die Verletzung der Zeitbedingungen der zeitkritischen Anwendung bewirken. Das Ausmaß der Ressourcennutzung durch Drittanwendungen muß daher gesteuert werden. Kommt es zu einer Variation der Eingangsdatenrate und damit einhergehend auch des Ressourcenbedarfs der zeitkritischen Anwendung, so muß die Ressourcennutzung durch Drittanwendungen dynamisch zur Laufzeit angepaßt werden, um eine Verletzung der Zeitbedingungen zu vermeiden.

Auf Basis dieser Herausforderungen läßt sich die **Zielstellung der Arbeit** folgendermaßen formulieren: Ziel ist es, ein Software Framework bereitzustellen, welches den Betrieb von Drittanwendungen in einem dedizierten Cluster ermöglicht um ungenutzte Ressourcen auszunutzen. Das Framework muß gewährleisten, daß die Funktionalität einer zeitkritischen Anwendung nicht beinträchtigt wird. Das Software Framework kann das Ressourcennutzungsverhalten der Drittanwendungen beeinflussen, hat jedoch keine Möglichkeit die zeitkritische Anwendung selbst zu steuern.[3] Durch den Betrieb von Drittanwendungen muß die Effizienz des Clusters gesteigert werden, d.h. *a*) die Anzahl berechneter Resultate wird gesteigert und *b*) die CPU-Auslastung des Clusters wird erhöht.

Die zeitkritische Anwendung wird im folgenden Main Application (MainApp) genannt. Eine im Cluster zu betreibende Drittanwendungen heißt Secondary Application (SecApp).

Existierende Ansätze Eine Evaluation ergab, daß nur ein Produkt, Condor [3], die Effizienzsteigerung eines Clusters durch SecApps zum Ziel hat und dabei die Nichtbeeinträchtigung einer separat existierenden Mainapp berücksichtigt. Dieses Produkt bietet jedoch nur limitierte Möglichkeiten, die Ressourcennutzung durch SecApps zur Laufzeit anzupassen. Ebenso ist der Schutz vor Softwarefehlern und Sicherheitslücken von SecApps nicht gewährleistet. Daher ist dieses Produkt nicht für die genannte Zielstellung verwendbar. Es existieren neben diesem Produkt Arbeiten, welche sich Teilaspekten der Zielstellung widmen. So ist sowohl der spezifikationsgemäße Betrieb einer einzelnen Anwendung, als auch die Effizienzsteigerung eines Clusters ein Thema in Forschung und Entwicklung. Um schwankenden Ressourcenanforderungen einer Anwendung nachzukommen, wird derzeit der Einsatz von Virtualisierungstechnologien in Betracht gezogen (u.a. [10][9][2]). Diese bieten Manipulationsprimitive, mit denen der Ressourcenbedarf virtualisierter Anwendungen dynamisch gedeckt werden kann.[4] Exisitierende Ansätze gehen aber von der Virtualisierung der MainApp aus, erforschen nur ein einzelnes Manipulationsprimitiv (z.B. Hot-Migration) oder bieten keine hinreichend getestete Implementierung. Im Bereich der Clustereffizienzsteigerung hat Job Scheduling eine lange Historie. Job Scheduler (u.a. [3][5][7]) ermöglichen den Parallelbetrieb mehrerer Anwendungen in einem Cluster und sind in der Lage bestimmte Zielmetriken wie genutzte CPU, Job Durchsatz und Job Wartezeit zu optimieren. Sie bieten jedoch wenig Möglichkeiten, einmal getroffene Ressourcenzuordnungsentscheidungen zu revidieren und sind daher nicht flexibel genug um auf die variierenden Ressourcenbedürfnisse einer MainApp zu reagieren. Ebenso bieten sie keine ausreichende Absicherung gegen Softwarebugs und Attacken und sind damit nicht geeignet, die Zielstellung dieser Arbeit zu realisieren.

[3]Dies schließt Ansätze aus, welche die API einer zeitkritischen Anwendungen nutzen um deren Funktionalität sicherzustellen. Ebenso können keine Hypervisorkommandos genutzt werden, um eine virtualisierte zeitkritische Anwendung zu skalieren. Grund für den Ausschluß ist die Nichtanwendbarkeit auf den Anwendungsfall HLT-Chain Applikation.

[4]Abhaengig von der konkreten Virtualisierungsplattform sind unter anderem Suspendierung, Hot-Migration und dynamische Anpassung der genutzten CPU-Cores einer virtuellen Maschine mögliche Laufzeitmodifikationen.

Vorgeschlagene Lösung Für den Betrieb von SecApps in einem dedizierten Cluster muß sichergestellt werden, daß die MainApp nicht in ihrem Laufzeitverhalten beeinträchtigt wird. Um diese Beeinträchtigung zu beurteilen bedarf es einer quantitative Performanzmetrik. In der vorliegenden Arbeit werden die Begriffe Interferenz, Betrag der Interferenz und starke Interferenz eingeführt. Interferenz liegt vor, wenn durch das Betreiben einer SecApp eine statistisch signifikante Abweichung für eine Performanzmetrik der MainApp beobachtet werden kann, z.B. eine signifikante Erhöhung der mittleren Antwortzeit pro Nutzeranfrage. Betrag der Interferenz bezeichnet die Differenz zwischen den Metrikwerten welche mit und ohne zusätzliche SecApps gemessen werden. Eine solche beobachtete Abweichung für die Performanzmetrik (Interferenz) läßt aber noch keinen Aufschluß darüber zu, ob die Beeinträchtigung der MainApp relevant für deren korrekte Funktionalität ist. Hierzu muß ein Zielwerts für die Performanzmetrik spezifiziert werden. Liegt eine solche Spezifikation vor, z.B. in Form eines Key Performance Indicators[5] oder eines anderweitig formulierten Zielwertes, z.B. maximale Anwortzeit pro Nutzeranfrage, dann lässt sich die Relevanz des Einflusses beurteilen. Starke Interferenz liegt vor, wenn sowohl eine statistisch signifikante Abweichung (Interferenz) existiert, als auch ein gewisser Zielwert für die Performanzmetrik der MainApp nicht erreicht wird. Somit kann man das Ziel, eine laufende MainApp nicht zu beinträchtigen, mit der Vermeidung starker Interferenz gleichsetzen.

Um starke Interferenz zu vermeiden, müssen deren potentielle Ursachen erkannt werden. Wie bereits erwähnt können *a*) Softwareinkompatibilität, *b*) Sicherheitslöcher *c*) Softwarefehler und *d*) Ressourcenkonkurrenz zur Beinträchtigung einer MainApp führen. In der vorliegenden Arbeit wird die Kapselung von SecApps in **Virtuellen Maschinen** (VMs) vorgeschlagen. Durch Virtualisierung werden die Laufzeitumgebungen von MainApp und SecApps effektiv voneinander getrennt. Dies verhindert zum einen die durch Sicherheits- und Softwareprobleme von SecApps ausgelöste Interferenz, zum anderen erlaubt es die Bereitstellung dedizierter Betriebssystemumgebungen für jede einzelne SecApp. Somit können auch SecApps betrieben werden, welche andernfalls aufgrund von Inkompatibilität nicht lauffähig wären. Um der Konkurrenz zwischen Anwendungen zu begegnen wird der Einsatz einer Entscheidungsinstanz vorgeschlagen. Diese Entscheidungsinstanz wird mit Kriterien versorgt anhand derer sie über die initiale Platzierung von Virtuellen Maschinen im Cluster, deren Laufzeitreplatzierung sowie über den lokalen Zugriff auf Knotenressourcen entscheidet. Diese Kriterien heißen **Policys**. Policys sind konfigurierbar und treffen Aussagen darüber, in welchem Maße Drittanwendungen eine bestimmte Ressource nutzen dürfen. Konkret ist eine Policy $P_A(X, Y)$ durch drei Parameter definiert. A ist eine Ressource, X ist ein Schwellwert und Y ein Zeitlimit. Durch die Definition einer Policy wird die maximale Zeit (Y) definiert für welche eine Drittapplikation an der Nutzung einer Ressource teilhaben darf, wenn die Auslastung der Ressource A den Wert X überschreitet. Beispielsweise ist die Policy $P_{CPU}(80, 10)$ zu interpretieren als: Das Sekundenmittel der CPU-Auslastung (eines Knotens) darf in maximal 10 aufeinanderfolgenden Messungen 80% überschreiten falls eine VM die Ressource CPU nutzt. Diese Vorgabe zwingt die Entscheidungsinstanz dazu, die CPU-Auslastung des Knotens möglichst unter 80% zu halten und im Notfall die CPU-Nutzung durch VMs zu beenden.

Basierend auf einer Reihe durch den Administrator definierter Policys entscheidet die Entscheidungsinstanz darüber, wann und wo im Cluster SecApps laufen dürfen und modifiziert wenn nötig deren Platzierung und Ressourcennutzung. Zur Abstraktion und uniformen Manipulation beliebiger SecApps werden diese durch sogenannte **Leases** repräsentiert. Ein Lease besteht aus einer Menge von

[5] Key Performance Indicator (KPI) ist ein in ITIL[4] vorgeschlagenes Konzept zur Spezifikation von Kriterien für die korrekte Funktionalität einer Anwendung.

VMs, Leases, Policys und Knoten begegnen kann. Das vorgeschlagene Konzept wurde plattformunabhängig in der Programiersprache Python umgesetzt. Als prototypische Virtualisierunglösung wurde Oracle VirtualBox eingesetzt. Plattformabhängige Komponenten zur Beschaffung von Ressourcenauslastungsstatistiken und zur lokalen Modifikation der Ressourcenutzung durch VMs wurden mit Linuxbordmitteln realisiert. Diese sind jedoch objektorientiert gekapselt und können bei Einsatz des Frameworks in alternativen Umgebungen ersetzt werden.

Empirische Resultate Im empirischen Teil der Arbeit wurden zwei Aspekte untersucht: Zum einen wurde überprüft, ob sich Policys eignen um die Anzahl der generierten SecApp Resultate, die CPU-Auslastung des Clusters und den Einfluß auf die MainApp (Interferenz) zu skalieren. Zum zweiten wurde getestet, ob es möglich ist, eine Policykonfiguration so zu finden, daß SecApps zusätzlich neben der HLT-Chain Applikation betrieben werden können ohne daß starke Interferenz auftritt. Für den ersten Aspekt wurden synthetische Anwendungen eingeführt, welche zeitlich und räumlich[8] variierende CPU bzw. Netzlasten im Cluster generieren. Der Einsatzzweck dieser synthetischen Anwendungen liegt darin, daß das Verhalten des Frameworks für verschiedenen Lastmuster, die unbekannte MainApps repräsentieren, überprüft werden kann. Es zeigte sich, daß sowohl die Anzahl der produzierten SecApp Resultate[9], als auch die zusätzliche CPU-Clusterauslastung mit den Policyparametern Schwellwert und Zeitlimit skalieren. Für die Interferenz wurde gezeigt, daß der Einfluß auf die MainApp mit sinkendem Schwellwert abnimmt. Auf Basis dieser Ergebnisse läßt sich sagen, daß Policys geeignet sind, um einen Kompromiß zwischen Nutzen von Drittanwendungen und erzeugter Interferenz für verschiedene Umgebungen und MainApps zu konfigurieren.
Konkret gezeigt wird die Nutzbarkeit des Ansatzes für den Anwendungsfall HLT-Chain Applikation. In Experimenten wurde das Software Framework mit geeigneter Policy-Konfiguration parallel zu einer laufenden HLT-Chain Applikation genutzt um Softwarecode zu kompilieren. Durch diese Parallelnutzung der Clusterumgebung wurde deren CPU Auslastung von 49% auf 79% erhöht, es wurden zusätzliche Ergebnisse berechnet und eine relevante Beeinträchtigung des Laufzeitverhaltens der HLT-Chain war nicht zu beobachten.

Zusammenfassung und Schlußfolgerung Die vorliegende Arbeit stellt ein Software Framework bereit, welches den Betrieb von Drittanwendungen neben einer zeitkritischen Anwendung in einem dedizierten Cluster ermöglicht. Kernproblem der Aufgabenstellung war sicherzustellen, daß die zeitkritische Anwendung nicht durch diese Drittanwendungen in ihrer Funktionalität beeinträchtigt wird. Die Kernideen zur Lösung dieses Problems sind *a*) die Isolation von Drittanwendungen durch Plattformvirtualisierung und *b*) die dynamische Anpassung der Ressourcennutzung durch VMs. Die Anpassung wird realisiert durch global initiierte VM-Manipulationen (Hot-Migration, Start/Stop, Suspend/Resume) und durch dezentrale Regelung des Ressourcenzugriffs durch VMs auf Basis von Betriebssystemmechanismen. Es wurde gezeigt, daß mittels bereitgestellter Konfigurationseinheiten (Policys) eine Abwägung zwischen Nutzen von Drittanwendungen und toleriertem Einfluß auf die zeitkritische Anwendung vorgenommen werden kann. Konkret wurde die Nutzbarkeit des Frameworks für die HLT-Chain Applikation gezeigt.
Die Arbeit erweitert den Stand der Forschung dahingehend, daß sie *a*) eine bisher nicht vorhandene Möglichkeit bietet Drittanwendungen sicher in einem dedizierten Cluster zu betreiben, *b*) hierzu verschiedene, bisher nicht kombinierte Methoden einsetzt und *c*) Ressourcenallokationsentscheidungen

[8]Räumliche Variation der Last bedeutet, daß die Auslastungen der Netzwerkschnittstelle bzw. der CPU der Knoten des Clusters unabhängig voneinander variiert werden.
[9]Als prototypische Drittanwendung wurde ein GCC Compilerbenchmark verwendet.

unter alleiniger Nutzung von Betriebssysteminformationen durchführt und damit einen generischen Ansatz für verschiedenste Umgebungen anbietet.

Literaturverzeichnis

[1] The ALICE Collaboration, "The ALICE Experiment at the CERN LHC," *J. Inst 3:S08002, 259pp*, (2008)

[2] Chakode R., Yenke B.O., and Mehaut J.F., "Resource Management of Virtual Infrastructure for On-demand SaaS Services," *Proc. International Conference on Cloud Computing and Service Science 2011, 352-361*, Netherlands, Noordwijkerhout, (2011)

[3] Liu C., Zhao Z., and Liu F., " An Insight into the Architecture of Condor - A Distributed Scheduler," *Proc. CNMT2009*, China, Wuhan, (2009)

[4] Information Technology Infrastructure Library (ITIL) Homepage, `http://www.itil-officialsite.com/`

[5] Jackson D.B., "Maui Scheduler: A Multifunction Cluster Scheduler," *Beowulf Cluster Computing with Linux*, MIT Press, Cambridge, MA, pp. 351-368, (2008)

[6] Linux Memory Management Information Homepage `http://linux-mm.org/OOM_Killer`

[7] Platform Load Sharing Facility, `http://www.platform.com/workload-management/high-performance-computing`

[8] Steinbeck T.M., Lindenstruth V., and Tilsner, H., "New Experiences with the ALICE High Level Trigger Data Transport Framework," *Proc. CHEP04*, Switzerland, Interlaken, (2004)

[9] VMware vSphere Homepage, `http://www.vmware.com/products/vsphere/mid-size-and-enterprise-business/overview.html`

[10] Zhu X. et al., "1000 Islands: An Integrated Approach to Resource Management for Virtualized Data Centers," *Proc. ICAC-09*, India, Tiruchirappalli, (2009)